POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Filtraggio e Monitoraggio di Funzioni a Livello Kernel

Relatore
prof. Antonio Lioy

Candidato
Andrea Traversa

Anno accademico 2017-2018

Sommario

Nel corso degli anni lo sviluppo tecnologico si è evoluto così tanto da risultare interessante in ogni settore, sia privato che aziendale, e che ha portato a una diffusione sempre più elevata di dispositivi computerizzati connessi spesso a sistemi importanti o comunque utilizzati per accedere a risorse anche molto private. Tutto ciò ha attirato l'attenzione di un numero sempre maggiore di criminali con le conoscenze informatiche necessarie (e non solo), i quali vedono questi dispositivi come ottime forme di guadagno. Per questo motivo applicare meccanismi di sicurezza informatica ai suddetti dispositivi è diventato sempre più importante. Purtroppo però non esiste una soluzione perfetta, utile quindi a bloccare ogni forma di attacco: esistono meccanismi di analisi statica di file, facilmente superabili con tecniche di cifratura; e ci sono soluzioni di analisi dinamica di file eseguibili, che però possono essere sconfitte da tecniche di anti-debug e simili. Viste le molteplici possibilità per superare queste tecniche, sono nati meccanismi di controllo di sistemi in tempo reale: esistono programmi che utilizzano determinate tecniche per cercare di capire automaticamente se un programma si sta comportando in modo pericoloso o meno e programmi il cui scopo è mettere in pratica regole precedentemente definite da un utente. Nel secondo caso si parla di soluzioni per monitorare, tracciare, filtrare e a volte addirittura gestire le funzionalità di un sistema nell'istante in cui esse vengono richiamate da una certa risorsa (lecita o meno, fidata o no, nuova o vecchia). Ci sono vari modi per effettuare questo tipo di sicurezza: esistono programmi per il monitoraggio della rete di un sistema, altri per la gestione della rete, altri ancora per il monitoraggio di funzioni e quasi nessuno per il filtraggio automatico di funzioni. Questo almeno prima dell'arrivo di eBPF.

eBPF (extended Berkeley Packet Filter) è infatti in grado di effettuare tutti i controlli sopra elencati definendo un linguaggio apposito, interpretabile dalla macchina virtuale presente nel kernel Linux. Grazie a esso è infatti possibile controllare in tempo reale il traffico di rete, stampando informazioni riguardanti ciò che sta succedendo, nel modo più ottimizzato e personalizzato possibile; gestire direttamente da dentro al kernel il networking definendo quali pacchetti sono leciti e quali no in base agli indirizzi IP, alle porte, ai protocolli, al contenuto dei pacchetti stessi, ..., arrivando a bloccare tutto ciò che non è desiderato; e si possono tracciare tutte le funzioni di un sistema nel momento in cui vengono chiamate, scoprendo chi le ha chiamate e con quali parametri con estrema facilità e velocità. L'unica limitazione che si ha è nel filtraggio di funzioni. eBPF da solo non può infatti effettuare questo tipo di operazione, ma si potrebbe usare il valore di ritorno del programma eBPF per indicare se una funzione va filtrata o meno e reagire di conseguenza (da un programma utente o, meglio ancora, da un modulo del kernel scritto appositamente). L'unico modo già esistente per filtrare funzioni con eBPF è tramite Seccomp, il quale permette di filtrare le system call chiamate da un certo processo in tempo reale, con l'ausilio di un apposito modulo del kernel.

Tutto ciò è applicabile in qualunque generico sistema Linux, dal più piccolo e insignificante al più importante. Ma non solo. Queste tecniche tornerebbero utili anche per esempio in sistemi che utilizzano container per offrire servizi verso l'esterno. Al giorno d'oggi infatti sono molti i server

che sfruttano tecniche di virtualizzazione e isolamento delle risorse tramite per esempio container e avere dei meccanismi di sicurezza in tempo reale applicabili a essi torna di nuovo molto utile. Proprio per questo scopo esiste un programma che sotto la superficie utilizza eBPF: Cilium, il quale permette a un utente di definire delle regole di gestione della rete di container in modo semplice e veloce e poi sfrutta eBPF per metterle in pratica. Ma la rete di un sistema non è l'unica a essere soggetta a attacchi: un file contenente un malware potrebbe superare i primi controlli di sicurezza; un processo potrebbe essere infettato in qualche modo; una vulnerabilità di un programma potrebbe essere sfruttata per eseguire del codice malevolo nel suo spazio di indirizzamento. Risulta quindi indispensabile effettuare controlli anche sulle funzioni interne di un sistema. Ma controllare ogni funzionalità di un sistema potrebbe non essere la scelta giusta perché risulta essere un'operazione lunga da mettere in pratica e poco prestante. Per questo motivo potrebbe essere più interessante tenere sotto controllo i processi in esecuzione nel sistema, anche quelli fidati, cercando di capire se stanno cercando di fare qualcosa di inaspettato. Questo può essere fatto tranquillamente e senza troppa difficoltà con eBPF; mentre con Seccomp si potrebbe arrivare addirittura a bloccare automaticamente un processo che chiama una system call non permessa o con parametri in ingresso illeciti.

Insomma, le possibilità sono tante e grazie a eBPF si ha ora una soluzione che permette tutti i possibili controlli, in modo altamente personalizzabile e tramite tecniche il più ottimizzate, stabili e sicure possibile. Non per altro, eBPF si è evoluto molto in fretta e tuttora continua a migliorare in ogni nuova versione del kernel e a essere utilizzato da sempre più programmi (non solo per scopi di sicurezza in tempo reale).

Indice

Capitolo 1

Controllare un sistema a livello kernel

1.1 Introduzione

Con il passare degli anni i sistemi computerizzati sono diventati fondamentali in ogni settore, sia privato che aziendale. In qualunque posto di lavoro, dal piccolo negozio dietro l'angolo alla grande e importante multinazionale, si trovano computer che facilitano almeno alcuni aspetti del lavoro di queste aziende. Si passa dal negoziante che utilizza il computer semplicemente per registrare le vendite, gli ordini e come supporto per i pagamenti elettronici fino a dispositivi che controllano autonomamente interi sistemi industriali. E nella vita privata della maggior parte delle persone l'utilizzo di dispositivi computerizzati è forse altrettanto importante: si è passati da un unico telefono fisso per un'intera famiglia a almeno un cellulare per ogni membro della famiglia, compresi i più giovani; da elettrodomestici controllabili solamente a mano a dispositivi gestibili anche a grandi distanze; ... E la stessa evoluzione è stata incontrata anche dalla rete Internet: con i sempre maggiori dispositivi connessi a essa, alcuni ventiquattro ore su ventiquattro, sette giorni su sette, si è passati da una rete per lo scambio di informazioni a una rete enorme che per molti è diventata insostituibile (e non solo per coloro che la utilizzano per guadagnare denaro).

Ovviamente però, più questi sistemi si diffondono, più diventano importanti in qualunque ambiente e più diventano target interessanti per criminali informatici. In fondo la maggior parte delle persone che utilizza questi dispositivi ha conoscenze limitate sul loro funzionamento e soprattutto si fida di essi e senza porsi troppi problemi li utilizza per comunicare informazioni, a volte molto private, o per accedere a servizi 'rischiosi' come conti bancari. Se si aggiunge che molti dispositivi connessi a Internet non sono protetti da password (o hanno credenziali di default o comunque facilmente indovinabili) e che tanti sistemi (anche industriali) non vengono aggiornati con la frequenza necessaria, è facile dedurre come mai gli attacchi informatici siano aumentati di numero e di criticità.

Si nota quindi come diventa fondamentale offrire dei meccanismi di difesa in grado di proteggere tutti questi sistemi, dal più importante al meno. Purtroppo però non esiste la difesa 'perfetta' (in nessun ambito, non solo quello informatico). Si può pensare infatti di proteggere la rete di un sistema con un firewall molto potente, ma un attacco potrebbe comunque aggirare il problema spingendo la vittima a connettersi a lui o a sfruttare protocolli non controllati dal firewall o a utilizzare proxy per reindirizzare il traffico, ... A questo punto si può pensare di non connettere un sistema a Internet (cosa certamente utile in sistemi che non necessitano di una connessione di rete), ma un attacco può comunque aggirare il blocco infettando un dispositivo portatile di chi utilizza questo sistema per poi passare al vero target da esso; oppure attaccando fisicamente il sistema stesso. Allora si può pensare di spegnere il sistema, ma un sistema spento non serve a niente (e può comunque essere soggetto a attacchi fisici). Ne si deduce quindi che siccome non si può puntare a una difesa assoluta, è fondamentale puntare a una difesa il più robusta e aggiornata possibile, composta magari da più elementi, ognuno con un compito ben definito.

1.2 La sicurezza in tempo reale

Uno dei meccanismi di difesa più utilizzati è il controllo di nuovi file non appena vengono salvati sul disco tramite la cosiddetta analisi statica. Questo tipo di controllo si basa sulla ricerca delle 'signature' (firme) di malware conosciuti: i byte di un nuovo file vengono letti e paragonati con un database contenente le firme dei vari malware. Se almeno una delle signature pericolose viene trovata all'interno del file, questo viene segnalato come malevolo. Ma ovviamente l'analisi statica non è abbastanza: un file può contenere del malware cifrato, che non contiene quindi la signature cercata e che verrà decifrato solo in fase di esecuzione; o del codice polimorfico (che cambia sé stesso man mano che viene eseguito); o delle istruzioni precedentemente modificate tramite un polymorphic engine (un programma che permette di cambiare le istruzioni in input sia come formato che come ordine mantenendo però lo stesso risultato finale); ... Per questo motivo all'analisi statica segue generalmente un'analisi dinamica: la prima volta che un nuovo file viene eseguito, esso viene prima mandato in esecuzione all'interno di una sandbox (un ambiente virtualmente isolato dal vero sistema), viene controllato al suo interno e solo se passa i controlli viene salvato nella lista dei file fidati e ne viene permessa l'esecuzione nel sistema reale. Anche questo meccanismo però ha i suoi difetti. Prima di tutto, un malware potrebbe essere contenuto in un file non eseguibile, ma attivarsi comunque perché quel formato di file permette di definire del codice al suo interno che verrà eseguito dal programma con cui verrà aperto o facendo leva su una vulnerabilità del programma usato per leggere questo tipo di file, cosa che ovviamente non attiva nemmeno l'analisi dinamica. Poi se anche questa analisi non fa altro che cercare signature, è possibile superarla con le stesse tecniche usate per superare l'analisi statica. Per questo motivo l'analisi dinamica si è evoluta oltre alla ricerca delle firme in modo tale da cercare anche di dedurre se un programma è pericoloso o meno per il sistema a seconda di quali funzioni chiama arrivando negli ultimi anni a utilizzare persino tecniche di machine learning.

Ma anche questo potrebbe non essere abbastanza. Un primo problema a questo punto è che cosa deve essere segnalato come pericoloso e che cosa no. Per esempio, si potrebbe pensare che un programma che sfrutta cifratura/decifratura del codice o tecniche polimorfiche o di anti-debug in fase di esecuzione sia per forza malevolo. Cosa non vera visto che molti software privati e videogiochi utilizzano queste tecniche per complicare la vita a coloro che cercano di craccarli. Oppure si potrebbero bloccare tutti i programmi che aprono un processo e lo modificano (tecnica che potrebbe essere usata da un malware per eseguire operazioni illecite da dentro un processo fidato anziché da sé stesso); ma anche in questo caso si potrebbe sbagliare. Il secondo problema riguarda invece la durata dell'analisi. L'esecuzione di un processo all'interno di una sandbox non può essere totale nel caso di processi lunghi o si ritarderebbe troppo l'esecuzione del processo nel vero sistema: alcuni programmi potrebbero dover eseguire operazioni per ore se non giorni all'interno della sandbox prima di attivarsi nel sistema reale, cosa ovviamente improponibile. Per questo motivo l'analisi dinamica generalmente viene chiusa dopo un po' di tempo (quando non viene direttamente terminata dall'utente per non dover aspettare) a volte aumentando il clock del sistema virtuale in modo da eseguire un numero maggiore di istruzioni in quel lasso di tempo. Questo porta ulteriori problemi: di alcuni programmi viene analizzata solo la prima parte (un malware potrebbe quindi aspettare un certo periodo di tempo prima di fare qualcosa di illecito); malware più intelligenti potrebbero fare calcoli sulle tempistiche dell'esecuzione di certe istruzioni e se i risultati non sono quelli aspettati, supporre di trovarsi in una sandbox con tempo accelerato e chiudersi (questo porterebbe l'analisi a credere che il programma sia terminato senza eseguire alcuna operazione pericolosa). Senza contare tutte le tecniche di anti-debug e anti-analisi che potrebbero sconfiggere una qualunque sandbox, come il self-debug (per evitare che qualcun altro usi tecniche di debug su di esso), l'effettuare controlli frequenti alla ricerca di debugger attivi su di sé, l'enumerazione dei processi attivi (per capire se si è in un sistema isolato o meno), il calcolare la differenza di tempo tra la data reale e quella del sistema (alcune sandbox usano date statiche), l'analisi di richieste di rete reali/fittizie e delle relative risposte (spesso le sandbox rispondono sempre in modo positivo alle richieste di rete dei processi che stanno analizzando per controllare come si comporteranno una volta ricevuta la risposta attesa), ...

Per questo motivo, dopo l'analisi statica e quella dinamica, si è arrivati a meccanismi di controllo in tempo reale. Grazie a essi, non si vanno a controllare unicamente i nuovi file, ma tutti i processi, anche quelli fidati, in ogni momento della loro esecuzione. Facendo attenzione a non diminuirne

troppo le prestazioni, l'obiettivo è di controllare l'esecuzione in tempo reale di tutto ciò che sta succedendo nel sistema, nell'esatto momento in cui succede, e cercare di capire se si tratta di operazioni lecite o meno. A questo punto è chiaro che diventa possibile analizzare anche tecniche di attacco che non sarebbero state individuate con i meccanismi precedenti (per esempio un attacco attivato da remoto).

1.3 Le funzioni a rischio

Per controllare un sistema in tempo reale al meglio, è fondamentale scegliere nel miglior modo possibile cosa analizzare e come farlo. Ovviamente tenere sotto controllo solamente le funzioni che si ritengono rischiose non è abbastanza visto che non è facile definire cosa è rischioso e cosa no. Per esempio si potrebbe pensare che un attacco da remoto abbia come obbiettivo l'ottenimento di una shell di sistema e decidere di monitorare questo evento. Ma come per molte altre cose, una shell può essere ottenuta in vari modi e inoltre esistono diversi tipi di shell (senza contare che un attaccante potrebbe scaricarne una dal web e eseguirla come se fosse un programma totalmente diverso). Si può quindi pensare di monitorare tutti i modi in cui potrebbe essere eseguita una shell (supponendo che sia possibile), ma anche questo non servirebbe a nulla se l'attaccante non sfruttasse la shell di sistema per ottenere ciò che gli interessa. Un esempio opposto potrebbe essere l'apertura di file: di per sé sembra un'operazione innocua, ma se un attaccante apre un file di sistema o uno che contiene delle password o delle informazioni private dell'utente, la cosa può diventare pericolosa; o se apre file importanti, li modifica o li cifra, non è certo una situazione ideale; o magari apre un file con un certo programma e sfrutta una sua vulnerabilità per ottenere qualcos'altro.

A questo punto è chiaro come sia necessario tenere sotto controllo tutte le funzionalità di un sistema, anche quelle che a prima vista sembrano meno importanti, perché un attacco può arrivare da una qualunque di esse, a volte anche dalle più inaspettate, facendo però attenzione anche alle prestazioni del sistema stesso. Prendiamo per esempio un server che offre un servizio pubblico su una porta e possiede servizi privati su altre porte. Mantenere dei controlli sulle funzioni interne del sistema è certamente utile, ma l'ideale è avere anche dei controlli sui pacchetti di rete: non appena qualcuno che non ha i permessi di accedere ai servizi privati prova a connettersi a una delle porte dedicate a essi, lo si scopre subito e si può reagire di conseguenza (magari bloccando la connessione e evitando di dover analizzare ciò che verrà richiesto al sistema); e se qualcuno si connettesse sulla porta pubblica e facesse delle richieste illecite, di nuovo, lo si potrebbe scoprire il prima possibile ottimizzando la reazione.

In conclusione, molte funzioni, la maggior parte dei processi, tante funzionalità interne e esterne di un sistema sono a rischio e nell'offrire sicurezza in tempo reale bisogna tenerne conto. Ma allo stesso tempo analizzare tutte le chiamate locali dei processi in esecuzione, le funzioni di libreria, le system call, le funzioni interne al kernel, i pacchetti di rete, i socket, le connessioni, ... potrebbe essere esagerato. Bisogna quindi trovare un equilibrio tra ottimizzazione e sicurezza, spingendosi magari più da una parte o dall'altra a seconda dei casi.

1.4 Livello kernel rispetto a livello utente

Nel realizzare sicurezza in tempo reale si può pensare di scrivere un programma di livello utente che analizzi le funzionalità che si è interessati a tenere sotto controllo (o di usarne uno tra quelli che già esistono). Ovviamente il modo in cui vengono analizzate queste funzionalità dipende da quale tecnica si vuole utilizzare e di quali funzionalità si tratta. Per esempio, nel caso dell'analisi del comportamento di un processo in esecuzione, si può decidere di attaccarsi a esso con un debugger programmato per effettuare in modo automatico determinati controlli. Nel caso del controllo della rete di un sistema, si potrebbero attivare determinati hook (punti di un sistema in cui è possibile registrare una funzione di callback, che verrà chiamata ogni qual volta qualcuno passa in quel punto) in uno o più punti dello stack di rete e effettuare i controlli da lì. O ancora, se si volesse analizzare chi ha chiamato una determinata funzione nel momento della chiamata stessa, si potrebbe attivare l'hook relativo alla data funzione. E così via.

Tutte queste soluzioni sono molto valide, ma mostrano immediatamente qual'è il principale svantaggio nel praticare questo tipo di sicurezza a livello utente: un debugger è in grado di analizzare un programma chiedendo al kernel di effettuare determinate operazioni su di esso; e gli hook, che siano di livello rete o su una qualunque funzione di un sistema, non vengono solo attivati dal kernel, ma anche gestiti da esso. Questo implica che tutte queste tecniche iniziano con il livello utente che chiede qualcosa al kernel, il quale gli risponde; ogni volta che scatta un certo evento, il kernel se ne accorge e avverte l'utente, il quale esegue le azioni predefinite (magari chiedendo il supporto del kernel) e poi avverte il kernel che ha finito; e il ciclo ricomincia. Si nota subito come lo scambio di informazioni tra i due livelli sia elevato. Ne si deduce quindi che una soluzione migliore sarebbe effettuare questi controlli direttamente dentro al kernel: si registrano i controlli da attivare, scattano gli eventi, si effettuano i controlli, si reagisce di conseguenza, ... tutto nello spazio dedicato al kernel, tutto il più in fretta possibile.

Prima di eBPF (la tecnologia che viene analizzata in questa tesi), effettuare monitoraggio e filtraggio di pacchetti di rete e il monitoraggio delle chiamate a funzione a livello kernel era già molto potente (anche se meno programmabile). Per quanto riguarda invece il filtraggio di funzioni, esisteva una buona soluzione che è stata migliorata dall'arrivo di eBPF stesso (della quale si parlerà nei capitoli successivi).

Capitolo 2

Tecniche di monitoraggio e filtraggio

2.1 Introduzione

Come anticipato in precedenza, esistono vari modi per applicare controlli in tempo reale sulle funzionalità di un sistema, che differiscono a seconda di quali funzioni si vuole analizzare. Nel monitorare (e magari filtrare) l'esecuzione di un programma, un debugger automatizzato potrebbe settare breakpoint nei punti d'interesse e eseguire determinate operazioni ogni volta che uno di essi viene incontrato dal programma. O si potrebbero abilitare determinati hook associandoli a funzioni di callback che eseguano determinate azioni. Un programma da analizzare potrebbe anche essere eseguito dopo aver iniettato nel contesto del suo processo una libreria dinamica contenente le funzioni di controllo che verranno chiamate al posto delle funzioni da controllare (le quali chiameranno queste ultime al proprio interno). Nel monitorare la rete di un sistema invece, si potrebbe semplicemente attivare un socket raw (in grado di leggere qualunque pacchetto di qualunque protocollo) su un'interfaccia e analizzare ogni pacchetto in transito su di essa. Se invece si volesse anche filtrare (o comunque gestire la rete in qualche modo), si potrebbero attivare hook di rete o si potrebbe direttamente programmare lo scheduler dei pacchetti di rete.

Insomma, le possibilità sono tante e diversi metodi potrebbero esistere per diversi sistemi operativi. Di seguito comunque verranno analizzate le soluzioni (precedenti a eBPF, ma ancora utilizzate) che sembrano essere più interessanti nei rispettivi campi di utilizzo, prendendo in considerazione unicamente quelle che effettuano le operazioni di analisi direttamente dentro al kernel del sistema (dove possibile). Inoltre ci si concentrerà solamente su sistemi Linux.

2.2 Monitoraggio della rete

Se si ha come obiettivo l'analisi in tempo reale di ciò che sta transitando su una o più interfacce di rete in entrambe le direzioni (ingresso e uscita), con magari la possibilità di selezionare solamente un sottoinsieme dei pacchetti in transito, si può parlare di monitoraggio della rete di un sistema.

Se l'obiettivo è fare ciò con l'ausilio di programmi già esistenti, le scelte più interessanti sono Wireshark e Tcpdump. Il primo è un programma grafico che possiede anche una versione da riga di comando (Tshark), il secondo invece è unicamente utilizzabile da riga di comando. Entrambi danno la possibilità all'utente di ricevere a video delle informazioni in tempo reale riguardanti i pacchetti in transito sull'interfaccia scelta permettendo anche di definire un filtro su quali pacchetti visualizzare. Questo filtro altro non è che una stringa contenente parole chiavi come nomi di protocolli di rete, indirizzi MAC, indirizzi IP, numeri di porta, ... e operazioni logiche. Sotto la superficie, il primo utilizza varie tecnologie per la cattura e l'analisi dei pacchetti mentre il secondo utilizza unicamente una di queste: BPF (Berkeley Packet Filter), una tecnologia che permette di leggere i pacchetti in

transito su di una certa interfaccia più velocemente rispetto a altri metodi (per esempio tramite l'uso di generici socket). Da sottolineare è il fatto che BPF non è semplicemente un programma o una libreria di livello utente o kernel, ma un vero e proprio linguaggio di programmazione. Infatti, all'interno del kernel Linux (dalla versione 2.1.75 in avanti) esiste una macchina virtuale il cui compito è proprio quello di interpretare programmi scritti in BPF. Nel caso del monitoraggio dei pacchetti di rete, è possibile attaccare un programma BPF a un socket per decidere se i pacchetti in transito su di una certa interfaccia di rete devono essere inviati o no al livello utente. Un'analisi più approfondita verrà affrontata nel capitolo 3.

Nel caso in cui si volesse scrivere da zero un programma con questo stesso obiettivo, lo si potrebbe fare sia a livello utente che a livello kernel. Nel primo caso, l'ideale resta sempre appoggiarsi a BPF, meglio ancora se gestito tramite la libreria Libpcap. Questa libreria infatti permette a un programma utente di interfacciarsi con molta facilità a BPF definendo funzioni per semplificare operazioni come attaccarsi a un'interfaccia di rete, caricare un filtro nella macchina virtuale, avviare e fermare l'analisi, ... e soprattutto definire il filtro tramite una stringa (dello stesso formato descritto in precedenza) che verrà tradotta automaticamente nel relativo programma BPF. Nel caso di programmazione a livello kernel invece si potrebbero attivare determinati hook di rete e associargli una funzione di callback il cui scopo potrebbe essere la stampa delle informazioni ottenute dal pacchetto corrente. Ogni volta che un pacchetto passa per lo stack di rete, l'hook si attiva e la funzione viene chiamata. Ovviamente la soluzione di livello utente risulta più semplice e veloce da programmare.

2.3 Filtraggio di pacchetti di rete

Quando ci si vuole spingere oltre al semplice monitoraggio della rete di un sistema, quando si vuole anche interagire con i pacchetti che si sta analizzando in tempo reale, magari per gettarne via alcuni o per reindirizzarli o per cambiare determinati campi del pacchetto, ... allora si parla di filtraggio, e più in generale, di gestione della rete.

In questo caso i programmi più utilizzati sono Tc e Iptables. Tc è un programma da riga di comando che permette di gestire direttamente dal livello utente il Traffic Control, l'insieme dei meccanismi del kernel Linux usati per l'invio e la ricezione dei pacchetti di rete. Più nel dettaglio, Tc è in grado di gestire la velocità di trasmissione dei pacchetti, di riordinare pacchetti in uscita, dargli priorità e dividerli in code, di settare delle regole per il traffico in arrivo e di buttare via pacchetti indesiderati (sia tra quelli in ingresso che in uscita). Infine, tc permette anche di sostituire le regole di filtraggio o le azioni predefinite dal Traffic Control stesso con programmi BPF: lo scopo del programma in questo caso sarebbe di analizzare il pacchetto e dire al kernel se esso fa parte della coda corrente (se installato in ingresso a una coda) o quale azione eseguire su di esso (se installato in uscita). Per quanto riguarda iptables, esso è un programma da riga di comando che permette di definire delle regole per la configurazione delle tabelle del firewall interno al kernel Linux. Queste regole sono definite da una prima parte che descrive il tipo di pacchetto che si vuole selezionare e da una seconda parte che definisce quale azione eseguire su quei pacchetti. Iptables è in grado di analizzare pacchetti in base all'interfaccia dalla quale arrivano o verso la quale sono diretti, all'indirizzo IP sorgente o destinazione, alla porta sorgente o destinazione, al protocollo di livello 4 usato, ... e permette di eseguire azioni di modifica dei pacchetti, di reindirizzamento, di rimozione, ... eccetera. Un'osservazione da fare è che è possibile scrivere la prima parte delle regole (la selezione dei pacchetti) tramite codice BPF anziché tramite le parole chiave definite da iptables stesso.

Entrambe le soluzioni descritte però permetto la gestione del traffico di rete unicamente ai livelli 3 e 4. Se si volessero definire delle regole di livello 7 bisognerebbe appoggiarsi a qualcos'altro. Purtroppo però non sembrano esistere soluzioni ideali a livello kernel. Infatti quello che generalmente viene fatto è utilizzare un WAF (Web Application Firewall). Un esempio è modsecurity: un modulo del server web Apache che permette di definire delle regole di filtraggio sul contenuto delle richieste HTTP, sul metodo utilizzato, sulla risorsa richiesta, sugli header, ... e persino sull'identità del client che ha effettuato la richiesta. Ogni volta che un client fa una richiesta, essa viene analizzata e viene inviata una risposta valida al client solamente se è permesso dalle regole.

Se si fosse invece interessati a scrivere un programma per il filtraggio (o più in generale la gestione) del traffico di rete, si può pensare di scriverne uno per la gestione del Traffic Control (ma a questo punto tanto vale usare Tc, a parte in casi molto particolari). Una seconda soluzione, certamente più interessante, ma complessa, è la scrittura di un modulo del kernel che attivi determinati hook (magari quelli gestiti da iptables) e gli associ determinate funzioni di analisi e le rispettive azioni. Il funzionamento a questo punto sarebbe molto simile a quello del modulo di iptables: quando arriva un pacchetto l'hook si attiva, la funzione associata lo analizza e mette in pratica l'azione predefinita.

2.4 Monitoraggio di chiamate a funzione

Se si vuole analizzare in tempo reale il comportamento di un sistema attraverso quali funzioni vengono chiamate, di livello utente e/o di livello kernel, da chi vengono chiamate e come vengono chiamate, e quindi con quali parametri, allora si parla di monitoraggio di funzioni.

Di programmi per fare questo ne esistono vari, ma i più interessanti sembrano essere Trace-cmd e Perf. Il primo è un programma da riga di comando che permette di gestire in modo più semplice e intuitivo Ftrace, l'utilità dei sistemi Linux per il tracciamento delle funzioni di livello kernel. Con tracciamento si intende la scoperta in tempo reale di quali funzioni vengono chiamate, come e da chi (è quindi un sinonimo di monitoraggio). Sotto la superficie, Ftrace è in grado di analizzare le funzioni in tempo reale abilitando i molteplici tracepoint sparsi per il kernel Linux (la cui definizione si trova nella cartella `/sys/kernel/debug/tracing/events/<categoria_evento>/<evento>`, raggiungibile unicamente dall'utente root). Essi sono pezzi di codice contenuti nei vari moduli che formano il kernel ai quali si può attaccare una funzione di callback che verrà chiamata ogni qual volta qualcuno passa da quel punto e che riceverà in input una struttura contenente i parametri ritenuti indispensabili per il debug di quel modulo/funzione. Una volta terminata l'esecuzione della funzione di callback, viene ripresa la normale esecuzione del modulo/funzione che si sta monitorando (una descrizione più dettagliata verrà affrontata nel capitolo 3). Per la sicurezza in tempo reale si può quindi pensare di usare Trace-cmd (o direttamente Ftrace) per monitorare in tempo reale qualunque funzionalità di livello kernel (dall'iterazione del sistema con il disco, al controllo delle prestazioni delle cpu, al contenuto e gestione della memoria, dello scheduler, ...), con la possibilità di definire un filtro che riduca il numero di eventi da monitorare, e ricevere a video le informazioni ottenute, stampate in un formato predefinito. Per quando riguarda Perf invece, esso è di nuovo un programma da riga di comando per l'analisi delle funzioni di un sistema in tempo reale (in questo caso non solo di livello kernel); è nato come un programma per l'analisi delle prestazioni di un sistema (appoggiandosi ai registri hardware per il conteggio degli eventi hardware), ma si è evoluto a tal punto da poter tracciare funzionalità tramite l'ausilio di tracepoint e di probe. Questi ultimi sono un altro meccanismo che il kernel Linux possiede per l'analisi delle funzionalità di un sistema. In questo caso non si tratta di codice predefinito, ma di vero e proprio debug: quando si vuole analizzare una funzione tramite un probe si può chiedere al kernel di registrare una certa funzione di callback a un certo indirizzo (o simbolo); il kernel setta un breakpoint a quell'indirizzo e vi associa la funzione ricevuta. Ogni volta che l'esecuzione di un programma passa per quel punto, il breakpoint si attiva e la funzione associata viene eseguita ricevendo in ingresso lo stato dei registri in quel dato istante. Ovviamente quando la funzione ritorna, l'esecuzione riprende da dove si era interrotta (di nuovo, maggiori informazioni verranno presentate nel capitolo 3). Grazie ai probe e ai tracepoint, quindi, si potrebbe usare perf per monitorare in tempo reale una qualunque chiamata a funzione di livello utente o kernel con i parametri che le sono stati passati; o i valori dei registri e dello stack di un qualunque punto del sistema; o il valore di ritorno di una qualunque funzione; ... insomma, si potrebbe tenere sotto controllo un intero sistema e ricevere a video un resoconto di ciò che sta accadendo.

Se invece si volesse scrivere un programma per il monitoraggio delle funzionalità di un sistema (senza dover ricompilare il kernel), l'ideale è sempre appoggiarsi ai probe e ai tracepoint. Per poterlo fare bisogna scrivere un modulo del kernel che registri i probe dove necessario o che attivi i tracepoint a cui si è interessati e che definisca le relative funzioni di analisi. Le informazioni catturate dal modulo potrebbero essere inviate al livello utente per la visualizzazione o gestite direttamente dentro al kernel.

2.5 Filtraggio di funzioni indesiderate

Quando si desidera avere una qualche forma di reazione in tempo reale (e magari automatizzata) a ciò che viene scoperto durante il monitoraggio delle funzioni di un sistema, che vada oltre al semplice stampare un messaggio a video, si ha il filtraggio, o più in generale, la gestione delle chiamate a funzione. Purtroppo però non sembrano esistere programmi ideali per questo tipo di compito. L'unica opzione sembrerebbe quindi quella di scrivere un programma da sé. Nel caso in cui il filtraggio debba essere eseguito all'interno di un sistema specifico (e conosciuto), si può pensare di scrivere questo programma in modo tale da poter gestire delle regole predefinite. Questo ovviamente non funzionerebbe se questo stesso programma dovesse essere usato in diversi sistemi perché non si può sapere a priori quali funzioni hanno il permesso di essere chiamate o meno, da chi possono essere chiamate o con quali parametri (in ingresso o in uscita). In questo secondo caso, quindi, l'ideale sarebbe scrivere un programma che permetta all'utente di definire delle regole di filtraggio che verranno trasformate nel codice vero e proprio. In entrambi i casi, l'ideale è appoggiarsi a Seccomp o a probe/tracepoint.

Seccomp è un'utilità incorporata nel kernel Linux (fin dalla versione 2.6.12) che permette a un programma di definire un filtro sulle system call che vuole permettere a sé stesso e ai suoi figli. Una volta che un programma ha abilitato questa tecnologia (tramite apposite funzioni), il kernel analizza in tempo reale ogni sua chiamata a una system call e tutti i parametri in ingresso e a seconda del filtro definito dal programma stesso, decide se lasciare passare la system call, se uccidere il processo o terminare il thread che ha effettuato la chiamata, ... (maggiori informazioni verranno fornite nel capitolo 5). Per la sicurezza in tempo reale, si può quindi pensare di scrivere un programma che inizializzi un filtro seccomp per un dato eseguibile e esegua esso nel proprio contesto o come processo figlio. A questo punto si avrà un totale controllo automatizzato sulle chiamate a system call di quel processo. Ovviamente bisognerebbe generare filtri diversi per programmi diversi e il filtraggio avverrebbe unicamente sulle system call (e/o sui loro parametri) e solo quando a chiamarle è uno dei processi eseguiti dal programma scritto.

Per poter analizzare una qualunque funzione di livello utente o kernel di un sistema, senza dover definire quali processi controllare e quali no, si potrebbero utilizzare probe o tracepoint. Si potrebbero infatti registrare probe sulle funzioni da controllare o attivare determinati tracepoint e associarvi funzioni che oltre a monitorare il comportamento della funzione ne modifichino l'esecuzione. Come definito nella sezione precedente però, l'unico modo per fare questo è scrivere un modulo del kernel cosa che rende la stesura di un programma del genere più complesso rispetto alla soluzione precedente, ma dà maggiore personalizzazione delle funzioni di controllo. Si potrebbe infatti pensare di spingersi oltre al filtraggio delle chiamate a funzione indesiderate arrivando al reindirizzamento: se una funzione che non dovrebbe essere chiamata venisse chiamata lo stesso, anziché uccidere il processo che l'ha effettuata, si potrebbe chiamare un'altra funzione dal suo interno, per l'esecuzione di una qualche funzionalità di supporto, o cambiare i parametri in ingresso o in uscita per studiare il comportamento di un attaccante anziché sbarragli la strada fin da subito.

Capitolo 3

eBPF - extended Berkeley Packet Filter

3.1 Cenni storici

Come dice il nome stesso, eBPF (extended Berkeley Packet Filter [1]) è un'evoluzione e estensione di (c)BPF (classic Berkeley Packet Filter) che, come accennato nel capitolo 2, era una tecnologia che permetteva di leggere i pacchetti in transito su di una certa interfaccia di rete permettendo anche di definire un filtro (sotto forma di programma BPF appunto) su quali pacchetti si era interessati a inviare a livello utente. Nato da un'idea di Steven McCanne e Van Jacobson (nel 1993) e visto per la prima volta nel kernel 2.1.75 (del 1997) il suo funzionamento è rappresentato in figura 3.1 e descritto brevemente in seguito.

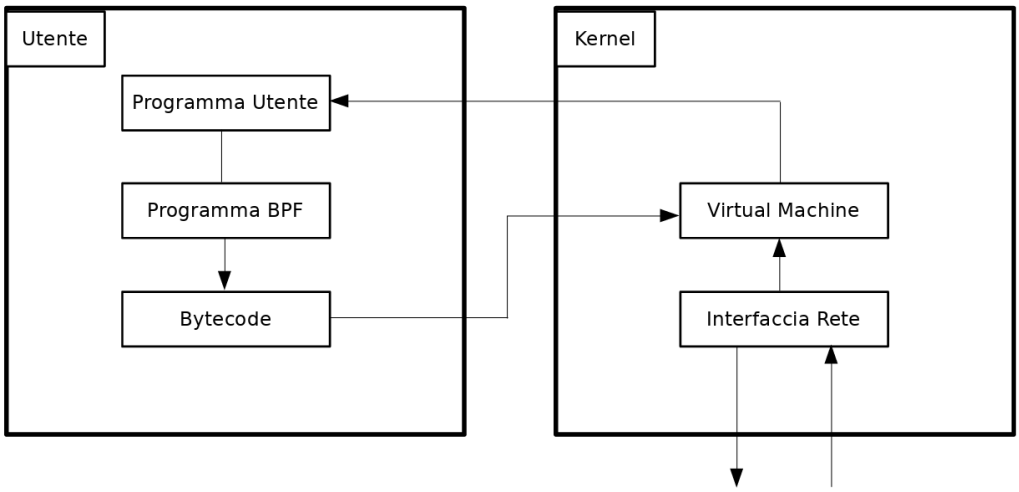

Figura 3.1. Schema funzionamento BPF.

Quando un programma utente desiderava utilizzare BPF per il monitoraggio della rete, la prima cosa che doveva fare era aprire un socket raw sull'interfaccia di rete che voleva osservare. Successivamente doveva definire un programma BPF (di cui si parlerà meglio a breve) che fosse in grado di decidere, ricevuti i byte di un pacchetto di rete in ingresso, se il pacchetto corrente dovesse essere inoltrato al livello utente o meno. A questo punto, il programma utente compilava il programma BPF in bytecode (codice non eseguibile, ma interpretabile dalla macchina virtuale presente nel kernel) e lo inviava al livello kernel, il quale si occupava di inserirlo all'interno della macchina virtuale. Infine il programma utente si metteva in ascolto sul socket precedentemente aperto. Ogni volta che un pacchetto transitava sull'interfaccia sotto controllo, il socket lo sentiva,

il kernel generava una copia del pacchetto e la inviava alla macchina virtuale, la quale eseguiva le istruzioni precedentemente caricate. A seconda del risultato dell'esecuzione, la copia del pacchetto corrente veniva inviata al livello utente o veniva scartata. Proprio grazie a questo meccanismo, BPF risultava più veloce (e quindi migliore) di altre soluzioni: i pacchetti venivano inviati al livello utente solo se superavano il filtro specificato, quindi solo quando necessario, diminuendo così il più possibile lo scambio di informazioni tra i due livelli. Un'osservazione da fare è il fatto che alla macchina virtuale veniva unicamente passata una copia del pacchetto: sia che il pacchetto venisse inoltrato al livello utente, sia che venisse gettato via, il traffico di rete non veniva influenzato in alcun modo. Questo dimostra come mai BPF potesse essere solamente usato per il monitoraggio della rete e non per la sua gestione o per il filtraggio di pacchetti (a meno che non ci si appoggiasse a qualche altra tecnologia).

Solo nel 2013 si cominciò a parlare di eBPF, da un'idea di Alexei Starovoitov, integrandolo per la prima volta nella versione 3.15 del kernel Linux. Inizialmente altro non era che una migliore versione di BPF [2]. Permetteva quindi, come spiegato nel capitolo precedente, di monitorare in tempo reale il traffico di rete di un sistema installando un socket raw sull'interfaccia a cui si era interessati e analizzando ogni pacchetto passante per quel punto tramite un programma scritto in linguaggio eBPF e interpretato dalla macchina virtuale interna al kernel. Le migliorie di eBPF derivavano, all'inizio, unicamente dalle ottimizzazioni e dai potenziamenti apportati al linguaggio BPF stesso (e di conseguenza dall'aggiornamento della macchina virtuale). Infatti, come si può notare dalla tabella 3.1 mentre BPF utilizzava registri a 32 bit, un registro temporaneo, uno utilizzato come registro principale e un vettore di sedici posizioni utilizzato come stack (oltre al program counter), eBPF ha undici registri a 64 bit (ciascuno utilizzabile anche come due registri da 32 bit) e un vero e proprio stack (sempre oltre al program counter). Entrambi i linguaggi hanno istruzioni di 64 bit di lunghezza, ma eBPF ha istruzioni strutturate in modo da avere maggiore ottimizzazione in fase di esecuzione (permettendo per esempio lo spostamento diretto di dati tra due registri qualunque e di avere meccanismi di salto 'jump or fall through') e più funzionalità (per esempio in eBPF è permesso fare salti all'indietro e quindi cicli, anche se ancora molto limitati). Sono state inoltre aggiunte varie istruzioni, tra cui l'istruzione call, e sostituite quelle non più utili (BPF aveva istruzioni di load e store nel registro primario e nel registro temporaneo separate, mentre eBPF ha istruzioni di load e store generiche, con anche la possibilità di gestire direttamente tutti i 64 bit dei registri). Infine, entrambi i linguaggi hanno un numero massimo di istruzioni peri a 4096, ma eBPF permette di sfruttare le 'tail calls' che servono per chiamare un programma eBPF dall'interno di un altro programma eBPF (permettendo quindi di concatenare l'esecuzione di più programmi e 'superare' il limite massimo di 4096).

	BPF	*eBPF*
Registri	32 bit: A, X, M [16], (pc)	64 bit (sotto-registri da 32 bit): R0-R10, stack, (pc)
Istruzioni	64 bit (u16:code, u8:jt, u8:jf, u32:k) load, store, branch, alu, miscellaneous, return max 4096	64 bit (u8:code, u8:dst_reg, u8:src_reg, s16:off, s32:imm) set BPF + 64 bit load/store/alu, calls max 4096 + 'tail calls'
Salti	forward	forward/backward

Tabella 3.1. Linguaggi BPF-eBPF a confronto.

Con il passare del tempo, eBPF si è evoluto ulteriormente (ed è tuttora in evoluzione) sia dal punto di vista del linguaggio che delle funzionalità. Per quanto riguarda il linguaggio sono state create direttamente dentro al kernel molte funzioni che facilitano la programmazione [3]. Un esempio potrebbe essere il caso in cui si utilizzasse eBPF per intercettare tutti i pacchetti in transito su di una certa interfaccia e reindirizzare quelli di un certo tipo (tramite la modifica di determinati campi dei pacchetti) o per svolgere una funzione di NAT. Una volta modificati i

pacchetti, bisognerebbe ricalcolare a mano il checksum e sostituirlo a quello vecchio. Dalla versione 4.1 del kernel invece, basta chiamare le funzioni bpf_l3_csum_replace (...) e bpf_l4_csum_replace (...) rispettivamente per il calcolo e la sostituzione dei checksum di livello 3 e 4. Per quanto riguarda le funzionalità invece si ha deciso di spostarsi dal semplice monitoraggio della rete fino a arrivare al monitoraggio (e in alcuni casi filtraggio) di tutte le funzioni di un sistema [3]. Dalla versione 4.1 del kernel per esempio è stata aggiunta la possibilità di gestire il Traffic Control direttamente con eBPF e di attivare probe a cui associare programmi eBPF. Dalla 4.7 si ha ottenuto anche la possibilità di attivare tracepoint e analizzarli direttamente tramite programmi eBPF. Dalla 4.8 è stato aggiunto un ulteriore metodo per la gestione del traffico di rete. E molto altro ancora. Avendo così tante possibili funzionalità, i programmi eBPF hanno acquisito un identificativo che ne caratterizza lo scopo: c'è quello per il monitoraggio della rete tramite socket, quello per l'analisi di funzioni tramite probe, quello per il debug di tracepoint e così via. Ovviamente i programmi eBPF sono anche stati aggiornati in modo tale da ricevere diversi parametri in input per diversi tipi di programma (a differenza di BPF che poteva unicamente ricevere dati specifici al pacchetto da analizzare, nel caso del networking, o i parametri in input alla system call da studiare, nel caso di seccomp). Per quanto riguarda i programmi di livello utente che un tempo utilizzavano BPF al loro interno (come Tcpdump, per esempio) e la libreria Libpcap, in alcuni casi sono stati aggiornati e sono ora in grado di generare direttamente programmi eBPF (come Seccomp per esempio), altri invece continuano a funzionare esattamente come prima, ma nel momento in cui il programma BPF generato a livello utente viene passato al kernel, quest'ultimo lo traduce automaticamente in linguaggio eBPF. Infine è stato aggiunto un controllore a fianco della macchina virtuale chiamato verifier il cui scopo è analizzare un programma eBPF prima che venga iniettato nella macchina virtuale per essere certi che ciò che verrà inserito sia sicuro per il sistema e stabile.

3.2 Schema di funzionamento

Ricapitolando, eBPF può essere definito una tecnologia in grado di analizzare, filtrare e tracciare eventi e funzioni direttamente a livello kernel. Il suo funzionamento è rappresentato in figura 3.2 e descritto brevemente in seguito (un approfondimento delle varie parti raffigurate verrà fatto nelle sezioni successive).

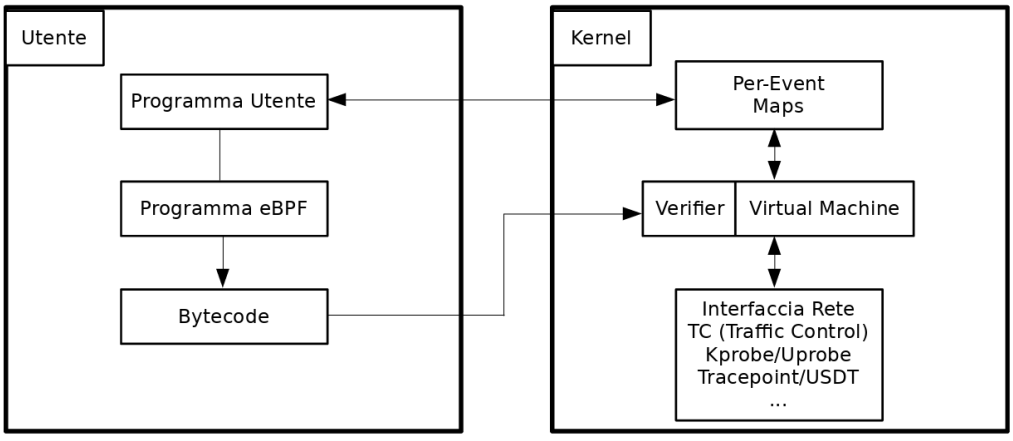

Figura 3.2. Schema funzionamento eBPF.

Come già visto per BPF, un programma utente può interfacciarsi con eBPF definendo in qualche modo un programma eBPF da eseguire nel kernel e, là dove necessario, inizializzare elementi di supporto (per esempio un socket raw nel caso di monitoraggio del traffico di rete tramite socket). Segue una fase di compilazione del programma eBPF che ovviamente si blocca nel caso in cui si riscontrino errori nel codice come istruzioni sbagliate o illecite, chiamate a funzioni non riconosciute, ... e in generale tutti quei problemi che possono essere scoperti in questa fase. Se la compilazione va a buon fine, il programma è trasformato in bytecode, codice non ancora eseguibile,

ma interpretabile dalla macchina virtuale che risiede nel kernel. Il bytecode generato viene passato al livello kernel insieme alla definizione del tipo di programma di cui si tratta. Prima di essere iniettato nella macchina virtuale però, viene passato al verifier, il cui compito, come accennato in precedenza, è quello di analizzare in modo statico il programma eBPF alla ricerca di possibili errori che potrebbero non essere risultati in fase di compilazione. Gli errori che il verifier potrebbe trovare possono essere universali (riguardanti tutti i programmi eBPF) o specifici per un dato tipo di programma. Nel primo caso, il verifier potrebbe per esempio riscontrare un ciclo infinito, cosa non permessa nei programmi eBPF: una funzione di analisi deve terminare non appena ha eseguito il suo scopo, non ha senso che rimanga attiva, anche perché in alcuni casi bloccherebbe l'esecuzione dell'evento stesso (e siccome il programma è eseguito nel kernel potrebbe arrivare a bloccare anche l'intero sistema); oppure potrebbe scoprire che il programma chiamerà una funzione esistente nel kernel, ma non permessa ai programmi eBPF: per motivi di sicurezza, si è deciso fin da subito di limitare al codice eBPF il solo utilizzo delle funzioni davvero necessarie al suo funzionamento (oltre a quelle che sono state create appositamente per esso). Nel secondo caso invece il verifier potrebbe notare che un programma di un certo tipo cercherà di leggere o di scrivere in locazioni di memoria non permesse o che cercherà di chiamare una funzione esistente e permessa a eBPF, ma non utilizzabile da quel tipo di programma (diversi tipi di programma hanno diversi permessi e funzioni a cui appoggiarsi). Se il programma eBPF supera anche il controllo del verifier, viene finalmente iniettato nella macchina virtuale. Non appena l'evento a cui è stato associato il programma si attiva, la macchina virtuale interpreta il codice passandogli come parametri in ingresso dati diversi per diversi tipi di programma. Questo svolge il compito per cui è stato creato (generalmente analizza i parametri in ingresso e reagisce in qualche modo) e poi ritorna un valore intero: per alcuni tipi di programma questo è solo un flag che sta a indicare il successo o il fallimento dell'esecuzione; per altri è l'indicazione di quale operazione il kernel deve eseguire sull'evento corrente (per esempio inoltrare o scartare il pacchetto di rete analizzato). Per quanto riguarda gli eventi analizzabili, come anticipato, sono tutte le funzioni, e più in generale le funzionalità, di un sistema: l'arrivo di un pacchetto di rete su di un'interfaccia (in ingresso o in uscita); l'ingresso in o l'uscita da una funzione di livello utente o di livello kernel; un evento di livello hardware; ... Infine, un programma eBPF può comunicare se necessario con un programma utente e con altri programmi eBPF; un programma utente invece può comunicare con un programma eBPF solo in modo molto limitato. Questo può essere fatto tramite soluzioni per-evento o appoggiandosi alle mappe. Nel primo caso, un programma eBPF può solamente inviare informazioni al livello utente (e non viceversa) e il programma utente le riceverà non appena sono pronte; nel secondo caso invece è possibile lo scambio di informazioni in entrambe le direzioni (e anche tra diversi programmi eBPF) in modo asincrono.

3.3 Scrittura dei programmi utente/eBPF

Come anticipato in precedenza, quando si vuole usufruire di eBPF, così come per il suo predecessore, bisogna scrivere un programma di livello utente e un programma in linguaggio eBPF da passare al livello kernel. Il secondo può essere scritto in diversi modi, ma solamente appoggiandosi alle librerie Libpcap e Libseccomp si è in grado di generare programmi eBPF a partire da una semplice stringa, nel primo caso, o da regole di filtraggio, nel secondo. Nel caso in cui si fosse interessati a usare funzionalità di eBPF non ricoperte da queste librerie, bisogna scrivere questi programmi a mano, cosa che generalmente non crea problemi visto che per lo più non si tratta di programmi troppo complessi. Comunque, il modo in cui viene scritto un programma eBPF dipende anche dal linguaggio utilizzato nella stesura del programma utente. Quest'ultimo poteva inizialmente essere scritto solamente in C, ma con il passare del tempo è nata una seconda opzione, più a alto livello e quindi anche più semplice: BCC.

Siccome durante lo svolgimento di questa tesi si ha utilizzato per lo più la seconda soluzione, la prima verrà affrontata solamente in modo più generico mentre la seconda verrà approfondita maggiormente.

```
#include <linux/bpf.h>
int bpf (int cmd, union bpf_attr *attr, unsigned int size);
```

Figura 3.3. Prototipo system call 'bpf'.

```
/* linux/bpf.h */
enum bpf_cmd {
    BPF_MAP_CREATE,
    BPF_MAP_LOOKUP_ELEM,
    BPF_MAP_UPDATE_ELEM,
    BPF_MAP_DELETE_ELEM,
    BPF_PROG_LOAD,
    BPF_PROG_ATTACH,
    BPF_PROG_DETACH,
    ...
};
```

Figura 3.4. Definizione del primo parametro della system call 'bpf'.

3.3.1 Programma utente in C

Il linguaggio C è il modo più diretto per interagire con eBPF. I modi più semplici e veloci per usufruire di questa tecnologia in questo linguaggio sono tramite le librerie Libpcap e Libseccomp, a seconda dello scopo del programma. Se si vuole semplicemente monitorare il traffico di rete tramite un socket, filtrando quali pacchetti inviare a livello utente e quali no, si può usare la prima libreria; se si vuole invece definire quali system call un dato eseguibile può chiamare, e con quali parametri, filtrando quelle non volute, si può usare la seconda. Ovviamente queste possibilità vanno bene solamente per questi due scopi e solamente se si è interessati a svolgere esattamente il compito per cui le due librerie sono state create. Se si volesse invece personalizzare ulteriormente il programma eBPF o se si volessero utilizzare programmi eBPF di altro tipo, per altri scopi, bisogna per forza scriverli e gestirli direttamente: questo è possibile tramite la system call 'bpf' (senza la 'e') il cui prototipo è mostrato in figura 3.3 e che permette di eseguire tutte le operazioni necessarie per la definizione, il caricamento e la gestione dei programmi eBPF e dei metodi di comunicazione tra i due livelli ricevendo tre parametri in ingresso. In figura 3.4 è possibile trovare parte della definizione del primo parametro, un valore intero che rappresenta l'identificativo dell'operazione che si vuole eseguire. In figura 3.5 si trova invece parte del secondo: un puntatore a una struttura (diversa per diversi comandi) contenente le informazioni necessarie per l'esecuzione dell'azione richiesta. Il terzo parametro invece altro non è che la dimensione in byte del secondo parametro. Per quanto riguarda il valore di ritorno, a parte in casi eccezionali, vale 0 per successo e -1 per fallimento.

Nel caso in cui, per esempio, si volesse creare una mappa per lo scambio di informazioni tra livello utente e programma eBPF in modo asincrono, il comando da passare alla system call sarebbe BPF_MAP_CREATE e la struttura da passare come secondo parametro sarebbe la prima presente nella figura 3.5 che contiene, tra le altre informazioni, il tipo di mappa, la dimensione della chiave, la dimensione del valore e la dimensione massima della mappa. In questo caso il valore di ritorno sarebbe il file descriptor della mappa creata (o -1 in caso di errore). Se invece si volesse caricare un nuovo programma eBPF nel kernel, il comando sarebbe BPF_PROG_LOAD e la struttura sarebbe la seconda presente nella figura 3.5 che contiene, tra le altre informazioni, il tipo di programma, il numero di istruzioni nel programma e le istruzioni stesse; il valore di ritorno in questo caso sarebbe il file descriptor associato al programma caricato.

```
/* linux/bpf.h */
union bpf_attr {
    struct { /* anonymous struct used by BPF_MAP_CREATE command */
        __u32 map_type; /* one of enum bpf_map_type */
        __u32 key_size; /* size of key in bytes */
        __u32 value_size; /* size of value in bytes */
        __u32 max_entries; /* max number of entries in a map */
        __u32 map_flags; /* BPF_MAP_CREATE related
                          * flags defined above.
                          */
        __u32 inner_map_fd; /* fd pointing to the inner map */
        __u32 numa_node; /* numa node (effective only if
                          * BPF_F_NUMA_NODE is set).
                          */
        char map_name[BPF_OBJ_NAME_LEN];
        __u32 map_ifindex; /* ifindex of netdev to create on */
    };
    struct { /* anonymous struct used by BPF_PROG_LOAD command */
        __u32 prog_type; /* one of enum bpf_prog_type */
        __u32 insn_cnt;
        __aligned_u64 insns;
        __aligned_u64 license;
        __u32 log_level; /* verbosity level of verifier */
        __u32 log_size; /* size of user buffer */
        __aligned_u64 log_buf; /* user supplied buffer */
        __u32 kern_version; /* checked when prog_type=kprobe */
        __u32 prog_flags;
        char prog_name[BPF_OBJ_NAME_LEN];
        __u32 prog_ifindex; /* ifindex of netdev to prep for */
    };
    ...
} __attribute__((aligned(8)));
```

Figura 3.5. Definizione del secondo parametro della system call 'bpf'.

Per quanto riguarda invece il programma eBPF, esiste più di un modo per scriverlo. Il modo più diretto è quello di riempire un vettore di strutture del tipo rappresentato in figura 3.6 così da definire l'intero programma istruzione per istruzione. A questo punto il puntatore a questo vettore può essere inserito nella struttura associata al comando BPF_PROG_LOAD (la seconda rappresentata nella figura 3.5), nel campo 'insns' e il programma può essere caricato nel kernel tramite la system call 'bpf'. Siccome però generare ogni istruzione a partire dai rispettivi valori in byte (opcode) può non essere la soluzione migliore, in uno dei file header associati a eBPF sono state definite delle costanti che permettono di riempire la struttura tramite parole chiavi. In pratica ogni possibile istruzione Assembler (del linguaggio eBPF) è stata mappata in modo tale da essere facilmente comprensibile al programmatore e in grado di definire automaticamente una struttura (e quindi un'intera istruzione) a partire dalle parole chiave usate. Un esempio è quello in figura 3.7: lo scopo del programma creato è quello di essere attaccato a un socket su di un'interfaccia di rete e di contare il numero di pacchetti che transitano su di essa divisi per protocollo ip (tcp, udp, ...); per ogni pacchetto viene quindi innanzitutto letto il protocollo e poi viene aperta una mappa appositamente creata nel programma di livello utente per sommare 1 al conteggio dei pacchetti per quel dato protocollo (le mappe sono strutture dati in forma chiave-valore di cui si parlerà meglio nelle prossime sezioni; qui viene usato il protocollo come chiave e il conteggio come valore).

Un'alternativa a questa soluzione, è quella di scrivere il programma eBPF direttamente in

```
/* linux/bpf.h */
struct bpf_insn {
    __u8 code; /* opcode */
    __u8 dst_reg:4; /* dest register */
    __u8 src_reg:4; /* source register */
    __s16 off; /* signed offset */
    __s32 imm; /* signed immediate constant */
};
```

Figura 3.6. Definizione della struttura per la rappresentazione di un'istruzione del linguaggio eBPF.

```
/* linux/bpf.h */
struct bpf_insn prog[] = {
    /* r6 = r1 */
    BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),
    /* r0 = ip->proto */
    BPF_LD_ABS(BPF_B, ETH_HLEN + offsetof(struct iphdr, protocol)),
    /* *(u32 *)(fp - 4) = r0 */
    BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4),
    /* r2 = fp */
    BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
    /* r2 = r2 - 4 */
    BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4),
    /* r1 = map_fd */
    BPF_LD_MAP_FD(BPF_REG_1, map_fd),
    /* r0 = map_lookup(r1, r2) */
    BPF_CALL_FUNC(BPF_FUNC_map_lookup_elem),
    /* if (r0 == 0) goto pc+2 */
    BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
    /* r1 = 1 */
    BPF_MOV64_IMM(BPF_REG_1, 1),
    /* lock *(u64 *) r0 += r1 */
    BPF_XADD(BPF_DW, BPF_REG_0, BPF_REG_1, 0, 0),
    /* r0 = 0 */
    BPF_MOV64_IMM(BPF_REG_0, 0),
    /* return r0 */
    BPF_EXIT_INSN(),
};
```

Figura 3.7. Programma eBPF di esempio in pseudo-Assembler (fonte: 'man bpf').

Assembler o in Restricted-C e poi compilarlo con l'ausilio di clang e llvm, settando come formato 'bpf' (di nuovo, senza la 'e'). L'Assembler è ovviamente il meno utilizzato in quanto più lungo e complesso (forse persino più dello 'pseudo-Assembler' descritto sopra); il Restricted-C invece è il linguaggio C, ma in versione ridotta: ci sono ovviamente restrizioni sulle funzioni che è possibile chiamare, come descritto in precedenza, e anche su alcune istruzioni (per esempio all'inizio non era possibile utilizzare cicli mentre ora sì, ma con ancora qualche limitazione). In ogni caso, una volta scritto il programma in uno di questi due modi e averlo compilato con l'apposita tecnologia, il problema maggiore è caricare il programma nel kernel: non esiste un modo diretto per farlo, ma è possibile prendere spunto dagli esempi riguardanti eBPF contenuti nel sorgente del kernel [4] (e più precisamente dal file bpf_load.c).

3.3.2 Programma utente con BCC

BCC [5] (BPF Compiler Collection) è un pacchetto di piccoli programmi che utilizzano eBPF per diversi tipi di analisi di un sistema: dal tracciamento di funzioni al calcolo di statistiche su determinati eventi (software o hardware) fino al monitoraggio delle prestazioni. Execsnoop, per esempio, è in grado di tracciare in tempo reale le chiamate alla system call execve e di stampare a video informazioni sui parametri ricevuti dalla funzione come il nome del processo da eseguire, la riga di comando e il PID del processo che ha eseguito la richiesta; con opensnoop invece è possibile scoprire in tempo reale quali file vengono aperti e da chi, con la possibilità di filtrare il monitoraggio sul PID del processo che ha chiamato la funzione; mentre gethostlatency permette di calcolare quanto tempo è passato da quando si ha fatto una richiesta a un DNS server a quando si ha ricevuto una risposta. Questi sono solo alcuni dei programmi presenti nel pacchetto, e come la maggior parte degli altri, sono scritti in Python sfruttando la libreria contenuta in BCC stesso. Infatti, i programmi presenti in BCC, oltre a essere utilità di analisi del sistema, sono anche esempi su come utilizzare la libreria, la quale trasporta le funzioni della system call 'bpf' a un livello più alto permettendo quindi a alcuni linguaggi di alto livello di interfacciarsi con eBPF con elevata facilità.

Per il momento i linguaggi utilizzabili sono Python (il più mantenuto, aggiornato, documentato e usato), Lua e C++. In figura 3.8 è rappresentato lo scheletro di un programma Python (l'unico tra i tre utilizzato durante lo svolgimento di questa tesi). Quando un programma di livello utente vuole usufruire di eBPF, deve innanzitutto importare la libreria BCC. A questo punto deve creare un oggetto BPF passandogli in ingresso un programma eBPF, il quale può essere definito all'interno di una stringa nel programma utente stesso (come nell'esempio in figura) o trovarsi in un file separato. Come si può notare, il programma eBPF viene scritto in Restricted-C e contiene diversi header, a seconda del tipo di programma eBPF, e una o più funzioni che definiscono le operazioni da effettuare nel momento in cui un certo evento si attiva. Nel caso per esempio del monitoraggio/gestione del traffico di rete, il programma dovrebbe controllare i byte del pacchetto che riceve in ingresso e ritornare un valore diverso a seconda del tipo di programma; nel caso di monitoraggio di funzioni, dovrebbe analizzare i parametri in ingresso a o in uscita dalla funzione; ... eccetera. Durante la creazione di questo oggetto, BCC compila automaticamente il programma eBPF e lo carica nel kernel. Se esso non viene fermato dal verifier, viene iniettato nella macchina virtuale e da questo momento in avanti lo stesso oggetto verrà usato per tutte le successive operazioni: prima tra tutte l'attaccare una delle funzioni definite nel programma eBPF all'evento che si è interessati a monitorare/gestire (chiamando funzioni diverse a seconda del tipo di programma). Seguono ulteriori possibili operazioni di inizializzazione; infine viene creato un ciclo infinito nel quale potrebbero essere gestite operazioni come la stampa delle informazioni ricevute dal programma eBPF; nel momento in cui si esce dal ciclo di livello utente, il programma eBPF viene scaricato dalla macchina virtuale e staccato dall'evento in modo automatico (maggiori informazioni verranno date nelle prossime sezioni).

Un'ultima considerazione da fare è che il Restricted-C usato in BCC ha qualche funzionalità in più rispetto a quello usato senza la libreria, per semplificare ulteriormente la stesura dei programmi eBPF [6]. Per esempio, quando si vuole definire una mappa per lo scambio di informazioni tra i due livelli, nella programmazione senza la libreria, bisogna farlo a livello utente chiamando la system call 'bpf' con i giusti parametri; con BCC invece una mappa viene definita direttamente dentro al programma eBPF (sarà poi la libreria a gestire il tutto). Un altro esempio potrebbe essere la creazione di istogrammi: siccome molti programmi del pacchetto BCC analizzano prestazioni o calcolano statistiche, sono state create delle funzioni di livello utente per la generazione automatica di istogrammi e delle funzioni di livello kernel per il calcolo dei dati necessari a essi.

3.4 Condivisione dati tra livelli kernel e utente

In alcuni casi i programmi utente e eBPF non necessitano di scambiarsi dati, per esempio, durante la gestione del traffico di rete, l'analisi dei pacchetti e le diverse reazioni possibili vengono tutte eseguite dentro al kernel. In questi casi il programma utente rimarrà bloccato nel ciclo infinito descritto nella sezione precedente (finché non viene terminato da un qualche segnale) senza eseguire

```
#!/usr/bin/python

from bcc import BPF
...

# Programma eBPF #
bpf_program = """\

    #include <...>
    ...

    int ebpf_function (<parametri_in_input>) {
        ...
        return N;
    }

"""

# Inizializza eBPF #
bpf = BPF (text = bpf_program)

...

# Loop Principale #
print ("\n[*] Premere ctrl+c per Terminare...\n")
while 1:
    ...
```

Figura 3.8. Scheletro di un programma Python scritto tramite la libreria BCC.

nessuna operazione. Ma in altri casi è fondamentale che i due siano in grado di contattarsi: per esempio nel caso di monitoraggio di un evento qualunque, è necessario che il programma eBPF avverta il livello utente che l'evento si è attivato (con magari anche qualche informazione riguardante il come è stato attivato); o nella gestione della rete si potrebbe avere un meccanismo di aggiornamento delle regole gestite dal programma eBPF in modo tale che il programma utente sia in grado di dire al livello kernel che la gestione deve essere cambiata senza dover ogni volta scaricare il programma eBPF, aggiornarlo e mandarlo di nuovo in esecuzione; ... In questi altri casi, come accennato nello schema precedente, è possibile utilizzare due tipi di comunicazione: per-evento o tramite l'uso di mappe.

La prima è una forma di comunicazione unidirezionale, utilizzabile unicamente da un programma eBPF per inviare informazioni a un programma utente. E' inoltre sincrona: il livello utente rimane generalmente bloccato in attesa di ricevere le informazioni a cui è interessato. Generalmente, perché in realtà si potrebbero usare meccanismi non bloccanti per la ricezione dei dati, anche se viene fatto solo in casi particolari (perché a questo punto conviene usare le mappe, che permettono lo scambio di dati in modo asincrono).

Il primo meccanismo per-evento è il socket, che viene usato unicamente quando si utilizza eBPF per il monitoraggio della rete tramite socket appunto. In figura 3.9 si può notare lo scheletro di un programma che utilizza questo metodo di comunicazione: il programma eBPF riceve in input una struttura contenente tra le altre cose i byte del pacchetto da analizzare, effettua i controlli che deve e ritorna un valore diverso a seconda se il pacchetto deve essere passato a livello utente o meno (maggiori informazioni verranno date nella prossima sezione); il programma di livello utente, dopo aver inizializzato l'oggetto BPF, caricato il programma eBPF nel kernel e averlo attaccato a un socket RAW aperto su di una certa interfaccia, crea un socket di livello utente, lo associa a

```
#!/usr/bin/python

from bcc import BPF
import socket, os
...

# Programma eBPF #
bpf_program = """\

    #include <bcc/proto.h>
    ...

    int filter (struct __sk_buff *skb) {
        ...
        return N;
    }

"""

# Inizializza eBPF #
bpf = BPF (text = bpf_program)

# Attacca il Programma eBPF a un Socket RAW #
filter_func = bpf.load_func ("filter", BPF.SOCKET_FILTER)
iface="<interface_name>"
BPF.attach_raw_socket (filter_func, iface)

# Inizializza il Socket di Livello Utente #
sock_fd = filter_func.sock
user_socket = socket.fromfd (sock_fd, socket.PF_PACKET, socket.SOCK_RAW,
    socket.IPPROTO_IP)
user_socket.setblocking (True)

# Loop Principale #
print ("\n[*] Premere ctrl+c per Terminare...\n")
while 1:
    try:
        packet = os.read (sock_fd, 2048)
        ...
    except KeyboardInterrupt:
        break
```

Figura 3.9. Scheletro di un programma BCC per la comunicazione tramite socket.

quello di livello kernel e poi si mette in attesa su di esso. Ogni volta che un pacchetto supera il filtro rappresentato dal programma eBPF, esso viene inviato a livello utente, il quale è libero di farne quello che vuole (per esempio stampare a video informazioni su di esso).

Il secondo metodo di comunicazione per-evento è la 'trace pipe': il kernel linux mette a disposizione un file (`/sys/kernel/debug/tracing/trace_pipe`) che viene usato come file d'appoggio per i moduli del kernel che hanno bisogno di inviare informazioni di debug a livello utente. Un programma eBPF di qualunque tipo quindi, può usare questo stesso file per inviare informazioni personalizzate riguardanti l'evento corrente. Questo meccanismo ha però due limitazioni: prima di tutto la pipe in questione è globale quindi non è detto che un programma eBPF sia l'unico a

```
#!/usr/bin/python

from bcc import BPF
...

# Programma eBPF #
bpf_program = """\

    #include <...>
    ...

    int ebpf_function (<parametri_in_input>) {
        bpf_trace_printk ("Hello, World!\\n");
        return N;
    }

"""

# Inizializza eBPF #
bpf = BPF (text = bpf_program)

...

# Loop Principale #
print ("\n[*] Premere ctrl+c per Terminare...\n")
print ("PID MESSAGE")
bpf.trace_print (fmt="{1} {5}")
```

Figura 3.10. Scheletro di un programma BCC per la comunicazione tramite trace-pipe.

utilizzarla in un dato momento, cosa che ovviamente comporta il dover filtrare a livello utente i dati letti per essere certi di visualizzare solamente quelli a cui si è interessati; la seconda limitazione riguarda il fatto che su questa pipe è possibile stampare un massimo di tre argomenti per volta, di cui una sola stringa, quindi se un programma eBPF dovesse inviare a livello utente un elevato numero di parametri, lo dovrebbe fare in due o più momenti. Per questi motivi, questo meccanismo di comunicazione viene per lo più usato in fase di debug. Comunque in figura 3.10 è rappresentato lo scheletro di un programma che utilizza la trace-pipe per inviare un messaggio di 'Hello World!' all'utente (tramite la funzione 'bpf_trace_printk') ogni qual volta l'evento a cui è stato attaccato il programma si attiva. Da notare come il ciclo finale sia stato sostituito da una singola funzione della libreria BCC, la quale permette di ricevere messaggi stampati sulla trace-pipe e di stamparli a video in un formato predefinito finché non riceve un segnale di uscita.

L'ultimo meccanismo per-evento è il ring-buffer, che di nuovo può essere usato con qualunque tipo di programma eBPF e che risulta essere il più utilizzato tra i tre (a meno di monitoraggio di rete tramite socket, in quel caso ovviamente viene utilizzato il socket). Si tratta di un buffer circolare allocato nella memoria associata al kernel nel quale un programma eBPF può scrivere e dal quale un programma utente può leggere. Essendo circolare la memoria riservata per esso non finirà mai, ma se il programma utente non leggesse i dati non appena sono pronti, rischia di perderne alcuni (cosa che in genere non succede). I dati da spedire a livello utente solitamente vengono aggregati in una struttura così da poterli inviare e ricevere tutti in una volta sola. In figura 3.11 è possibile notare lo scheletro di un programma che utilizza questo meccanismo di comunicazione. Nella definizione del programma eBPF si può notare la creazione di una struttura che conterrà i dati da inviare a livello utente, seguita dall'inizializzazione del ring-buffer stesso (di nome 'events'); nella funzione vera e propria invece viene inizializzata la struttura, viene salvato il

PID del processo che ha attivato l'evento che si sta monitorando e infine la struttura viene salvata sul ring-buffer. A livello utente invece, si inizializza innanzitutto l'oggetto BPF e poi viene definita la stessa struttura che è stata definita nel programma eBPF (tramite la libreria ctypes). Viene inoltre definita una funzione di callback e associata al ring-buffer precedentemente creato (tramite il nome assegnatogli). Infine si entra in un ciclo infinito nel quale si incontra una funzione bloccante sulla lettura dei dati dal ring-buffer. Ogni volta che l'evento da analizzare si attiverà, il programma eBPF viene eseguito, la struttura viene riempita e inserita nel ring-buffer; il programma utente sente che ci sono dei nuovi dati su di esso e quindi chiama la funzione di callback, che in questo caso legge semplicemente i dati dal ring-buffer e li stampa a video.

Per quanto riguarda le mappe, come detto in precedenza permettono lo scambio di dati in modo asincrono e in tutte le direzioni: da programma eBPF a programma utente; da programma utente a programma eBPF; tra diversi programmi eBPF. Si tratta di strutture dati in forma chiave-valore rappresentate da un nome grazie al quale un qualunque programma di livello utente o kernel può aggiungere un nuovo elemento (coppia chiave-valore), eliminare un elemento presente, modificare o semplicemente leggere il valore associato a una data chiave o iterare su tutte le chiavi della mappa. In fase di inizializzazione, una mappa necessita di un nome, del formato della chiave (quando non è predefinito dal tipo di mappa scelta), del tipo di valore associato a ogni chiave e del numero massimo di elementi che può contenere. Esistono diversi tipi di mappe, ottimizzate a seconda dell'uso; alcune di esse sono presenti in figura 3.12. La prima per esempio serve per creare una mappa associativa in cui si può scegliere sia il tipo di chiave che il tipo di valore dando massima personalizzazione ai dati da scambiare; la seconda permette di creare un vettore la cui chiave è un intero, cosa che la rende ottimale per ricerche sequenziali; la terza viene utilizzata per leggere i valori presenti nei contatori hardware di un sistema; le ultime due invece sono come i tipi hash e array, ma nel momento in cui vengono create, ne vengono allocate un numero pari al numero di cpu presenti nel sistema e ne viene associata una diversa per ogni cpu, in modo tale che ogni mappa contenga unicamente i dati associati alla rispettiva cpu.

Un'ultima considerazione da fare riguardo alle mappe, è la loro utilità nello scambiare dati tra diversi programmi eBPF. Infatti capita spesso che un programma eBPF dipenda da un altro programma eBPF; in questo caso il primo può salvare le informazioni necessarie al secondo in una mappa così che il secondo le possa leggere, elaborare e, se necessario eliminare. Un esempio potrebbe essere il programma gethostlatency (accennato in precedenza): il suo scopo è quello di analizzare il tempo che passa da quando viene fatta una richiesta al DNS (tramite le funzioni getaddrinfo o gethostbyname) a quando viene ricevuta una risposta. Per farlo deve poter calcolare il tempo che queste funzioni impiegano a essere eseguite. Questo implica che il programma dovrebbe potersi attaccare sia in ingresso che in uscita a ognuna delle funzioni, ma ogni programma eBPF può essere associato a un solo evento. Quello che viene fatto è rappresentato, in parte, in figura 3.13: nel programma eBPF vengono create due funzioni, una da attaccare in ingresso alle due funzioni a cui si è interessati e l'altra in uscita; vengono anche definite due strutture una da usare come valore in una mappa di tipo hash (inizializzata subito dopo alla definizione delle due strutture) e l'altra da usare come contenitore per i dati da inserire su di un ring-buffer; la prima funzione leggerà, tra le altre cose, il PID del processo che ha chiamato una delle due funzioni da monitorare e calcolerà il timestamp (sull'ingresso alle funzioni) e salverà entrambi nella mappa, il PID come chiave e il timestamp all'interno della struttura come valore; la seconda funzione calcolerà il timestamp (sull'uscita dalle funzioni) e controllerà se il PID del processo corrente esiste nella mappa; in caso positivo, viene letta la struttura associata a esso, viene calcolata la differenza di tempo tra i due timestamp ottenuti e infine vengono salvati tutti i dati interessanti all'interno della seconda struttura per poi inviarli a livello utente tramite il ring-buffer.

3.5 Funzionalità

Come già detto in precedenza, eBPF è in grado di tracciare qualunque evento di livello utente e di livello kernel, sia software che hardware, che è possibile trovare in un sistema Linux. Questa affermazione è descritta più nel dettaglio nella figura 3.14, dove si può notare una rappresentazione di un sistema operativo: il livello kernel si trova in basso, il livello utente in alto, le system call al centro e di fianco si ha il collegamento con la CPU. La figura mostra inoltre quali eventi sono

```
#!/usr/bin/python

from bcc import BPF
import ctypes as ct
...

# Programma eBPF #
bpf_program = """\

    #include <...>
    ...

    struct data_t {
        u32 pid;
        ...
    };
    BPF_PERF_OUTPUT (events);

    int ebpf_function (struct pt_regs *ctx) {
        struct data_t data = {};
        data.pid = bpf_get_current_pid_tgid ();
        ...
        events.perf_submit (ctx, &data, sizeof (data));
        return N;
    }

"""

# Inizializza eBPF #
bpf = BPF (text = bpf_program)
...

# Definisci la Struttura da Leggere dal Ring Buffer #
class Data (ct.Structure):
    _fields_ = [("pid", ct.c_uint),
                ...]

# MANAGE_INFO_EVENT #
def manage_info_event (cpu, data, size):
    event = ct.cast (data, ct.POINTER (Data)).contents
    print("%-6d ..." % (event.pid, ...))
bpf ["events"].open_perf_buffer (manage_info_event)

# Loop Principale #
print ("\n[*] Premere ctrl+c per Terminare...\n")
while 1:
    bpf.perf_buffer_poll ()
```

Figura 3.11. Scheletro di un programma BCC per la comunicazione tramite ring-buffer.

tracciabili da quale tipo di programma eBPF (in realtà esistono più tipi, qui ne sono rappresentati solo alcuni). Per esempio si vede come grazie ai tracepoint eBPF sia in grado di tracciare qualunque evento software di livello kernel e di livello utente: quelli riguardanti i device, il file system, lo scheduler, la memoria, lo stack di rete, le system call, le librerie e le applicazioni. Si nota anche

```
/* linux/bpf.h */
enum bpf_map_type {
    BPF_MAP_TYPE_HASH,
    BPF_MAP_TYPE_ARRAY,
    BPF_MAP_TYPE_PERF_EVENT_ARRAY,
    BPF_MAP_TYPE_PERCPU_HASH,
    BPF_MAP_TYPE_PERCPU_ARRAY,
    ...
};
```

Figura 3.12. Definizione dei tipi di mappa utilizzabili in eBPF.

che con i probe si possono analizzare le stesse cose, ma divise tra kprobe e uprobe, rispettivamente probe di livello kernel e di livello utente. Infine si vede che gli eventi hardware vanno invece tracciati con un tipo apposito, mentre non sono rappresentati per esempio i tipi di programma eBPF riguardanti direttamente il monitoraggio o la gestione della rete.

Di seguito vengono analizzati i tipi di programma eBPF più comuni (per il momento) in ordine di arrivo nel kernel; oltre a quelli descritti ne esistono altri che sono però più nuovi e per lo più utilizzabili in casi particolari. Tutti gli esempi citati nelle prossime sezioni sono allegati alla tesi e le istruzioni su come eseguirli possono essere trovate nella guida dell'utente (capitolo 8).

3.5.1 Monitoraggio della rete tramite socket

Il primo tipo di programma eBPF, uscito nel kernel 3.19, è BPF_PROG_TYPE_SOCKET_FILTER. Come suggerisce il nome stesso, esso permette di monitorare la rete di un sistema appoggiandosi a un socket (è l'equivalente di BPF). Sull'interfaccia di rete da tenere sotto controllo viene aperto un socket raw; ogni volta che un pacchetto passa per quel punto, il socket lo sente e il kernel attiva il programma eBPF per l'analisi di una sua copia. Il programma eBPF altro non è che il filtro in grado di decidere se il pacchetto deve essere inviato a livello utente (tramite il socket sul quale il programma utente è in ascolto). In ingresso esso riceve un puntatore a una struttura __sk_buff che contiene, tra le altre informazioni, il tipo di pacchetto, il protocollo, l'interfaccia su cui si trova, il pacchetto stesso e la sua lunghezza. Il programma eBPF analizza la struttura (generalmente i byte del pacchetto) e ritorna un valore pari a 0 se la copia del pacchetto deve essere scartata o -1 se deve essere inviata a livello utente. Quando il livello utente riceve una qualche copia di un qualche pacchetto, può farne ciò che desidera (generalmente stampa a video le informazioni che lo riguardano). Un'ultima considerazione da fare, è che la struttura in ingresso al programma eBPF è modificabile dal programma stesso, ma generalmente questo non ha alcuna utilità visto che è associata a una copia del pacchetto e qualunque modifica venisse fatta non altererebbe in alcun modo l'originale.

Di seguito viene descritto un esempio di un programma il cui scopo è monitorare i pacchetti di rete scambiati tra il sistema stesso e un certo indirizzo IP (130.192.181.193: www.polito.it). In figura 3.15 è rappresentato il programma eBPF sotto forma di stringa da passare alla libreria BCC: i byte del pacchetto vengono innanzitutto castati a diverse strutture che rappresentano i diversi header contenuti dal pacchetto; essi vengono successivamente controllati per decidere se il pacchetto dovrà essere inviato a livello utente o meno. Se il pacchetto non è IPv4, non è TCP o comunque non è diretto verso o ricevuto dall'indirizzo IP scelto, allora il programma ritorna 0 e il kernel sa che non deve fare nulla; altrimenti il programma ritorna -1, valore che indica al kernel che il pacchetto dovrà essere inviato a livello utente. In figura 3.16 è invece rappresentato il programma utente: si inizializza l'oggetto BPF, si carica nel kernel il programma eBPF definito in precedenza, si attacca il programma a un socket di livello kernel aperto sull'interfaccia 'eth0', si apre un socket di livello utente connesso a quello di livello kernel e infine si entra nel loop infinito per la ricezione di nuovi pacchetti. Ogni pacchetto che il kernel invia a livello utente viene letto,

```
bpf_text = """
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>
struct val_t {
    ...
    u64 ts;
};
struct data_t {
    u32 pid;
    u64 delta;
    ...
};
BPF_HASH (start, u32, struct val_t);
BPF_PERF_OUTPUT (events);

int do_entry (struct pt_regs *ctx) {
    ...
    struct val_t val = {};
    u32 pid = bpf_get_current_pid_tgid();
    ...
    val.ts = bpf_ktime_get_ns();
    start.update (&pid, &val);
    return 0;
}

int do_return (struct pt_regs *ctx) {
    struct val_t *valp;
    struct data_t data = {};
    u32 pid = bpf_get_current_pid_tgid ();
    u64 tsp = bpf_ktime_get_ns ();
    valp = start.lookup (&pid);
    if (valp == 0)
        return 0; // missed start
    ...
    data.pid = valp->pid;
    data.delta = tsp - valp->ts;
    events.perf_submit (ctx, &data, sizeof(data));
    start.delete (&pid);
    return 0;
}
"""
```

Figura 3.13. Parte principale del programma eBPF gethostlatency.py appartenente al pacchetto BCC (fonte: https://github.com/iovisor/bcc/blob/master/tools/gethostlatency.py).

viene tradotto negli header IP e TCP e poi vengono stampate a video alcune informazioni su di esso (indirizzo IP e porta mittente, indirizzo IP e porta destinatario e i valori dei flag TCP).

3.5.2 Gestione del Traffic Control

Nella versione del kernel 4.1 sono stati aggiunti i tipi BPF_PROG_TYPE_SCHED_CLS (classifier) e BPF_PROG_TYPE_SCHED_ACT (action). Entrambi servono per la gestione del Traffic Control, il quale, come spiegato nel capitolo 2, permette di controllare lo scheduler dei pacchetti di rete

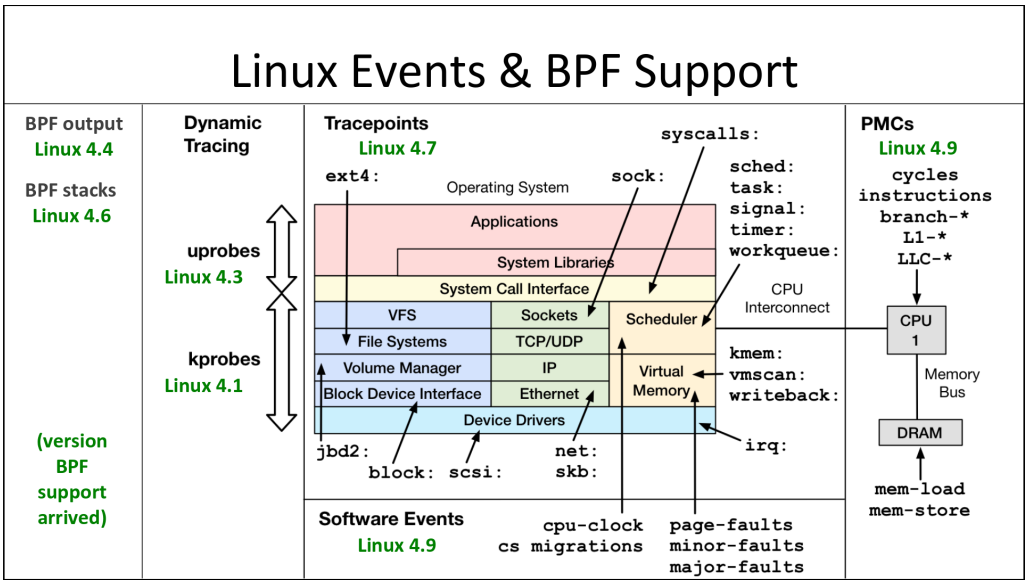

Figura 3.14. Funzionalità tracciabili da eBPF con relative versioni del kernel (fonte: http://www.brendangregg.com/Slides/SCALE2017_perf_analysis_eBPF.pdf).

```
bpf_program = """

    #include <bcc/proto.h>

    int filter (struct __sk_buff *skb) {
        struct ethernet_t *ether_header = 0;
        struct ip_t *ip_header = (struct ip_t *) sizeof (struct ethernet_t);
        struct tcp_t *tcp_header = (struct tcp_t *) (sizeof (struct
            ethernet_t) + sizeof (struct ip_t));

        /* Ignora il Pacchetto se Non è IPv4 */
        if (ether_header->type != 0x0800)
            return 0;

        /* Ignora il Pacchetto se Non è TCP */
        if (ip_header->nextp != 0x06)
            return 0;

        /* Ignora il Pacchetto se Non è Diretto/Ricevuto verso/da l'indirizzo
            IP Scelto (130.192.181.193) */
        if (ip_header->src != 0x82c0b5c1 && ip_header->dst != 0x82c0b5c1)
            return 0;

        /* Invia il Pacchetto a Livello Utente */
        return -1;
    }
"""
```

Figura 3.15. Prima parte del programma `Programmi/esempi_eBPF/socket_program.py` allegato alla tesi.

```
# Inizializza eBPF #
bpf = BPF (text = bpf_program)

# Attacca il Programma eBPF a un Socket di Livello Kernel #
filter_func = bpf.load_func ("filter", BPF.SOCKET_FILTER)
BPF.attach_raw_socket (filter_func, "eth0")

# Inizializza il Socket di Livello Utente #
sock_fd = filter_func.sock
user_socket = socket.fromfd (sock_fd, socket.PF_PACKET, socket.SOCK_RAW,
    socket.IPPROTO_IP)
user_socket.setblocking (True)

# Loop Principale #
ether_header_len = 14
print ("\n[*] Premere ctrl+c per Terminare...\n")
while 1:
    try:
        # Leggi il Pacchetto
        packet = os.read (sock_fd, 2048)

        # Traduci i Byte del Pacchetto negli Header IP-TCP
        ip_header = unpack ('!BBHHHBBH4s4s',
            packet[ether_header_len:ether_header_len+20])
        ip_header_len = (ip_header[0] & 0xF) * 4
        tcp_header = unpack ('!HHLLBBHHH',
            packet[ether_header_len+ip_header_len :
            ether_header_len+ip_header_len+20])

        # Stampa Informazioni sul Pacchetto
        src_ip = socket.inet_ntoa (ip_header[8]);
        dst_ip = socket.inet_ntoa (ip_header[9]);
        print ("[*] %s:%d -> %s:%d (S:%d P:%d A:%d F:%d)" % (str(src_ip),
            tcp_header[0], str(dst_ip), tcp_header[1], \
                    (lambda : 1 if (tcp_header[5] & 2) > 0 else 0)(), \
                    (lambda : 1 if (tcp_header[5] & 8) > 0 else 0)(), \
                    (lambda : 1 if (tcp_header[5] & 16) > 0 else 0)(), \
                    (lambda : 1 if (tcp_header[5] & 1) > 0 else 0)()))

    except KeyboardInterrupt:
        # Termina Monitoraggio #
        break
```

Figura 3.16. Seconda parte del programma `Programmi/esempi_eBPF/socket_program.py` allegato alla tesi.

presente nel kernel Linux [7] dividendo i pacchetti in transito in code a cui è associata una certa azione. Il primo tipo di programma (CLS) può essere installato in ingresso a una coda precedentemente creata e utilizzato per decidere se il pacchetto corrente dovrà essere inoltrato o meno in quella coda; il secondo tipo invece (ACT) può essere installato in uscita a una coda e serve per decidere quale azione eseguire sui pacchetti che vengono accodati in essa. Come si può dedurre, i due tipi sono, per il momento, interscambiabili e usarli assieme risulterebbe ridondante (è inutile analizzare un pacchetto sia in ingresso che in uscita alla stessa coda): si può pensare per esempio di scrivere un programma CLS che selezioni tutti i pacchetti ricevuti da un certo indirizzo IP e

accodarli in una coda a cui è stata associata un'azione di scartamento dei pacchetti; ma la stessa cosa potrebbe essere fatta installando un programma ACT al fondo di una coda generica il cui scopo è analizzare i pacchetti e, nel caso di indirizzo IP sorgente pari a quello indesiderato, di eseguire un'azione di scartamento; utilizzare la prima versione o la seconda è solo una questione di gusti; usarle entrambe sarebbe di certo poco utile e molto meno ottimizzante che usare unicamente una delle due soluzioni.

Anche in questo caso il programma eBPF (CLS o ACT) riceve in ingresso un puntatore a una struttura __sk_buff. Anche qui la struttura può essere modificata, ma in questo caso, siccome contiene i dati del pacchetto originale e non una loro copia, le modifiche potrebbero influenzare il traffico (a seconda della modifica) dando quindi la possibilità di mettere in pratica meccanismi di NAT, reindirizzamento, load balancing, filtraggio, ... direttamente dentro al kernel. I valori di ritorno dei programmi sono invece quelli che suggeriscono al kernel quale operazione deve eseguire: se si ritorna 0 in un CLS, si sta dicendo che il pacchetto non appartiene alla coda corrente, se si ritorna -1 invece sì; se si ritorna 0 in un ACT si indica che il pacchetto deve proseguire, se si ritorna 2 si sta dicendo che il pacchetto deve essere scartato e se si ritorna -1 si indica che bisogna usare l'azione di default associata alla coda (se esiste). Siccome qui si tratta di gestione di traffico, potrebbe non essere necessario avvertire il programma utente di ciò che sta succedendo, ma se lo si volesse fare, bisognerebbe utilizzare uno qualunque dei metodi descritti sopra (a parte il socket), scelto in base alle necessità.

Di seguito viene descritto un esempio di un programma che crea e avvia un firewall che permette al sistema di comunicare solamente con un certo indirizzo IP (130.192.181.193: www.polito.it). In figura 3.17 è rappresentato il programma eBPF sotto forma di stringa da passare alla libreria BCC: i byte del pacchetto vengono innanzitutto castati a diverse strutture rappresentati i diversi header del pacchetto stesso; essi vengono successivamente controllati per decidere se il pacchetto dovrà essere filtrato o meno. Se il pacchetto non è IPv4, non è TCP (con l'eccezione del protocollo DNS) o comunque non è diretto verso o ricevuto dall'indirizzo IP scelto, allora il programma ritorna -1 così che il kernel invii il pacchetto nella coda a cui verrà associato il programma eBPF; nel caso invece di traffico DNS e scambio dati con l'indirizzo IP scelto, il programma ritorna 0, valore che indica al kernel che non deve fare nulla (e quindi lasciare che il pacchetto prosegua la sua strada normalmente); inoltre nel caso di scambio dati con un indirizzo diverso da quello scelto, ma comunque con protocollo TCP, il programma invia al livello utente un messaggio di avvenuto filtraggio contenente gli indirizzi IP sorgente e destinazione del pacchetto. In figura 3.18 è invece rappresentato il programma utente: viene inizializzato Iproute, l'utilità che permette la gestione di Tc, viene inizializzato l'oggetto BPF, viene caricato nel kernel il programma eBPF precedentemente definito (di tipo CLS), viene creata una coda nel Traffic Control installando in ingresso il suddetto programma eBPF e associando in uscita un'azione di 'drop'; infine si attendono messaggi dal kernel e li si stampa non appena arrivano. Quando il programma eBPF dirà al kernel di accodare il pacchetto nella coda a cui lui è associato, il pacchetto verrà automaticamente scartato dall'azione di 'drop'; quando invece il programma dirà al kernel di lasciarlo andare, il suo percorso non subirà alcuna modifica.

3.5.3 Analisi di funzioni tramite probe

Il tipo BPF_PROG_TYPE_KPROBE è arrivato con la versione 4.1 del kernel e permette l'analisi di funzioni tramite probe [8]. Come spiegato nel capitolo 2, un probe può essere attivato su un qualunque indirizzo di livello kernel o utente chiedendo al kernel di settare un breakpoint a quell'indirizzo e registrare una certa funzione di callback (in questo caso il programma eBPF). Ogni volta che qualcuno passa per quell'indirizzo, il breakpoint scatta e il kernel cede il controllo alla relativa funzione. Quando quest'ultima ha finito di svolgere il suo compito, il kernel fa proseguire l'esecuzione da dov'era stata interrotta. I programmi di questo tipo ricevono in ingresso un puntatore alla struttura pt_regs contenente i valori di tutti i registri nel momento in cui il probe si è attivato. Il programma generalmente analizza i registri (tramite i quali è in grado di accedere ai valori in ingresso o in uscita a/da una funzione e allo stack), manda determinate informazioni al livello utente tramite mappe o ring-buffer (o pipe) e poi ritorna 0 in caso di successo o -1 in caso di errore (come ogni modulo del kernel). Importante è sottolineare il fatto che la struttura

```
bpf_program = """

    #include <bcc/proto.h>

    int filter (struct __sk_buff *skb) {
        struct ethernet_t *ether_header = 0;
        struct ip_t *ip_header = (struct ip_t *) sizeof (struct ethernet_t);
        struct udp_t *udp_header = (struct udp_t *) (sizeof (struct
            ethernet_t) + sizeof (struct ip_t));

        /* Filtra il Pacchetto se Non è IPv4 */
        if (ether_header->type != 0x0800)
            return -1;

        /* Lascia Passare il Pacchetto se è DNS */
        if (ip_header->nextp == 0x11 && (udp_header->dport == 0x0035 ||
            udp_header->sport == 0x0035))
            return 0;

        /* Filtra il Pacchetto se Non è TCP */
        if (ip_header->nextp != 0x06)
            return -1;

        /* Lascia Passare il Pacchetto se è Diretto/Arriva verso/da
            l'Indirizzo IP scelto (130.192.181.193) */
        if (ip_header->src == 0x82c0b5c1 || ip_header->dst == 0x82c0b5c1)
            return 0;

        /* Filtra il Pacchetto */
        bpf_trace_printk ("Filtrato il Pacchetto: %lx->%lx\\n",
            ip_header->src, ip_header->dst);
        return -1;
    }
"""
```

Figura 3.17. Prima parte del programma `Programmi/esempi_eBPF/tc_program.py` allegato alla tesi.

in ingresso non è modificabile e che quindi le funzioni analizzate non possono essere influenzate in alcun modo (utilizzando solamente eBPF). Questo non perché non sia possibile modificare i registri da una funzione associata a un probe, ma perché per motivi di sicurezza è stato scelto di limitare i programmi eBPF di questo tipo al solo monitoraggio/tracciamento di funzioni, non al filtraggio/reindirizzamento. Quindi questo tipo di programma viene usato sia per il monitoraggio in tempo reale delle chiamate a funzione che per il calcolo di statistiche sulle chiamate a funzione o sui parametri passati a esse che per controllare le performance di un sistema. Inoltre risulta spesso necessario attaccare due programmi di questo tipo alla stessa funzione (sia in ingresso che in uscita) con passaggio di parametri tra il primo e il secondo (tramite mappe). Un'ultima considerazione da fare è che nonostante il tipo di programma si chiami KPROBE, con esso è possibile installare probe sia di livello kernel che di livello utente; in BCC questa distinzione è più visibile: quando si vuole attaccare un programma eBPF a un probe tramite questa libreria, lo si fa con le funzioni attach_kprobe, attach_kretprobe, attach_uprobe, attach_uretprobe (che sotto la superficie, utilizzano tutte un programma eBPF di tipo KPROBE).

Di seguito vengono descritte le parti principali di un programma appartenente al pacchetto BCC ('execsnoop.py') che serve per monitorare l'esecuzione di nuovi processi tramite l'analisi in

```
# Inizializza IProute #
ipr = IPRoute()
ipr_db = IPDB (nl=ipr)
interface = ipr_db.interfaces.eth0

# Inizializza eBPF #
bpf = BPF (text=bpf_program)
pbr = bpf.load_func ("filter", BPF.SCHED_CLS)

# Inizializza TC #
ipr.tc ("add", "ingress", interface.index, "ffff:")
ipr.tc ("add-filter", "bpf", interface.index, ":1", fd=pbr.fd, name=pbr.name,
    parent="ffff:", action="drop", classid=1)

# Loop Principale #
print ("\n[*] Premere ctrl+c per Terminare...\n")
try:
    bpf.trace_print ()
finally:
    ipr.tc ("del", "ingress", interface.index, "ffff:")
    ipr_db.release ()
```

Figura 3.18. Seconda parte del programma `Programmi/esempi_eBPF/tc_program.py` allegato alla tesi.

tempo reale di chiamate alla system call 'execve'. In figura 3.19 è rappresentato il programma eBPF sotto forma di stringa da passare alla libreria BCC: innanzitutto viene definita una struttura che conterrà i dati da inviare a livello utente tramite il ring-buffer di nome 'events' e poi vengono definite due funzioni, una da attaccare all'ingresso e una all'uscita della system call 'execve'. La prima funzione legge i parametri in ingresso alla 'execve' (il nome del file da eseguire e i parametri da passargli sulla riga di comando), il PID e il comm (nome) del processo che ha chiamato la 'execve', li salva nella struttura e li invia a livello utente grazie alle funzioni 'submit_arg' (definite nello stesso programma eBPF, ma non riportate qui visto che l'unica operazione che eseguono è l'inoltrare i dati che gli si passa verso il ring-buffer come spiegato in precedenza). La seconda funzione invece legge il valore ritornato dalla 'execve', il PID e il comm (nome) del processo che ha chiamato la 'execve' e li invia a livello utente tramite lo stesso ring-buffer. Il livello utente sarà in grado di capire quale delle due funzioni gli ha inviato i dati correnti grazie al type della struttura. In figura 3.20 è invece rappresentato il programma utente: viene inizializzato l'oggetto BPF, le due funzioni descritte prima vengono attaccate in ingresso e in uscita alla system call 'execve', vengono definite due classi, una per contenere i dati che verranno ricevuti sul ring-buffer e l'altra per la traduzione dell'evento che ha generato quei dati, e infine viene associata la funzione 'print_event' al ring-buffer. Ogni volta che dei dati sono pronti sul ring-buffer, la funzione viene chiamata: se i dati sono stati inviati dalla funzione in ingresso alla system call, vengono salvati; se invece sono stati inviati dalla seconda li si stampa tutti.

3.5.4 Analisi di Tracepoint

Dalla versione del kernel 4.7 si ha a disposizione anche il tipo BPF_PROG_TYPE_TRACEPOINT che permette di definire un programma eBPF per l'analisi di tracepoint [9] di livello kernel o utente. Come spiegato nel capitolo 2, essi permettono di analizzare dei punti predefiniti nel codice (utente o kernel) abilitandoli e associandogli una funzione di callback (il programma eBPF in questo caso). Quando qualcuno passerà per quel punto, il controllo interno al codice si renderà conto che il tracepoint è attivo e finirà quindi per chiamare la funzione associata a esso. Nel momento in cui il programma eBPF ritornerà si riprenderà dall'istruzione successiva alla chiamata a funzione (come

```
bpf_text = """
...
struct data_t {
    u32 pid; // PID as in the userspace term (i.e. task->tgid in kernel)
    char comm[TASK_COMM_LEN];
    enum event_type type;
    char argv[ARGSIZE];
    int retval;
};
BPF_PERF_OUTPUT(events);

...

int syscall__execve(struct pt_regs *ctx, const char __user *filename,
           const char __user *const __user *__argv,
           const char __user *const __user *__envp) {
    // create data here and pass to submit_arg to save stack space (#555)
    struct data_t data = {};
    data.pid = bpf_get_current_pid_tgid() >> 32;
    bpf_get_current_comm(&data.comm, sizeof(data.comm));
    data.type = EVENT_ARG;

    __submit_arg(ctx, (void *)filename, &data);

    // skip first arg, as we submitted filename
    for (int i = 1; i < MAXARG; i++) {
        if (submit_arg(ctx, (void *)&__argv[i], &data) == 0)
             goto out;
    }

    ...
out:
    return 0;
}

int do_ret_sys_execve(struct pt_regs *ctx)
{
    struct data_t data = {};
    data.pid = bpf_get_current_pid_tgid() >> 32;
    bpf_get_current_comm(&data.comm, sizeof(data.comm));
    data.type = EVENT_RET;
    data.retval = PT_REGS_RC(ctx);
    events.perf_submit(ctx, &data, sizeof(data));

    return 0;
}
"""
```

Figura 3.19. Prima parte del programma `Programmi/esempi_eBPF/probe_program.py` allegato alla tesi.

per una qualunque altra chiamata a funzione). I programmi di questo tipo ricevono in ingresso un puntatore a struttura contenete i parametri interessanti per il debug di quel punto (definiti nel file `/sys/kernel/debug/tracing/events/<categoria_evento>/<evento>/format`) indirizzabile

```
# initialize BPF
b = BPF(text=bpf_text)
execve_fnname = b.get_syscall_fnname("execve")
b.attach_kprobe(event=execve_fnname, fn_name="syscall__execve")
b.attach_kretprobe(event=execve_fnname, fn_name="do_ret_sys_execve")

...
class Data(ct.Structure):
    _fields_ = [
        ("pid", ct.c_uint),
        ("comm", ct.c_char * TASK_COMM_LEN),
        ("type", ct.c_int),
        ("argv", ct.c_char * ARGSIZE),
        ("retval", ct.c_int),
    ]
class EventType(object):
    EVENT_ARG = 0
    EVENT_RET = 1
...

# process event
def print_event(cpu, data, size):
    event = ct.cast(data, ct.POINTER(Data)).contents

    if event.type == EventType.EVENT_ARG:
        argv[event.pid].append(event.argv)
    elif event.type == EventType.EVENT_RET:
        ...
            argv_text = b' '.join(argv[event.pid]).replace(b'\n', b'\\n')
            printb(b"%-16s %-6d %-6s %3d %s" % (event.comm, event.pid,
                   ppid, event.retval, argv_text))
        try:
            del(argv[event.pid])
        except Exception:
            pass

# loop with callback to print_event
b["events"].open_perf_buffer(print_event)
while 1:
    b.perf_buffer_poll()
```

Figura 3.20. Seconda parte del programma `Programmi/esempi_eBPF/probe_program.py` allegato alla tesi.

con il nome di 'args' con BCC. Per esempio, per tracepoint installati in ingresso a una funzione, tra le altre informazioni ci sono anche i parametri in ingresso alla funzione stessa; per quelli installati in uscita a una funzione invece è presente anche il valore di ritorno; e per quelli installati in punti generici, si trovano almeno i valori dei registri. Un programma eBPF di questo tipo generalmente svolge lo stesso lavoro di un programma di tipo PROBE (analisi dei parametri in ingresso e scambio di informazioni con il livello utente) e poi ritorna 0 in caso di successo o -1 in caso di errore. Come per il tipo precedente, anche qui i parametri in ingresso non sono modificabili e quindi l'esecuzione del programma non può essere influenzata in alcun modo, ma solo analizzata; per questo motivo i tipi PROBE e TRACEPOINT possono essere interscambiabili quando si vuole analizzare una certa funzione (in ingresso e/o in uscita). Infine, anche con questo tipo di programma è possibile

tracciare direttamente eventi sia di livello utente che di livello kernel mentre con BCC bisogna appoggiarsi alla funzione attach_tracepoint per i tracepoint di livello kernel o usare le funzioni interne alla classe USDT (User Statically-Defined Tracing) per quelli di livello utente.

Di seguito vengono descritte le parti principali di un programma appartenente al pacchetto BCC ('tcpaccept.py') che serve per monitorare lo stato di socket tcp tramite l'attivazione del tracepoint 'sock/inet_sock_set_state'. In figura 3.21 è rappresentato il programma eBPF sotto forma di stringa da passare alla libreria BCC: viene definita una funzione per il monitoraggio del tracepoint 'sock/inet_sock_set_state' dentro alla quale viene controllato il protocollo del socket (parametro ricevuto in input; come spiegato sopra è possibile ricevere una lista dei parametri contenuti nella struttura di input leggendo il file '/sys/kernel/debug/tracing/events/sock/inet_sock_set_state/format' come root) e altri valori utili, poi viene riempita una struttura di diverso tipo, definite nello stesso programma eBPF, a seconda del protocollo che il socket supporta (ipv4 o ipv6) con i dati da inviare a livello utente (il PID e il comm del processo che ha attivato il tracepoint, gli indirizzi IP sorgente/destinazione e la porta sorgente). Infine i dati vengono inviati a livello utente tramite uno dei due ring-buffer definiti nello stesso programma. In figura 3.22 è invece rappresentata una parte del programma utente: vengono create le due classi che rappresentano le strutture da ricevere sui due ring-buffer (una IPv4 e l'altra IPv6), viene inizializzato l'oggetto BPF, vengono definite le due funzioni di callback e associate ai rispettivi ring-buffer. Infine si aspettano dati dal kernel in un ciclo infinito. Ogni volta che qualche programma attiva il tracepoint, la rispettiva funzione si attiva e vengono stampati a video il PID e il comm del processo che sta utilizzando il socket in analisi, gli indirizzi IP sorgente e destinazione e la porta sorgente.

3.5.5 Gestione del traffico di rete con XDP

Nella versione del kernel 4.8 è stato aggiunto un nuovo tipo di programma per la gestione del traffico di rete: BPF_PROG_TYPE_XDP (eXpress Data Path). Esso permette di analizzare e modificare in tempo reale i pacchetti in transito su di una certa interfaccia di rete appoggiandosi a un nuovo hook (installato appositamente per questo tipo di programma). Esso si trova nel punto più basso possibile dello stack di rete, un punto ideale per questo tipo di lavoro perché i pacchetti in ingresso non sono ancora stati elaborati dal sistema e quelli in uscita non verranno più modificati. Nel momento in cui si carica un programma di questo tipo nella macchina virtuale, il kernel attiva l'hook; ogni volta che un pacchetto passa da quel punto, esso si attiva e il kernel passa il controllo al programma eBPF. In ingresso quest'ultimo riceve un puntatore a una struttura xdp_md che contiene unicamente un puntatore al primo byte e uno all'ultimo byte del pacchetto che ha attivato l'hook (questo proprio perché si trova così in basso nello stack di rete e non si possono avere altre informazioni). Il programma analizza e, se necessario, modifica il pacchetto e poi ritorna XDP_ABORTED in caso di errore, XDP_DROP per dire al kernel che il pacchetto deve essere scartato o XDP_PASS per dire al kernel di inoltrare il pacchetto. Da notare è il fatto che le uniche operazioni eseguibili sul pacchetto sono inoltra o scarta (a parte in caso di errore), ma nonostante questo non si parla unicamente di filtraggio di pacchetti, ma di gestione del traffico di rete: siccome il pacchetto che si riceve in input è il pacchetto originale e siccome i suoi byte sono anche modificabili, è possibile ottenere qualcosa in più che il filtraggio, semplicemente modificando determinati campi del pacchetto. Per esempio, se si volessero reindirizzare i pacchetti di un certo tipo verso un certo host, basterebbe modificare l'indirizzo IP del destinatario contenuto nel pacchetto stesso e poi dire al kernel di inoltrarlo. A questo punto il pacchetto avrà un destinatario diverso e il reindirizzamento verrà fatto automaticamente dalle regole di routing. La stessa cosa si può fare se si volesse reindirizzare qualcosa verso una certa porta: in questo caso basterebbe modificare il valore della porta di destinazione e poi inoltrare il pacchetto modificato. E se si volesse creare un meccanismo di NAT basterebbe modificare l'indirizzo IP sorgente e così via.

Di seguito viene descritto un esempio di un programma che crea e avvia un firewall che permette al sistema di comunicare solamente con un certo indirizzo IP (130.192.181.193: www.polito.it). In figura 3.23 è rappresentato il programma eBPF sotto forma di stringa da passare alla libreria BCC: i byte del pacchetto vengono innanzitutto castati a diverse strutture rappresentati i diversi header del pacchetto stesso; essi vengono successivamente controllati per decidere se il pacchetto dovrà essere filtrato o meno. Se il pacchetto non è TCP (con l'eccezione del protocollo DNS) o comunque

```
...
bpf_text_tracepoint = """
TRACEPOINT_PROBE(sock, inet_sock_set_state)
{
    if (args->protocol != IPPROTO_TCP)
        return 0;
    u32 pid = bpf_get_current_pid_tgid();
    // pull in details
    u16 family = 0, lport = 0;
    family = args->family;
    lport = args->sport;

    if (family == AF_INET) {
        struct ipv4_data_t data4 = {.pid = pid, .ip = 4};
        data4.ts_us = bpf_ktime_get_ns() / 1000;
        __builtin_memcpy(&data4.saddr, args->saddr, sizeof(data4.saddr));
        __builtin_memcpy(&data4.daddr, args->daddr, sizeof(data4.daddr));
        data4.lport = lport;
        bpf_get_current_comm(&data4.task, sizeof(data4.task));
        ipv4_events.perf_submit(args, &data4, sizeof(data4));
    } else if (family == AF_INET6) {
        struct ipv6_data_t data6 = {.pid = pid, .ip = 6};
        data6.ts_us = bpf_ktime_get_ns() / 1000;
        __builtin_memcpy(&data6.saddr, args->saddr, sizeof(data6.saddr));
        __builtin_memcpy(&data6.daddr, args->daddr, sizeof(data6.daddr));
        data6.lport = lport;
        bpf_get_current_comm(&data6.task, sizeof(data6.task));
        ipv6_events.perf_submit(args, &data6, sizeof(data6));
    }
    // else drop

    return 0;
}
"""
```

Figura 3.21. Prima parte del programma `Programmi/esempi_eBPF/tracepoint_program.py` allegato alla tesi.

non è diretto verso o ricevuto dall'indirizzo IP scelto, allora il programma ritorna XDP_DROP che indica al kernel che il pacchetto dovrà essere eliminato; nel caso invece di traffico non IPv4 (pacchetti ARP per esempio), DNS e dello scambio dati con l'indirizzo IP scelto, il programma ritorna XDP_PASS, valore che invece indica al kernel che deve lasciare che il pacchetto prosegua la sua strada normalmente; inoltre nel caso di scambio dati con un indirizzo diverso da quello scelto, ma comunque con protocollo TCP, il programma invia al livello utente un messaggio di avvenuto filtraggio contenente gli indirizzi IP sorgente e destinazione del pacchetto. In figura 3.24 è invece rappresentato il programma utente: vengono inizializzati l'oggetto BPF e l'utilità XDP, che risiede nel kernel, a cui si associa la funzione definita nel programma eBPF. Infine si attendono messaggi dal kernel e li si stampa non appena arrivano.

3.6 eBPF per la sicurezza

Avendo analizzato ora eBPF, è facile notare come sia possibile mettere in atto meccanismi di sicurezza in tempo reale con il solo ausilio di questa tecnologia. Grazie alle sue funzionalità di

```
...
class Data_ipv4(ct.Structure):
    _fields_ = [
        ("ts_us", ct.c_ulonglong),
        ("pid", ct.c_ulonglong),
        ("saddr", ct.c_uint),
        ("daddr", ct.c_uint),
        ("ip", ct.c_ulonglong),
        ("lport", ct.c_ulonglong),
        ("task", ct.c_char * TASK_COMM_LEN)
    ]
class Data_ipv6(ct.Structure):
    ...

# process event
def print_ipv4_event(cpu, data, size):
    event = ct.cast(data, ct.POINTER(Data_ipv4)).contents
    ...
    print("%-6d %-12.12s %-2d %-16s %-16s %-4d" % (event.pid,
        event.task.decode(), event.ip,
        inet_ntop(AF_INET, pack("I", event.daddr)),
        inet_ntop(AF_INET, pack("I", event.saddr)), event.lport))

def print_ipv6_event(cpu, data, size):
    ...

# initialize BPF
b = BPF(text=bpf_text)

...

# read events
b["ipv4_events"].open_perf_buffer(print_ipv4_event)
b["ipv6_events"].open_perf_buffer(print_ipv6_event)
while 1:
    b.perf_buffer_poll()
```

Figura 3.22. Seconda parte del programma `Programmi/esempi_eBPF/tracepoint_program.py` allegato alla tesi.

monitoraggio, tracciamento, filtraggio e gestione di determinati eventi sarebbe infatti possibile controllare un intero sistema (ricordando però di gestire nel miglior modo possibile l'equilibrio tra ottimizzazione e sicurezza) effettuando diversi tipi di controlli tramite diversi tipi di programmi eBPF. Il tipo SOCKET_FILTER per esempio potrebbe essere utilizzato per analizzare in tempo reale il traffico di rete e stampare a video delle informazioni su tutti i pacchetti inviati e/o ricevuti o su di un loro sottoinsieme, cosa che torna molto utile in qualunque sistema, dal più piccolo al più grande, purché connesso a internet: si potrebbe infatti scoprire a chi il sistema si è connesso e, nel caso di traffico non cifrato, anche quali informazioni sta inviando o ricevendo; ovviamente nel caso di un'elevata quantità di traffico risulterebbe più complicato notare qualcosa di sospetto (a meno che non si stia cercando qualcosa in particolare), ma potendo salvare l'analisi, la si potrebbe analizzare con calma in seguito. Nel caso in cui si volessero invece automatizzare delle azioni su determinati tipi di pacchetti in transito, si potrebbero usare i programmi SCHED_CLS/SCHED_ACT o XDP: grazie a essi si potrebbero filtrare direttamente le richieste/risposte indesiderate per evitare che qualcuno possa accedere a risorse private, meccanismo che serve per lo più in sistemi in cui si

```
bpf_prog = """\
    ...

    int filter (struct xdp_md *ctx) {
        void* data = (void*)(long) ctx->data;
        void* data_end = (void*)(long) ctx->data_end;

        /* Inizializza gli Header */
        struct ethhdr *eth = data;
        uint64_t nh_off = sizeof (*eth);
        if (data + nh_off > data_end)
            return XDP_PASS;
        struct iphdr *iph = data + nh_off;
        if ((void*) &iph[1] > data_end)
            return XDP_PASS;
        struct udphdr *udph = ((void *) iph) + sizeof (*iph);
        if ((void*) &udph[1] > data_end)
            return XDP_PASS;

        /* Lascia Passare il Pacchetto se Non è IPv4 */
        if (eth->h_proto != htons (ETH_P_IP))
            return XDP_PASS;

        /* Lascia Passare il Pacchetto se è DNS */
        if (iph->protocol == 0x11 && (udph->dest == 0x3500 || udph->source ==
            0x3500))
            return XDP_PASS;

        /* Filtra il Pacchetto se Non è TCP */
        if (iph->protocol != 0x06)
            return XDP_DROP;

        /* Lascia Passare il Pacchetto se è Diretto/Arriva verso/da
            l'Indirizzo IP scelto (130.192.181.193) */
        if (iph->saddr == 0xc1b5c082 || iph->daddr == 0xc1b5c082)
            return XDP_PASS;

        /* Filtra il Pacchetto */
        bpf_trace_printk ("Filtrato il Pacchetto: %lx->%lx\\n", iph->saddr,
            iph->daddr);
        return XDP_DROP;
    }
"""
```

Figura 3.23. Prima parte del programma `Programmi/esempi_eBPF/xdp_program.py` allegato alla tesi.

conosce effettivamente il traffico di rete (come un server per esempio): si può filtrare in base agli indirizzi sorgente/destinazione, alle porte sorgente/destinazione, ai protocolli, alle richieste, ... e per quanto riguarda il traffico non cifrato, si potrebbe anche analizzare il contenuto dei pacchetti e avere magari delle regole che filtrino anche in base a questo (a differenza della maggior parte delle altre soluzioni). Insomma, grazie a eBPF sarebbe possibile creare un intero firewall di rete direttamente dentro al kernel.

Per quanto riguarda il tracciamento di funzioni interne a un sistema, si potrebbero usare i tipi

```
# Inizializza eBPF & XDP #
bpf = BPF (text = bpf_prog)
bpf_func = bpf.load_func ("filter", BPF.XDP)
interface = "eth0"
bpf.attach_xdp (interface, bpf_func, 0)

# Loop Principale #
print ("\n[*] Premere ctrl+c per Terminare...\n")
try:
    bpf.trace_print ()
finally:
    bpf.remove_xdp (interface, 0)
print ("\n")
```

Figura 3.24. Seconda parte del programma `Programmi/esempi_eBPF/xdp_program.py` allegato alla tesi.

KPROBE o TRACEPOINT con i quali si potrebbero analizzare in tempo reale almeno alcune funzionalità che si ritengono rischiose. Per esempio si potrebbe scoprire quali processi vengono eseguiti, da chi e con quali parametri; o quali file vengono aperti, da chi e per quale motivo. In entrambi i casi è semplice capire se ciò che sta succedendo è lecito o illecito. Ci sono però situazioni in cui questo non è così facile: si potrebbe analizzare il comportamento di un eseguibile, stampando a video quali chiamate a funzione vengono fatte e con quali parametri, ma potrebbe non essere così semplice sapere se i parametri sono leciti o meno (nel caso di parametri molto dinamici). Ma di nuovo, trattandosi di monitoraggio, non è necessario pianificare azioni da eseguire in modo automatico e il più in fretta possibile, in alcuni casi si potrebbero semplicemente analizzare i risultati successivamente. Inoltre con questi tipi di programma sarebbe anche possibile analizzare le connessioni aperte e tutte le altre funzionalità di networking. Generalmente questo viene fatto con i tipi descritti sopra, ma qui si potrebbero effettuare controlli diversi, il più interessante dei quali è certamente l'analisi di pacchetti cifrati: se la cifratura venisse fatta da funzioni di librerie conosciute, sarebbe possibile attaccarsi a queste funzioni e analizzare i parametri in ingresso per ottenere il payload prima che venga cifrato (per traffico in uscita) o dopo che è stato decifrato (per traffico in ingresso). Tutti questi controlli sulle funzioni interne, tornano ovviamente utili su tutti i tipi di sistema e trattandosi unicamente di monitoraggio, le regole sono più semplici da definire anche su sistemi che non si conoscono a fondo.

Ovviamente tutto ciò riguarda l'uso diretto di eBPF. Se si volessero usare programmi che utilizzano al proprio interno eBPF (o BPF, il quale verrà tradotto in eBPF direttamente dal kernel), i risultati sarebbero per lo più gli stessi: monitoraggio tramite SOCKET_FILTER nel caso di Wireshark e Tcpdump, gestione di rete tramite Iptables e Tc. L'unica differenza si ha con Seccomp: anche se utilizza programmi eBPF per l'analisi delle system call, è comunque in grado di filtrare funzioni (mentre se si utilizzasse direttamente eBPF le si potrebbe solo monitorare). Questo perché ha un suo modulo del kernel che è in grado di reagire in modo diverso a seconda del valore di ritorno del programma eBPF e non perché quest'ultimo ha più permessi. Ne si deduce quindi che se si volessero applicare regole di filtraggio tramite programmi eBPF di tipo KPROBE o TRACEPOINT lo si potrebbe fare, ma si dovrebbe ricorrere a un modulo d'appoggio (per filtraggio interno al kernel) o fare in modo che il programma eBPF invii informazioni al programma di livello utente per poi reagire da quest'ultimo.

3.7 Definire regole di monitoraggio/filtraggio

Mettere in pratica meccanismi di sicurezza in tempo reale con eBPF implica che vengano create delle regole (i programmi eBPF appunto) che definiscono quali eventi filtrare o almeno segnalare al livello utente. A differenza dell'uso di eBPF per il calcolo di prestazioni o di statistiche, dove basta

applicare formule matematiche per ottenere i risultati cercati, in questo caso risulta più complesso definire delle regole ideali; inoltre alcune regole sarebbero utili per tutti i tipi di sistemi, mentre altre servirebbero unicamente su sistemi specifici o in casi particolari. In tutti i casi comunque, è buona norma seguire le stesse indicazioni che si seguono nella creazione di policy di un qualunque tipo, per evitare che queste regole siano troppo restrittive (rischiando di bloccare alcuni aspetti di un sistema, nel caso del filtraggio, o di inviare troppe informazioni al livello utente, nel caso del monitoraggio) o troppo permissive (rischiando che siano facilmente aggirabili).

Prima di tutto conviene, dove possibile, creare una whitelist rispetto a una blacklist: è meglio creare delle regole che definiscano ciò che è permesso rispetto a ciò che non lo è. Supponiamo per esempio che un programma debba aprire un file, scrivere qualcosa in esso e poi chiuderlo e che si voglia essere certi che un possibile attaccante non sfrutti una vulnerabilità in questo programma per leggere un file da disco. Nel creare una regola che permette a questo programma di chiamare solo le system call 'open', 'write' e 'close' (magari tramite Seccomp), non si dà altra scelta al programma, e quindi al possibile attaccante, che di usare queste tre funzioni rendendo quindi impossibile la lettura di un file, ma se venisse creata una regola che esplicitamente evita a questo programma di chiamare la funzione 'read', l'attaccante non potrà leggere un file tramite essa, ma potrà sempre usare la system call 'mmap' per mappare il file in memoria e leggerlo direttamente da lì. Ne si deduce quindi che la whitelist è la soluzione migliore perché più sicura, ma ovviamente è anche la più complessa da creare. Infatti nella realtà difficilmente un programma funzionerà semplicemente tramite l'uso di tre system call. Questo implica che la whitelist debba contenere maggiori regole e preferibilmente non solo basate sulle funzioni da chiamare, ma anche sui parametri che esse ricevono. Un esempio potrebbe essere un programma che tra le altre cose apre almeno un file, lo legge e lo chiude. In questo caso, la 'read' dovrà per forza fare parte delle system call permesse e un attaccante potrebbe chiamarla per leggere un file privato. A questo punto si dovrebbe creare una regola che permetta alla 'open' di aprire solamente i file che quel dato programma ha il permesso di aprire (whitelist) o che non permetta alla 'open' di aprire file privati (blacklist). Di nuovo, si nota come il primo caso sia migliore del secondo perché non è aggirabile in alcun modo. Ovviamente ci sono casi in cui non è possibile creare una whitelist: un programma che utilizza una certa funzione passandogli parametri sempre diversi (magari chiesti all'utente o generati in qualche modo); oppure un programma che è talmente vasto da rendere la creazione della whitelist troppo complessa e magari anche pesante in fase di elaborazione, cosa che diminuirebbe le prestazioni del processo stesso. In questo caso sarà necessario creare una blacklist, facendo però attenzione a inserire tutti i modi in cui una certa cosa può essere ottenuta, non solo il primo che viene in mente o il più utilizzato dagli attaccanti. Quindi nel primo esempio (evitare a un attaccante la possibilità di leggere un file), nella blacklist dovranno essere inserite almeno le system call 'read' e 'mmap'; nel secondo esempio invece, bisognerà inserire delle regole per bloccare l'apertura di file che si chiamano in un certo modo (facendo attenzione al fatto che lo stesso file può essere aperto da diverse cartelle, quindi con path diverso, e magari tramite un link, quindi con nome completamente diverso) e bloccare, almeno in parte, le system call 'link', 'linkat', 'symlink' e 'symlinkat' per evitare che un attaccante le possa utilizzare per creare nuovi link ai file privati per poi aprirli tramite essi. Ovviamente questo ragionamento non riguarda solamente le system call, ma tutte le policy, anche quelle legate alla rete di un sistema.

Un'altra cosa fondamentale nella definizione di regole di sicurezza, è trovare il modo migliore per effettuare determinati controlli. Questo può essere molto utile per motivi di ottimizzazione: filtrare traffico di rete direttamente sullo stack di rete è molto più utile che lasciare passare ogni pacchetto e poi filtrare le richieste una volta raggiunto il livello applicativo, cosa che invece potrebbe essere obbligatorio in determinate situazioni: supponiamo per esempio che si voglia evitare una specifica richiesta verso un certo server web (magari perché la risorsa che si otterrebbe con tale richiesta è privata e ne si vuole bloccare l'accesso dall'esterno); in questo caso si potrebbe creare un programma eBPF che analizzi direttamente il contenuto dei pacchetti e filtri in base alla risorsa richiesta. Questo è certamente giusto nel caso di connessioni non cifrate, ma se nella comunicazione si utilizzasse HTTPS, il contenuto del pacchetto sarebbe cifrato e la richiesta non potrebbe essere analizzata come si deve. Per questo motivo qui converrebbe utilizzare un programma eBPF di tipo KPROBE: come spiegato in precedenza, ci si potrebbe attaccare a tutte le funzioni di libreria che vengono utilizzate dai programmi in esecuzione per la cifratura del traffico e analizzare i buffer che esse ricevono, ottenendo quindi il contenuto dei pacchetti in chiaro.

In conclusione, eBPF è uno strumento particolarmente potente anche nel campo della sicurezza in tempo reale, ma bisogna fare attenzione a sfruttarlo nel miglior modo possibile per evitare che un attaccante aggiri le regole definite.

Capitolo 4

Gestire una rete di container con eBPF

4.1 Introduzione

Ci sono situazioni in cui un server potrebbe voler offrire dei servizi ai client connessi tramite l'utilizzo di sistemi virtualizzati: sistemi che risiedono all'interno della stessa macchina del sistema principale, ma virtualmente isolati da esso. Questo tornerebbe molto utile per esempio su server che contengono più siti web che devono rimanere separati tra di loro o in situazioni in cui si potrebbe dover ricorrere a un ripristino il più veloce possibile del sistema (magari perché infettato da un malware). Infatti, dover gestire regole di isolamento dei siti web nel primo caso o dover ripristinare ogni volta l'intero sistema nel secondo, potrebbe non essere la soluzione ideale. Per questo motivo si ha cominciato a suddividere un server in una serie di elementi virtualizzati ciascuno con il proprio scopo e i propri contenuti statici, dinamici e/o personalizzati a seconda del client che li sta richiedendo.

Il primo tipo di sistema virtualizzato che è nato è quello che si ottiene tramite una macchina virtuale. Come dice il nome stesso, con essa si virtualizza un'intera macchina, sia a livello software che hardware, cosa che ha certamente il vantaggio di avere sistemi completamente isolati (la virtualizzazione dell'hardware li fa sembrare installati sue due macchine diverse) con addirittura la possibilità di installare sistemi operativi diversi tra loro, ma ha ovviamente lo svantaggio di avere un carico di elaborazione elevato. E più macchine virtuali sono eseguite in parallelo, più risulta pesante per il vero hardware del sistema e più si riducono le prestazioni di tutti i sistemi, compreso quello reale. Per questo motivo è nato il secondo tipo di ambiente virtualizzato: i container. Essi permettono di isolare un certo servizio dal sistema vero e proprio in modo molto leggero, utilizzando tecnologie offerte dal sistema su cui sono stati installati. In Linux per esempio, i container si appoggiano ai Cgroups (Control Groups) e ai Namespaces. I primi sono un insieme di meccanismi che permettono di aggregare determinati processi (compresi tutti i loro futuri figli) in gruppi gerarchici ai quali vengono assegnate risorse predefinite; lo scopo è decidere a priori quali e quante risorse assegnare a questi gruppi e poi monitorarne l'utilizzo. Queste limitazioni possono riguardare la quantità di memoria fisica del kernel o totale a cui i processi possono accedere, la percentuale di CPU che possono utilizzare, i limiti in lettura e scrittura verso dispositivi di I/O e l'utilizzo della rete (gestita tramite il Traffic Control). I Namespaces invece permettono di limitare ciò che viene mostrato a un dato processo (e quindi ciò che può utilizzare) o a un gruppo di processi e ne esistono diversi tipi: i PID Namespaces, per esempio, servono per limitare i processi in esecuzione che un dato processo può vedere, il primo processo inserito in un namespace di questo tipo riceverà quindi un PID pari a 1 e penserà di essere il primo processo dell'intero sistema; i Network Namespaces invece permettono di isolare la rete di determinati processi dalla rete del sistema e quindi i processi in essi hanno proprie interfacce di rete, tabelle di routing, indirizzi IP, porte, ...; i Mount Namespaces servono a isolare il file system, creando delle cartelle di root virtuali; e così via. Utilizzando queste due tecnologie, i container sono appunto in grado di isolare determinati processi

dal sistema. A differenza delle macchine virtuali, qui si isolano i processi, non l'intero sistema; questo implica che i container siano molto più leggeri delle macchine virtuali (permettendo a un sistema di poter gestire molti più servizi isolati a un costo di elaborazione ridotto) e per questo motivo al giorno d'oggi i server che vogliono offrire servizi virtualizzati e isolati utilizzano per lo più essi.

Questa maggiore leggerezza però comporta che l'isolamento dei container sia molto minore di quello delle macchine virtuali. Infatti, siccome i container non virtualizzano anche il sistema operativo e l'hardware (a differenza delle macchine virtuali), devono usare direttamente quelli del sistema vero e proprio. Questo implica che un processo eseguito in un container utilizza le risorse del kernel del vero sistema (non possono essere quindi virtualizzati programmi di sistemi operativi diversi da quello principale), ne consegue che una vulnerabilità o la mal configurazione di un container può diventare un problema per l'intero sistema. Per questo motivo nell'applicare tecniche di sicurezza in tempo reale in un sistema, bisogna anche tenere conto di ciò che accade nei container presenti: per quanto riguarda il monitoraggio o il filtraggio delle funzioni utilizzate dai programmi interni a un container, non risulta esserci differenza tra le tecniche usate sul sistema principale (si tratta sempre di processi, anche se isolati); molto più interessante invece è la gestione del traffico di rete perché nella maggior parte dei sistemi in cui vengono offerti servizi all'esterno tramite risorse virtualizzate, non si trova unicamente un container, ma un'intera rete di essi. Questa rete può servire per bilanciare il traffico o per isolare le varie risorse contenute nei vari container. In ogni caso spesso queste risorse vanno collegate tra loro e per farlo viene solitamente creata una rete di container interna al sistema stesso (connessa tramite un virtual switch). Ovviamente questa rete potrebbe essere completamente isolata dalle altre del sistema, ma generalmente non è così visto che i servizi contenuti nei container sono per i client che si connettono al server, non per il sistema locale. Ci deve quindi essere un qualche collegamento tra un client che richiede qualcosa al server e la rete di container. Proprio questo collegamento può diventare un punto di attacco da remoto.

In conclusione, vista l'importanza che i container stanno acquisendo e il numero sempre più elevato di server web che offrono servizi tramite essi, in questo capitolo si analizzerà un esempio realistico (anche se molto semplificato) di un server che offre dei servizi contenuti in una rete di container. Questi servizi saranno volutamente vulnerabili a alcuni attacchi così da poter dimostrare quali possono essere alcune delle problematiche (perché ovviamente altri problemi possono esistere in servizi diversi da quello preso in esame), quali possono essere le conseguenze di queste problematiche e soprattutto quali possono essere le soluzioni, tra cui ovviamente quelle definibili tramite eBPF stesso.

4.2 Caso studio

In figura 4.1 è possibile trovare una rappresentazione del caso studio preso in esame per l'analisi delle reti di container. Si ha un sistema collegato alla rete Internet e che mostra verso l'esterno due porte aperte (la porta 80 e la 22). Dall'interno invece si vede una rete di container nella quale si trovano innanzitutto due container raggiungibili dall'esterno grazie a delle regole di routing che reindirizzano tutti i pacchetti diretti verso le porte 80 e 22 del sistema principale rispettivamente verso la porta 80 del load balancer e la porta 22 del pannello amministratore; e poi si vedono una serie di server web contenenti tutti lo stesso sito web (i cui file si trovano nel volume rappresentato, condiviso non solo da tutti i server, ma anche dal sistema principale e dal pannello amministratore) tutti in grado di comunicare con il database contenuto in un altro container (al quale anche l'amministratore ha accesso) e il cui utilizzo è bilanciato dal load balancer. Di conseguenza, dall'esterno il sistema sembra un server web qualunque (con due porte aperte, di cui una accessibile 'solamente' dall'amministratore della rete) anche se in realtà offre dei servizi bilanciati e divisi su più container.

In figura 4.2 sono invece visualizzati i comandi necessari per la creazione (il primo) e la distruzione (il secondo) della rete di container appena descritta. Essi sono da eseguire dalla cartella `Programmi/rete_container` dopo aver installato le risorse necessarie (definite nella guida dell'utente, capitolo 8). Maggiori informazioni riguardo al tool utilizzato verranno date nelle guide dell'utente (capitolo 8) e del programmatore (capitolo 9).

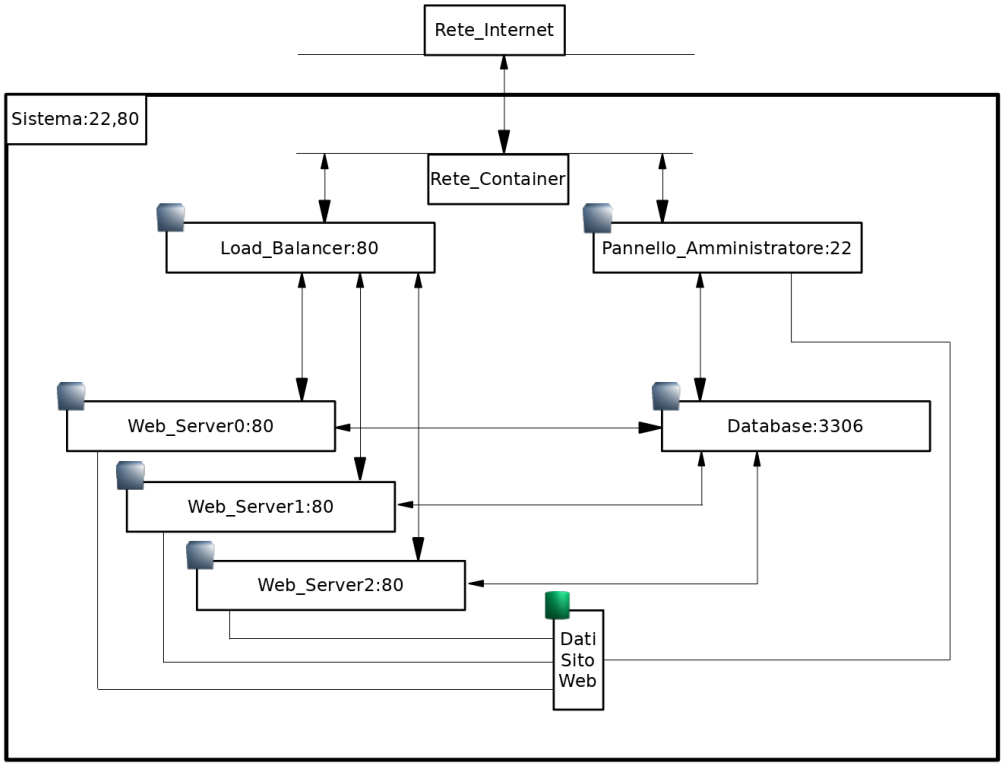

Figura 4.1. Rappresentazione del caso studio per il filtraggio in reti di container.

```
sudo ./net_manager.sh start source_init.pp 0
sudo ./net_manager.sh stop source_exit.pp 0
```

Figura 4.2. Comandi per la creazione/distruzione della rete di container presa in esame.

4.3 Analisi del caso studio

Supponendo di avere eseguito il comando definito sopra, si passa ora a analizzare il sistema che contiene la rete risultante dall'esterno, dal punto di vista di un possibile attaccante. La prima operazione da fare è scoprire l'indirizzo IP del target. In questo caso ovviamente si tratta dell'indirizzo del sistema su cui è stata eseguita la rete di container (qui si utilizzerà l'indirizzo '192.168.1.5'). Poi si passa a analizzare il target stesso alla ricerca delle porte aperte e dei servizi che esse offrono. Tra i tool più utilizzati per questo scopo c'è Nmap che permette di ottenere il risultato voluto grazie per esempio al comando presente in figura 4.3: risultano aperte la porta 22 con il servizio SSH attivo e la porta 80 con un servizio HTTP (come previsto dalla descrizione del caso studio). A questo punto ci si sposta a analizzare un servizio per volta.

Nell'analizzare la porta 80, si visita l'indirizzo http://192.168.1.5 e si incontra subito una pagina molto semplice che contiene unicamente del testo, l'indirizzo IP del server da cui è stata scaricata la pagina (per dimostrare la presenta del load balancer) e un link. Vista la sua semplicità non sembra avere alcuna vulnerabilità; si passa quindi a cliccare sul link e a essere rediretti alla risorsa '/public.php' che contiene una lista di elementi che sembrano essere scaricati da un database e un form che permette di cercare uno degli elementi elencati tramite ID. Di nuovo non sembra esserci alcun problema. Più interessante invece risulta essere il form che redirige verso la risorsa '/vulnerable.php': inserendo un ID valido si ottengono tutte le informazioni relative a quell'ID; non inserendo niente o inserendo un ID invalido non si ottiene alcuna informazione; inserendo

```
nmap -sV -p- 192.168.1.5
```

Figura 4.3. Comando per la scoperta delle porte aperte sul sistema target.

```
...
    <h1 align="center">Risultato Ricerca</h1>
    <p>E' stato cercato l'ID: <?php echo $_GET["id"]?></p>
    <?php
        $server = "172.17.0.5";
        $username = "ws_user";
        $password = "ws_user_pass";
        $database = "web_server_db";

        $conn = new mysqli ($server, $username, $password, $database);
        if ($conn->connect_error) {
            die ("Connection failed: " . $conn->connect_error);
        }
        $id = $_GET["id"];
        $sql = "SELECT * FROM resources WHERE id = '$id';";
        $result = $conn->query ($sql);
        if ($result->num_rows > 0) {
            while ($row = $result->fetch_assoc ()) {
                echo "ID: " . $row ["id"] .
                        "<br>Title: " . $row ["title"] .
                        "<br>URL: " . $row ["link"] .
                        "<br>Descrizione: " . $row ["notes"] . "<br><br>";
            }
        }
        $conn->close ();
    ?>
...
```

Figura 4.4. Pagina vulnerabile del sito web allegata alla tesi come `Programmi/rete_container/web_server/source/vulnerable.php`.

delle stringhe speciali è possibile notare che l'input non viene filtrato (cosa che potrebbe portare a problemi di sicurezza); in tutti i casi viene anche stampato l'ID cercato. Il codice sorgente è presente in figura 4.4: dopo aver stampato l'ID cercato, vengono inizializzati i dati necessari alla connessione al database, viene letto il parametro ricevuto dal form e viene inserito senza alcun controllo o filtraggio all'interno della query da inviare al database, il quale ritornerà i dati richiesti che verranno poi stampati a video.

Avendo notato che l'input al form non viene filtrato prima di essere stampato a video nè prima di essere inserito nella query sql, un attaccante a questo punto potrebbe sfruttare la cosa in due modi diversi: attaccare il database tramite SQL Injection; attaccare i client (amministratore compreso) con il Reflective XSS. Nel primo caso l'idea è sfruttare il fatto che l'input viene inserito in una query SQL trasmessa al database ricevendo il suo risultato come messaggio sulla pagina di risposta. Non essendoci alcun controllo sull'input inserito dall'utente, un attaccante ha il controllo sulla query e può quindi chiedere al database le informazioni che gli interessano e leggerle sulla pagina di risposta. Un esempio di questo si può trovare in figura 4.5: avendo inserito nel form presente alla pagina '/public.php' la stringa che è stata stampata all'inizio della pagina (compresi l'apice iniziale, necessario per chiudere la clausola where della vera query, e il cancelletto finale,

necessario per commentare la parte finale della vera query) si ha ottenuto come risultato l'hostname del database, l'utente che ha eseguito la richiesta, il nome del database e la versione di Mysql installata. Per quanto invece riguarda l'XSS, siccome l'input dell'utente viene stampato a video senza nessuna forma di filtraggio, è possibile inserire nel form del codice html/javascript che verrà interpretato dal browser sulla pagina di risposta. Un esempio del funzionamento è rappresentato in figura 4.6: si può notare che avendo inserito nel form della pagina '/public.php' la stringa contenuta nella barra dell'URL (da dopo 'id=' al fondo, con al posto del '%2F' l'equivalente ASCII, quindi il carattere '/') si ha ottenuto un messaggio di alert. Ovviamente questi sono solamente due semplici esempi, nel caso di un attacco reale, con il primo tipo di attacco si potrebbero leggere tutti i dati contenuti nel database (credenziali, dati personali degli utenti, cookie, chiavi, numeri di carte di credito... sperando per gli utenti che i dati sensibili non siano salvati in chiaro) e nel secondo caso si potrebbero rubare cookie, si potrebbe reindirizzare la pagina verso un sito web malevolo, si potrebbe spingere un client a installare del malware... e così via.

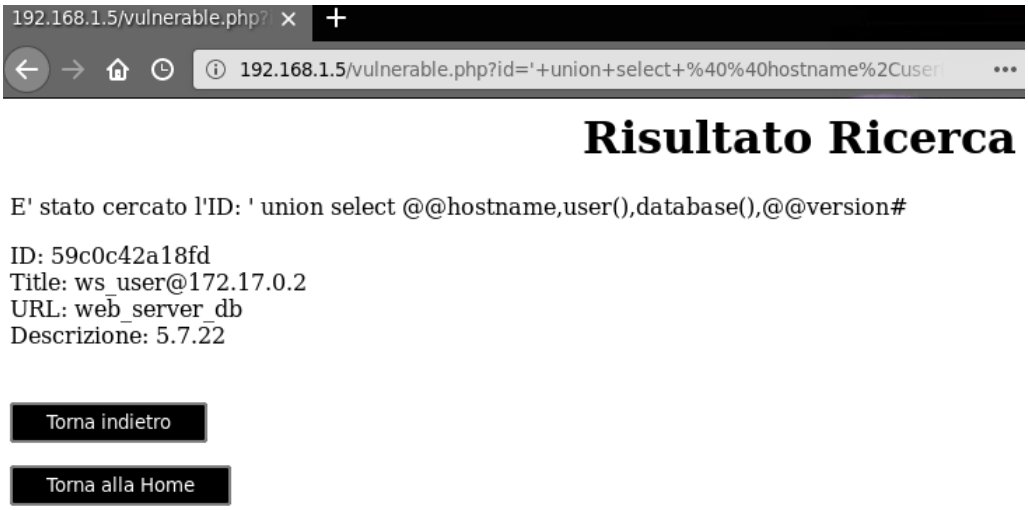

Figura 4.5. Dimostrazione del successo dell'attacco SQL Injection.

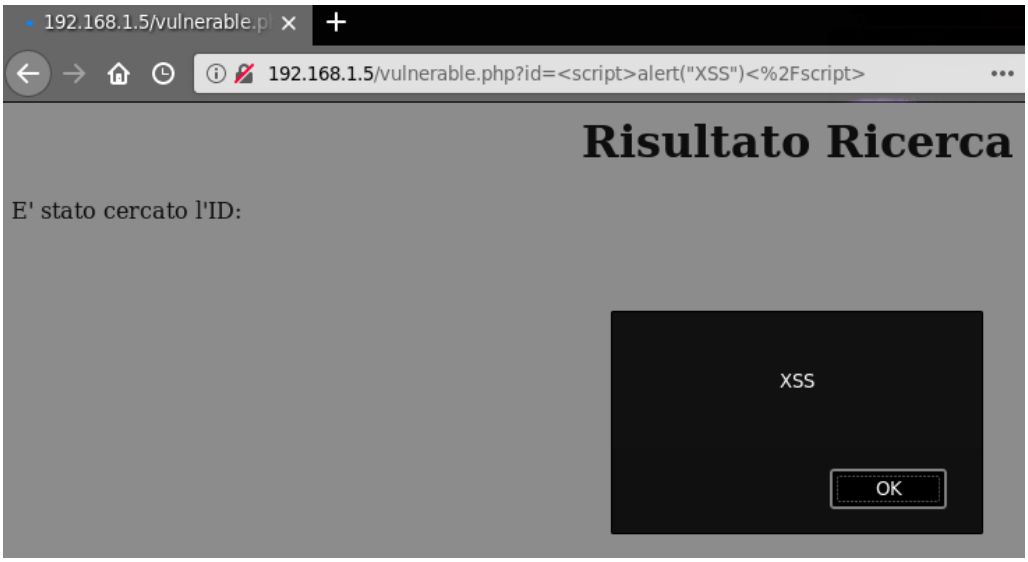

Figura 4.6. Dimostrazione del successo dell'attacco Reflective XSS.

A questo punto, avendo analizzato ogni pagina 'pubblica' del sito web, l'ultima cosa che rimane da fare è cercare risorse che potrebbero non essere direttamente connesse alle pagine visitate

```
dirb http://192.168.1.5 -X .php
```

Figura 4.7. Comando per il fuzzing delle risorse del sistema target.

```
ssh root@192.168.1.5
```

Figura 4.8. Comando per la connessione al server SSH presente sul sistema target.

(tramite link o pulsanti per esempio), ma che potrebbero comunque esistere permettendo quindi a un client di farne la richiesta (direttamente) e riceverle in risposta. Per poter scoprire se esistono risorse di questo tipo, si può innanzitutto analizzare il file robots.txt (un file che indica quali parti del sito non sono accessibili ai crawler dei motori di ricerca) e poi fare fuzzing. La prima soluzione in questo caso non porta da nessuna parte (siccome il file non esiste). Per mettere in pratica la seconda viene invece usato Dirb, un tool che permette di cercare oggetti presenti su un sito web a partire da un dizionario. Eseguendolo come in figura 4.7 si ottiene un risultato che dimostra l'esistenza di una risorsa di nome 'private.php' (oltre a due delle pagine già analizzate). Indirizzando il browser verso l'URL http://192.168.1.5 si trova una pagina di solo testo, quindi non vulnerabile, contenente però informazioni molto interessanti: lo username e la password del servizio SSH e quindi del pannello di amministrazione della rete. Ovviamente questo è solo un esempio, raramente una risorsa privata conterrà informazioni così sensibili, ma potrebbe comunque contenere informazioni molto interessanti.

Per poter invece analizzare la porta 22 si esegue il comando raffigurato in figura 4.8 per connettersi alla prima porta aperta sul sistema e notare che la connessione va a buon fine. Ovviamente il servizio SSH chiederà una password e si potrà usufruire della shell interna al sistema solamente se la si conosce (o indovina, nel caso di password semplice). Se il servizio fosse aggiornato e la password molto robusta, potrebbe essere 'impossibile' per un attaccante entrare direttamente e quindi dovrebbe utilizzare altre tecniche per cercare di rubare la password richiesta (per esempio scoprire la pagina '/private.php' descritta prima). In ogni caso, una volta trovata la password ('admin_pass') un attaccante potrebbe entrare e ritrovarsi all'interno del sistema con gli stessi permessi che ha l'amministratore: può quindi analizzare l'intero sito web per scoprire come funziona, quali algoritmi di cifratura/hashing utilizza, con quali chiavi/seed; può modificare il sito web per attaccare tutti i client che si connettono a esso; può usare le credenziali del database trovate nel sorgente del sito web e usarle per gestire il database con gli stessi permessi del sito web; se scoprisse le credenziali di amministrazione del database potrebbe gestirlo con maggiori permessi; potrebbe cercare di attaccare gli altri container; potrebbe cercare di attaccare il sistema principale tramite vulnerabilità del kernel o del sistema di isolamento (se ne esistessero e l'attaccante le conoscesse) oppure inserendo del malware nel volume montato (la cartella contenente i file sorgente del sito web) nella speranza che qualcuno li esegua o spingendo qualcuno a eseguirli; ... Comunque, da notare è il fatto che se fino a ora un attaccante non poteva sapere che il server utilizza una rete di container per offrire i suoi servizi, ora lo può immaginare perché la shell che ha ottenuto è ovviamente isolata dal sistema principale (cosa che come minimo si capisce dal fatto che i processi attivi sono pochissimi e il PID 1 è assegnato a sshd e non a systemd).

Un'ultima considerazione da fare riguardo al pannello di amministrazione (se non tenuto sotto controllo) è che non è l'unico container dal quale un attaccante potrebbe attaccare gli altri presenti nella stessa rete: se il sito web avesse una vulnerabilità che permette a un attaccante di eseguire comandi di sistema (in questo caso comandi interni al container del sito web), l'attaccante potrebbe entrare in uno dei container contenenti il sito web e da lì attaccare gli altri. Questo perché di default, quando si crea una rete di container, tutti i container possono parlare con tutti gli altri: per testare questo può essere usato il comando descritto in figura 4.9 che scarica la Home Page del 'web-server1' (nome assegnato al container che rappresenta il 'Web_Server1', che ha indirizzo

```
sudo docker exec web-server0 curl "http://172.17.0.3"
```

Figura 4.9. Test per dimostrare la possibilità di connessione tra il 'Web_Server0' e il 'Web_Server1'.

IP '172.17.0.3') dal 'web-server0' (nome assegnato al container che rappresenta il 'Web_Server0') nonostante nella logica del caso studio i tre server non siano connessi tra di loro (perché sono uno la copia dell'altro e non c'è alcun motivo per connetterli). Ciò implica che il punto di collegamento tra il mondo esterno e la rete di container non è l'unico a essere vulnerabile, ma anche la rete stessa: se un attaccante entrasse in un container qualunque potrebbe scoprire gli indirizzi IP degli altri, scoprire quali porte aperte e quali servizi ognuno di essi ha attivi, sniffare il traffico tra i container con la possibilità di scoprire richieste private, api vulnerabili, ... eccetera.

Tutto ciò può sembrare ovvio, ma in realtà per il momento quando viene creata una rete di container spesso ci si fida dei componenti che contiene perché si trovano tutti all'interno del sistema (e quindi non sembra un problema collegarli tramite connessioni non cifrate) e isolati dal sistema principale; si pensa quindi che proteggendo il punto di collegamento tra la rete e il mondo esterno si stia proteggendo l'intera rete. Come spiegato però questo non è abbastanza.

Ricapitolando, nel caso preso in esame un attaccante esterno al sistema potrebbe scoprire due vulnerabilità puramente di livello 7, SQL Injection e Reflective XSS, una di livello 3-7, la risorsa privata, e una (non proprio una vulnerabilità, ma comunque utile da tenere sotto controllo) di livello 3, il servizio SSH. Ovviamente se l'amministratore del server conoscesse questi problemi potrebbe risolverli facilmente: filtrare l'input del form vulnerabile in modo che nessuno sia in grado di mettere in pratica uno dei due attacchi scoperti; limitare l'accesso alla pagina privata e al servizio SSH al solo amministratore. Supponendo però di non conoscere queste vulnerabilità (perché un sito reale potrebbe essere molto più grande, con molti più file e form e anche solo uno di essi potrebbe contenere qualche vulnerabilità) e di voler applicare un firewall in grado di proteggere il sistema da possibili attacchi dei tipi appena elencati, verranno ora analizzate possibili soluzioni e poi verranno messe in pratica per dimostrare il loro funzionamento nel caso preso in esame.

4.4 Programmare un firewall con eBPF

Come anticipato nel capitolo precedente, a livello di rete, con eBPF è possibile creare un firewall direttamente dentro al kernel che filtri, o più in generale gestisca, il traffico di rete là dove necessario. Per quanto riguarda server che offrono servizi tramite container (come nel caso preso in considerazione), siccome i client accedono al sistema tramite l'interfaccia di rete connessa alla rete pubblica, si può pensare di installare dei programmi eBPF in quel punto. Questo è molto utile per limitare gli attacchi dall'esterno, ma se per caso un attaccante riuscisse a entrare in uno dei container, i controlli non avrebbero più alcun effetto nella rete che li connette (come spiegato in precedenza). Ovviamente se il container infettato non è connesso a nessun altro, gli unici attacchi possibili sarebbero quelli diretti al kernel (ma questo non ha a che fare con il networking). Se invece fosse connesso allo switch a cui sono collegati tutti i container della rete interna al sistema, l'attaccante potrebbe a questo punto cercare di arrivare agli altri facendo richieste mirate. Per evitare questo, l'ideale è avere anche dei programmi eBPF su quest'altra interfaccia (anche se virtuale) e tenere sotto controllo tutte le connessioni e le richieste tra i container connessi a essa. Ovviamente, se il traffico tra i container non fosse cifrato (scelta più che lecita visto che lo scambio di informazioni avverrebbe unicamente all'interno del sistema stesso) ci si potrebbe fermare qui; ma nel caso di traffico cifrato, per analizzare le richieste bisognerebbe aggiungere controlli sulle funzioni di cifratura/decifratura dei pacchetti e siccome queste funzioni sono interne ai vari container, in questo caso risulterebbe indispensabile aggiungere anche delle regole eBPF in ognuno di essi.

Per quanto riguarda i tipi di programma eBPF da utilizzare, nel caso di controllo del traffico cifrato, l'ideale sarebbe il tipo PROBE (o TRACEPOINT se le funzioni di cifratura/decifratura

li contenessero). Ovviamente questo permetterebbe unicamente di monitorare le richieste cifrate, si dovrebbe poi applicare un ulteriore meccanismo d'appoggio per il filtraggio (se necessario). Per tutto il resto, la gestione del traffico di rete (o almeno il filtraggio), si potrebbe invece scegliere uno tra SCHED_CLS, SCHED_ACT e XDP. Non c'è una scelta giusta o sbagliata, ma solo la scelta ottimale a seconda di come si vuole implementare il firewall: se l'idea è quella di differenziare il traffico a seconda dell'azione da prendere sui vari pacchetti allora si può usare uno dei primi due; se invece si vuole analizzare e modificare, dove necessario, direttamente i pacchetti conviene usare XDP. Un esempio del primo caso potrebbe essere un firewall che vuole dividere i pacchetti in quelli da inoltrare così come sono, quelli da buttare e quelli sui quali eseguire ulteriori operazioni; oppure un firewall integrato in un sistema di bilanciamento del traffico o di suddivisione in code di pacchetti a seconda della loro priorità. In questi casi si potrebbe usare il tipo SCHED_CLS per analizzare i pacchetti e dividerli in code o il tipo SCHED_ACT per decidere quale operazione effettuare sui pacchetti presenti in una data coda. Un esempio del secondo caso invece è l'implementazione di un firewall più a basso livello: i pacchetti vengono inoltrati o buttati così come sono oppure si vanno a modificare i byte del pacchetto direttamente dalla funzione di analisi e poi si inoltra la versione aggiornata (provocando indirettamente un reindirizzamento o una qualunque altra forma di manipolazione dei pacchetti).

In entrambi i casi, così come per l'analisi del traffico cifrato, bisognerebbe scrivere tutti i programmi eBPF necessari facendo attenzione di scegliere il miglior punto in cui inserire ciascuno e soprattutto tenendo bene presente le norme di creazione delle regole per questo tipo di sicurezza definite nel capitolo precedente (whitelist rispetto a blacklist). Infine, siccome scrivere programmi eBPF per sistemi complessi può diventare lungo (non troppo difficile, ma comunque lungo), si potrebbe decidere di usare eBPF indirettamente, tramite un programma che date delle regole di alto livello generi e gestisca i programmi eBPF automaticamente (come Tcpdump per il monitoraggio della rete). Nel caso della gestione del traffico di rete in sistemi che contengono container, ciò può essere fatto tramite Cilium.

4.5 Cilium per la sicurezza dei container

Cilium [10] è un programma che permette di gestire un'intera rete di container sia dal punto di vista del routing che dal punto di vista delle policy di networking. Infatti per poter gestire un container con questa tecnologia bisogna innanzitutto connetterlo alla rete Cilium (creata in precedenza) e poi assegnargli un nome e un'etichetta: il nome verrà usato come hostname del container stesso (risolvibile tramite una richiesta DNS nell'indirizzo IP assegnato al container nel momento in cui è stato eseguito), e quindi per il routing, mentre l'etichetta verrà usata durante la definizione delle regole di filtraggio (se necessarie). Ovviamente il nome può essere risolto nel relativo indirizzo solamente da dentro la rete di container; dal resto del sistema bisogna per forza usare l'indirizzo IP (a meno che non ci si appoggi a qualche altra soluzione).

Inizialmente la rete di container risulta privata e quindi tutti i container sono in grado di connettersi a tutti gli altri su ogni porta e di fare qualunque richiesta [11]. E' però possibile connettere la rete al mondo esterno definendo delle regole di routing sull'host principale (tramite altre tecnologie perché Cilium non si occupa direttamente di questo). In questo caso (ma non solo) potrebbe diventare molto importante definire delle regole che permettano a determinati container di scambiare certe informazioni solamente con altri determinati container. Per questo motivo Cilium permette di definire delle regole di livello 3, 4 e/o 7 in formato JSON che verranno tradotte in programmi eBPF e caricati nel kernel in modo automatico. Da sottolineare è il fatto che nell'istante in cui viene installata una regola su un certo 'endpoint' (sinonimo di container utilizzato nell'architettura Cilium), quella regola diventa automaticamente una whitelist per quell'endpoint. Per esempio se si definisse una regola per permettere al container1 di ricevere traffico in ingresso dal container2, essa verrebbe installata sul container1 come 'il container1 può ricevere traffico solamente dal container2, ma può inviare traffico a tutti'; nessun altro container sarebbe quindi in grado di contattare il container1. Ovviamente si ha la possibilità di installare più regole sullo stesso endpoint e quindi di definire un firewall anche molto complesso. Un'ultima cosa da precisare riguardo alle policy è che Cilium permette il cosiddetto 'stateful connection tracking': se un container accetta in ingresso pacchetti in arrivo da un certo container, la stessa policy lascia passare anche i pacchetti di risposta

(quelli appartenenti al contesto della stessa connessione TCP/UDP); ovviamente vale anche per policy in uscita. Questo implica che non si è obbligati a installare due policy per fare comunicare due container, ma solamente una.

In figura 4.10 è rappresentata l'architettura di Cilium mentre le istruzioni su come crearla e su come utilizzarla si possono trovare nella guida dell'utente (capitolo 8). Come si può notare l'elemento centrale è il daemon, il quale gestisce l'intera rete di container eseguendo i comandi ricevuti dal Cilium CLI che è un tool da riga di comando che un utente può utilizzare per caricare nuove policy nella rete o rimuoverne di vecchie, per analizzare gli endpoint connessi alla rete, per monitorare la rete creata, per effettuare debug, ... Per quanto riguarda le policy di networking (ciò che più interessa qui) si può vedere che il daemon va a generare il codice eBPF da iniettare nella rete e poi lo attiva nel punto più ideale a seconda della funzione della policy: se la regola riguarda uno specifico endpoint,o meglio un'etichetta appartenente almeno a un endpoint interno alla rete cilium, il programma viene caricato direttamente sull'interfaccia di tutti i container descritti da quell'etichetta; se la regola ha a che fare con la connessione tra almeno un endpoint e il mondo esterno potrebbe venire installata direttamente sull'interfaccia di connessione tra la rete Cilium e il mondo esterno. Per quanto riguarda le policy che il daemon trasforma nei rispettivi programmi eBPF, come detto in precedenza, gli vengono passati tramite il Cilium CLI in formato JSON.

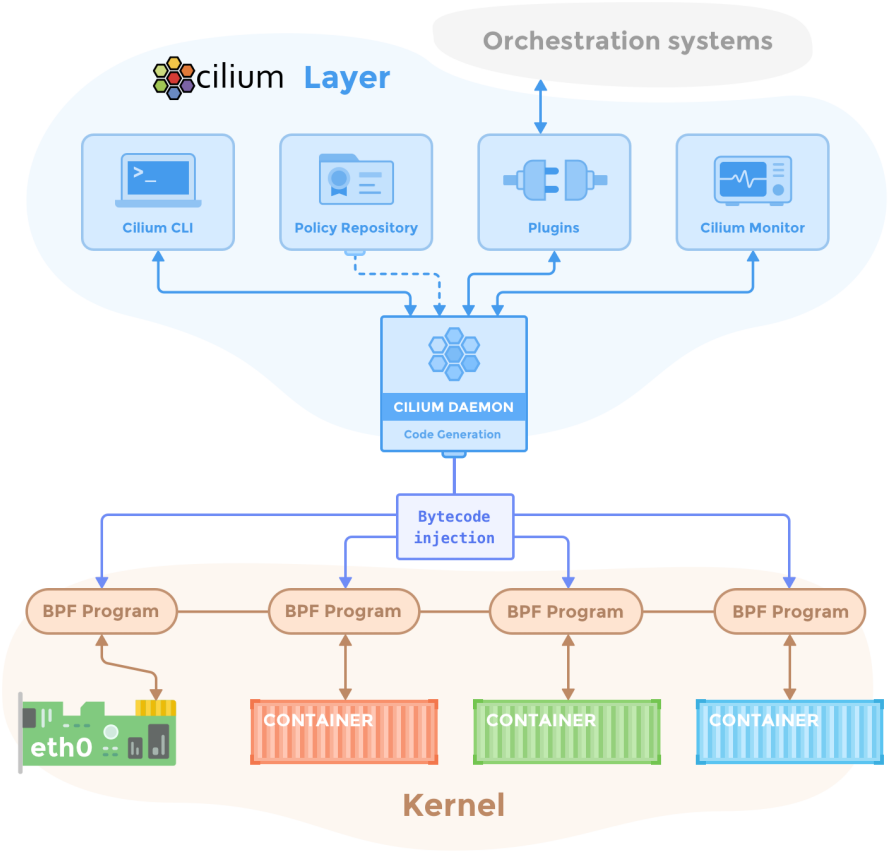

Figura 4.10. Architettura cilium (fonte: http://docs.cilium.io/en/latest/_images/cilium-arch.png).

In figura 4.11 si può trovare un esempio di una policy di livello 3-4: supponendo che nella rete associata a Cilium siano presenti almeno due container etichettati rispettivamente 'role=backend' e 'role=frontend', grazie a essa si crea una whitelist sul nodo di backend che permette traffico in ingresso solamente se il mittente è il frontend (permettendo, come detto sopra, anche il transito dei pacchetti di risposta) e solamente se sta utilizzando il protocollo TCP verso la porta di destinazione 80. Con questa sola policy caricata nella rete cilium, nessun altro container potrà contattare il backend. Più nel dettaglio, l'istruzione 'labels' associa delle informazioni alla policy appena

```
[{
    "labels": [{"key": "name", "value": "l4-rule"}],
    "endpointSelector": {"matchLabels":{"role":"backend"}},
    "ingress": [{
        "fromEndpoints": [
          {"matchLabels":{"role":"frontend"}}
        ],
        "toPorts": [
            {"ports":[ {"port": "80", "protocol": "TCP"}]}
        ]
    }]
}]
```

Figura 4.11. Esempio di regola di livello 3-4 da installare tramite cilium (fonte: http://docs.cilium.io/en/latest/policy/language).

```
[{
    "labels": [{"key": "name", "value": "cidr-l4-rule"}],
    "endpointSelector": {"matchLabels":{"role":"crawler"}},
    "egress": [{
        "toCIDR": [
            "192.0.2.0/24"
        ],
        "toPorts": [
            {"ports":[ {"port": "80", "protocol": "TCP"}]}
        ]
    }]
}]
```

Figura 4.12. Altro esempio di regola di livello 3-4 da installare tramite cilium (fonte: http://docs.cilium.io/en/latest/policy/language).

creata; l'istruzione 'endpointSelector' definisce su quale endpoint la policy dovrà essere installata; 'ingress' definisce una policy sul traffico in ingresso; 'fromEndpoints' definisce il livello 3 della policy e quindi definisce quali endpoint hanno il permesso di connettersi all'endpoint selezionato in precedenza (possono essere più di uno); e 'toPorts' definisce il livello 4 della policy. Ovviamente non è obbligatorio avere una policy che ricopra sia il livello 3 che il 4: se mancasse l'istruzione 'toPorts' (o 'fromPorts') sarebbe sottinteso il 'verso/da tutte le porte/protocolli'; se invece mancasse l'istruzione 'fromEndpoints' (o 'toEndpoints') sarebbe sottinteso il 'da/verso tutti gli endpoint'...

In figura 4.12 invece si può trovare un altro esempio di una policy di livello 3-4: supponendo che nella rete associata a cilium sia presente almeno un container etichettato 'role=crawler', grazie a essa si crea una whitelist su questo endpoint che permette solamente il traffico in uscita verso il blocco di indirizzi IP 192.0.2.0/24 e solamente se il protocollo usato è il TCP e la porta di destinazione è la porta 80. Le uniche istruzioni diverse rispetto all'esempio precedente sono 'egress', che indica una policy sul traffico in uscita, e 'toCIDR', che rappresenta il livello 3 della policy e che riguarda un blocco di indirizzi esterno alla rete di container (altrimenti sarebbe stata usata l'istruzione 'toEndpoints').

Infine in figura 4.13 si può trovare un esempio di una policy di livello 7: supponendo che nella rete associata a Cilium sia presente almeno un container etichettato 'app=myService', grazie a essa si crea una whitelist su questo endpoint che permette solamente il traffico in ingresso da tutti i container della rete (perché non è definita l'istruzione 'fromEndpoints') verso la porta 80, ma solo se

```
[{
    "labels": [{"key": "name", "value": "l7-rule"}],
    "endpointSelector": {"matchLabels":{"app":"myService"}},
    "ingress": [{
        "toPorts": [{
            "ports": [
                {"port": "80", "protocol": "TCP"}
            ],
            "rules": {
                "HTTP": [
                    {
                        "method": "GET",
                        "path": "/path1$"
                    },{
                        "method": "PUT",
                        "path": "/path2$",
                        "headers": ["X-My-Header: true"]
                    }
                ]
            }
        }]
    }]
}]
```

Figura 4.13. Esempio di regola di livello 7 da installare tramite cilium (fonte: http://docs.cilium.io/en/latest/policy/language).

il protocollo utilizzato è TCP e la cui richiesta HTTP è pari a 'GET /path1' oppure a 'PUT path2' con l'header 'X-My-Header' presente e settato a 'true'. Da sottolineare è il fatto che il path/risorsa della richiesta è definito/a in 'Go Regex' (POSIX Regex del linguaggio Go) che permette di definire un pattern anche molto complesso tramite l'uso di caratteri speciali. Nell'esempio in questione il '$' viene tradotto come 'al fondo della stringa', di conseguenza 'al fondo della richiesta HTTP'. Questo significa che la richiesta HTTP con metodo GET deve per forza finire per '/path1'; lo stesso discorso vale per il secondo caso (un esempio più complesso verrà analizzato nella prossima sezione).

Le regole così definite, possono essere passate al daemon tramite il Cilium CLI, il quale, come detto in precedenza, le trasformerà in programmi eBPF e li caricherà nel kernel. Da sottolineare è il fatto che non appena le regole saranno iniettate nella macchina virtuale entreranno in vigore. Questo significa che possono essere definite in qualunque momento senza dover riavviare né i container presenti nella rete cilium né la rete e neanche il daemon. Inoltre, grazie al fatto che le policy di networking sono basate sulle etichette e non sugli indirizzi IP, ogni volta che un nuovo container entra nella rete, il daemon controlla l'etichetta che lo descrive e gli associa tutte le regole di filtraggio precedentemente create (se ne esistono). Ovviamente regole di networking create successivamente, verranno automaticamente aggiunte a tutti i container etichettati in quel modo. Questo implica che cilium è in grado di gestire policy di networking anche in reti di container dinamici (che vengono creati e distrutti, anche a alte velocità); con altre soluzioni invece, soluzioni basate su indirizzi IP, ogni volta che un container viene distrutto bisognerebbe togliere le regole associate a esso e ogni volta che un nuovo container entra nella rete (o uno già attivo cambia per un qualunque motivo indirizzo IP), bisognerebbe aggiornare le regole associate a esso.

In conclusione, per il filtraggio di pacchetti di rete per motivi di sicurezza in reti di container, Cilium risulta un'ottima soluzione per la creazione e la gestione sia della rete stessa che del firewall per tenerla al sicuro, tutto ciò sfruttando programmi eBPF. Per questo motivo nel caso studio descritto in precedenza è stato deciso di utilizzare Cilium nella parte sperimentale anziché usare

direttamente eBPF.

4.6 Protezione del caso studio

Per poter proteggere la rete di container analizzata dalle vulnerabilità riscontrate, bisogna innanzitutto definire delle regole che blocchino gli attacchi descritti, ma senza creare problemi al funzionamento della rete stessa. Partendo dal livello 3 bisogna perciò limitare l'accesso al pannello di amministrazione solamente all'amministratore (per semplicità nel caso preso in considerazione si può immaginare che l'amministratore si connetta al pannello sempre con lo stesso indirizzo IP, per esempio l'indirizzo '192.168.1.10'); si deve permettere a chiunque di accedere al load balancer (perché il sito web deve essere pubblico); e si devono limitare le connessioni tra i vari container a quelle strettamente necessarie al funzionamento della rete: il load balancer deve poter scaricare le pagine web dai tre container contenenti le copie del sito web, i tre server web devono poter scaricare i dati dal database e l'amministratore deve poter accedere al database. Segue una lista contenente ciò che è appena stato descritto in forma più schematica:

- 192.168.1.10 <-> Pannello_Amministratore
- World <-> Load_Balancer
- Load_Balancer <-> Web_Server0
- Load_Balancer <-> Web_Server1
- Load_Balancer <-> Web_Server2
- Web_Server0 <-> Database
- Web_Server1 <-> Database
- Web_Server2 <-> Database
- Pannello_Amministratore <-> Database

Siccome si è deciso di usare Cilium e siccome i tre server web vengono utilizzati per lo stesso scopo e hanno le stesse regole di networking (come si può notare dalla lista sopra), allora si può pensare di assegnare a tutti e tre la stessa etichetta (cosa che viene fatta nel caso studio) così da poter semplificare la lista delle regole e ottenere il risultato seguente:

- 192.168.1.10 <-> Pannello_Amministratore
- World <-> Load_Balancer
- Load_Balancer <-> Web_ServerN
- Web_ServerN <-> Database
- Pannello_Amministratore <-> Database

Per mettere questo in pratica bisognerebbe eseguire l'architettura Cilium, lanciare tutti i container del caso studio dentro alla rete gestita da Cilium, scrivere queste regole nel formato JSON descritto nella sezione precedente e infine caricarle tramite il Cilium CLI. Grazie al tool implementato è possibile ottenere tutto questo in modo automatico: supponendo di avere ancora la rete del caso studio attiva, è possibile proteggerla con un firewall di questo tipo eseguendo il primo comando rappresentato in figura 4.14 dalla cartella `Programmi/rete_container` (per rimuoverlo bisogna invece eseguire il secondo comando). Questo fa sì che i container vengano spostati nella rete Cilium (dopo averla creata), che le regole definite dall'utente a alto livello (spiegate nella guida dell'utente, capitolo 8) vengano tradotte nelle rispettive regole JSON e che vengano caricate tramite il Cilium CLI. A questo punto se si cercasse di connettersi al pannello amministratore da

```
sudo ./net_manager.sh install source_init.pp 4
sudo ./net_manager.sh uninstall 4
```

Figura 4.14. Comandi per l'installazione e la rimozione del firewall cilium di livello 3 nella rete di container presa in esame.

un indirizzo IP diverso da quello di amministrazione (anche da localhost) non si otterrebbe alcuna risposta perché le richieste di connessione vengono filtrate; se si cercasse invece di visitare il sito web, esso continuerebbe a funzionare normalmente; infine se si cercasse di connettere due container che non dovrebbero poter comunicare l'uno con l'altro (per esempio due 'Web_Server'), di nuovo non si otterrebbe alcuna risposta.

Per quanto invece riguarda le vulnerabilità del sito web esse non possono ovviamente essere risolte con un firewall di livello 3. Bisogna quindi definire delle regole a livello applicativo (livello 7) che permettano a chiunque arrivi dal mondo esterno di visitare il sito web (e quindi di connettersi ai container contenenti esso), ma solamente all'amministratore di visitare la pagina '/private.php'; inoltre bisogna bloccare l'inserimento di input non lecito nel form vulnerabile. A questo punto però sorge un problema: siccome l'unico modo per accedere ai tre server web è attraverso il load balancer, in teoria si dovrebbero creare delle regole che mettano in pratica ciò che è appena stato descritto sul load balancer con mittente l'indirizzo IP del client che ha fatto la richiesta, ma per il momento cilium permette il filtraggio di livello 7 solamente tra container (e quindi non si può avere un indirizzo IP come sorgente o destinatario). Una soluzione per aggirare questo problema è quella di creare queste regole tra i vari web server e il load balancer, ma l'indirizzo IP dei pacchetti che giungeranno ai server web attraverso esso non saranno quelli dei client veri e propri, ma sempre e solo quello del load balancer. Quest'ultimo è stato perciò configurato in modo da aggiungere un header nelle richieste fatte ai server web del tipo 'X-Forwarded-For'. Questo header conterrà come campo l'indirizzo IP del client che ha effettivamente fatto la richiesta. Di conseguenza il firewall da definire (sempre supponendo che si utilizzi Cilium e che tutti i server web siano stati eseguiti con la stessa etichetta) potrebbe essere il seguente:

- Pannello_Amministratore + Header 'X-Forwarded-For: 192.168.1.10' <-> Web_ServerN
- Pannello_Amministratore <-> Web_ServerN solo se la richiesta è valida

che implica che solamente l'amministratore sia in grado di fare qualunque richiesta al server web; per tutti gli altri la richiesta non viene filtrata solamente se rispetta il filtro desiderato, rappresentato dalla stringa Regex '$\hat{}$/$|\hat{}$/index.php$|\hat{}$/public.php$|\hat{}$/vulnerable.php(\\?id=)?[0-9]*$'. Essa va letta nel seguente modo: la richiesta è valida solamente se inizia (carattere $\hat{}$) e finisce (carattere $) per '/' (si tratta quindi della Home Page del sito web); o (carattere |) se inizia e finisce per '/index.php'; o se inizia e finisce per '/public.php'; o se inizia per '/vulnerable.php' ed è opzionalmente (carattere ?) seguita da '?id=' e da 0 a infiniti (carattere) numeri (stringa [0-9]).

Per poter applicare il firewall appena descritto (tramite il tool implementato) al caso studio precedentemente eseguito (senza nessun firewall attivo), è necessario eseguire il primo comando in figura 4.15 dalla cartella `Programmi/rete_container` (per rimuoverlo eseguire il secondo). Se a questo punto si provasse a visitare il sito web dall'indirizzo IP dell'amministratore non si noterebbe alcun cambiamento rispetto al caso senza firewall, ma se ci si collegasse da un altro dispositivo si noterebbe che la pagina '/private.php' non è più accessibile e che il form vulnerabile non è più attaccabile (accetta solamente valori numerici; in tutti gli altri casi ritorna una risposta di errore).

Ovviamente il caso ideale è quello in cui si ha sia il firewall di livello 3 che quello di livello 7; per mettere ciò in pratica è possibile eseguire il primo comando rappresentato in figura 4.16 dalla cartella `Programmi/rete_container` (di nuovo il secondo serve per rimuovere il firewall). In questo modo si potrà notare che la rete di container è protetta da ogni problema/attacco trovato durante l'analisi del caso studio.

```
sudo ./net_manager.sh install source_init.pp 5
sudo ./net_manager.sh uninstall 4
```

Figura 4.15. Comandi per l'installazione e la rimozione del firewall cilium di livello 7 nella rete di container presa in esame.

```
sudo ./net_manager.sh install source_init.pp 6
sudo ./net_manager.sh uninstall 4
```

Figura 4.16. Comandi per l'installazione e la rimozione del firewall cilium di livello 3 e 7 nella rete di container presa in esame.

4.6.1 Ulteriori considerazioni

Quando si ha parlato di Cilium, si ha descritto come esso utilizzi il nome di un container per l'indirizzamento e la sua etichetta per le policy di networking. Inoltre si ha spiegato che nel momento in cui un nuovo container viene aggiunto alla rete controllata da Cilium, esso eredita automaticamente le policy precedentemente create e che tutto ciò è molto utile nel caso di container dinamici. Prendendo in considerazione il caso studio si può pensare che per un qualche motivo i siti web siano contenuti in container dinamici, che debbano quindi essere distrutti e ricreati. Grazie a Cilium questo può essere fatto senza problemi perché il load balancer saprà comunque contattare i server nonostante abbiano cambiato indirizzo IP grazie all'indirizzamento per nome e le policy in ingresso e in uscita rimarrebbero valide perché associate all'etichetta (che non è cambiata). Un'altra situazione potrebbe essere il sovraccarico dei tre server web che porterebbe a dover installare nuovi container contenenti il sito web; di nuovo l'indirizzamento e le policy funzionano automaticamente (se i nuovi server vengono eseguiti con la stessa etichetta di quelli originali). L'unica cosa da fare in questo caso è aggiornare il load balancer in modo che esso non trasmetta le richieste ricevute unicamente ai tre server originali, ma anche a quelli nuovi.

Supponendo di avere eseguito il caso studio e averlo protetto con il firewall Cilium (in uno dei tre modi descritti in precedenza), è possibile testare ciò che è appena stato descritto con i comandi presenti in figura 4.17: con il primo viene riavviato un 'Web_Server' dopo l'altro, infatti aggiornando continuamente la Home Page del sito web (durante l'esecuzione del comando) si può notare come in principio la pagina venga servita da tre indirizzi IP, per un certo tempo ce ne siano solamente due funzionanti e poi tornino a essere tre, ma uno con indirizzo IP diverso; con il secondo comando invece vengono creati tre nuovi 'Web_Server', aggiornando la Home Page infatti si nota che sono sei gli indirizzi IP che servono la pagina; infine con l'ultimo comando si distruggono i tre 'Web_Server' creati con il secondo comando. Tutto ciò senza dover modificare le policy di networking.

Come considerazione finale bisogna specificare che il programma implementato per la gestione del caso studio, del suo firewall e dei test appena descritti, non è stato scritto unicamente per questo. Esso è infatti in grado di gestire reti generiche di container: può eseguire delle reti insieme a un firewall che le protegga (generato a partire da regole definite dall'utente); e può installare un firewall o eseguire comandi generici su reti di container già esistenti. Inoltre per poter paragonare Cilium a altre soluzioni di filtraggio di livello 3-4 e/o 7 il programma permette di utilizzare anche firewall diversi da Cilium. Comunque maggiori informazioni al riguardo verranno date nelle guide dell'utente (capitolo 8) e del programmatore (capitolo 9).

```
sudo ./net_manager.sh exec source_exec1.pp
sudo ./net_manager.sh exec source_exec2_create.pp
sudo ./net_manager.sh exec source_exec2_destroy.pp
```

Figura 4.17. Comandi per testare la dinamicità dei container nella rete di container presa in esame protetta dal firewall cilium.

Capitolo 5

Controllare il comportamento di programmi con eBPF

5.1 Introduzione

Come già affrontato nel capitolo introduttivo, esistono varie forme di attacco che possono essere attivate in diverso modo. Alcuni attacchi non derivano per forza da nuovi file introdotti in un sistema, ma da processi già in esecuzione e spesso da remoto (nel caso di processi server o client). Nel proteggersi da questo tipo di malware, l'ideale sarebbe monitorare l'intero sistema e cercare di capire se qualcosa di ciò che sta accadendo è illecito. Nel capitolo 2 però si è visto come non sia così semplice definire delle regole per il monitoraggio (o peggio ancora, per il filtraggio) di un intero sistema, rischiando di dare troppa importanza alla sicurezza diminuendo di molto le prestazioni del sistema o di dare poca importanza alla sicurezza per avere prestazioni ideali. Perciò qui si vuole affrontare il problema da un punto di vista diverso: siccome controllare l'intero sistema potrebbe essere problematico, si ha deciso di spostare l'attenzione ai processi in esecuzione; si analizzerà quindi un file eseguibile volutamente vulnerabile mandato in esecuzione nel vero sistema (non in una sandbox o in una macchina virtuale). Lo scopo sarà quello di tenere sotto controllo le operazioni fatte dal processo risultante nel tentativo di accorgersi se viene infettato e reagire di conseguenza tramite eBPF.

5.2 Caso studio

In questo capitolo si prende in esame un programma il cui scopo è leggere delle informazioni salvate in un vettore con anche la possibilità di aggiornarle. Dalla figura 5.1 si può notare che il programma definisce due vettori, inizializzandone solo uno, e poi entra in un ciclo infinito nel quale viene richiesto immediatamente un comando da eseguire: nel caso di 'read' verrà stampato a video il contenuto di 'buffer', nel caso di 'write' verranno modificati i valori contenuti in 'buffer', nel caso di 'exit' si uscirà dal ciclo, e quindi il programma terminerà, e nel caso di comando errato verrà stampato un messaggio di errore. Per poter lanciare il programma in questione, eseguire il comando rappresentato in figura 5.2 dalla cartella `Programmi/programma_target`, il cui scopo è compilare il file C sorgente, eseguirlo e poi cancellare l'eseguibile generato.

5.3 Analisi del caso studio

Prima di analizzare il programma preso in esame, bisogna sottolineare che l'analisi descritta di seguito è stata fatta su un sistema Arch Linux a 64 bit. La stessa analisi potrebbe essere fatta su un sistema diverso, ma potrebbe essere necessario modificare alcuni parametri dell'exploit creato (il programma che sfrutta le vulnerabilità trovate per mettere in pratica in modo automatico le

```
#include <stdio.h>
#include <string.h>

#define BUF_LEN 8
#define CMD_LEN 128

/* MAIN */
int main () {
    char command [CMD_LEN];
    char buffer [BUF_LEN] = {'1', '2', '3', '4', '5', '6', '7', '8'};
    int el_num, i;

    /* Loop sul Comando da Eseguire */
    while (1) {
        printf ("Inserisci il Comando da Eseguire (read | write | exit): ");
        fflush (stdout);
        scanf ("%s", command);

        if (strcmp (command, "read") == 0) { // Leggi Dati
            ...

        } else if (strcmp (command, "write") == 0) { // Scrivi Dati
            ...

        } else if (strcmp (command, "exit") == 0) // Esci
            break;

        else { // Errore
            ...
        }
    }

    return 1;
}
```

Figura 5.1. Scheletro del programma `Programmi/programma_target/vulnerable_program.c` allegato alla tesi.

```
bash program_manager.sh 1
```

Figura 5.2. Esecuzione basilare dello script `Programmi/programma_target/program_manager.sh` allegato alla tesi.

operazioni spiegate di seguito). Per poter utilizzare ciò che è stato implementato in un sistema diverso da Arch Linux senza dover modificare niente, è possibile creare un ambiente Arch all'interno di un container grazie al Dockerfile presente nella cartella `Programmi/programma_target`. Seguendo le istruzioni nell'apposita sezione della guida dell'utente (capitolo 8) è possibile ottenere una shell Arch (interna al container creato) nella quale si potranno eseguire i comandi così come sono rappresentati nel resto del capitolo (e nella guida).

Passando ora alla vera e propria analisi, dopo aver eseguito il programma target, si passa a analizzarlo dall'esterno dal punto di vista di un possibile attaccante. Innanzitutto si testano

tutte le funzionalità per capire come funziona il programma: si prova a inserire il comando 'read', viene chiesto il numero di valori da leggere e poi vengono stampati i valori richiesti a partire dal primo; con il comando 'write' si ha di nuovo la richiesta di quanti valori si vuole scrivere, segue la richiesta di un valore per volta e l'inserimento all'interno del vettore dalla prima posizione (cosa che si scopre chiedendo una 'read' dopo aver finito di scrivere i valori); con il comando 'exit' il programma termina; e con un comando errato il comando viene stampato a video insieme a un messaggio di errore. Inoltre viene subito testato il comportamento del programma nel caso in cui si inserisse un numero di valori da leggere/scrivere maggiori di quelli esistenti (che si scopre essere 8), ma non si riscontra alcun problema visto che il programma stampa un messaggio di errore e torna a aspettare un nuovo comando. In figura 5.3 è presente il codice che mostra che l'analisi è corretta.

A questo punto si passa a analizzare il comando letto. Inserendo un comando molto lungo viene stampato a video il solito errore di comando errato, ma non sembra accadere nient'altro finché non si esegue un comando di 'exit' che fa terminare il programma, ma con un errore di '*** stack smashing detected ***' anziché con una chiusura normale. Questo fa capire quattro cose all'attaccante: prima di tutto che il vettore nel quale viene salvato il comando è vulnerabile a Buffer Overflow (sullo stack); poi che l'overflow non ha alcun effetto sulle variabili locali alla funzione che gestisce i comandi (infatti l'errore viene riscontrato unicamente in fase di uscita dal ciclo) e quindi sulla funzione stessa (il main in questo caso); che la funzione in cui si trova il ciclo sulla lettura del comando ritorna solo nel momento in cui si inserisce un comando di 'exit' (di nuovo, questo perché non si ha errore fino all'uscita dal ciclo); e infine che il programma target è stato compilato con la protezione dello stack attiva. Questa protezione inserisce a livello di codice nel programma da proteggere la funzionalità degli Stack Canaries: subito dopo aver chiamato una funzione a rischio di Buffer Overflow, il programma spinge sullo stack un valore intero casuale; subito prima dell'istruzione di ritorno di quella stessa funzione questo valore viene controllato e il programma prosegue la sua esecuzione solamente se il valore è valido, altrimenti viene ucciso con il segnale 'SIGABRT' (da qui il messaggio descritto in precedenza). Tutto questo porta l'attaccante a capire che nell'exploit (l'attacco che vuole mettere in pratica) per poter sfruttare l'overflow deve per forza uscire dal ciclo tramite il comando 'exit' (per spingere il programma a interpretare lo stack infettato), magari direttamente nella stringa che causa l'overflow, facendo però attenzione a non modificare lo Stack Canary (o il programma verrà ucciso e lui non avrà ottenuto nulla).

Lasciando un attimo da parte lo Stack Canary (verrà analizzato più avanti), un attaccante potrebbe decidere di provare a infettare lo stack con dello Schellcode (codice, preferibilmente scritto in Assembler nel modo più compatto possibile, generalmente utilizzato per infettare file/processi) e poi sovrascrivere l'indirizzo di ritorno con l'indirizzo in cui inizia lo Shellcode (più avanti verrà spiegato come sapere quando finisce il vettore, quando inizia lo Stack Canary, dove si trova l'indirizzo di ritorno, ...) così che, dopo aver inserito il comando 'exit', il programma proseguirà fino all'istruzione di 'ret' (al fondo del main) e poi salterà alle istruzioni del malware anziché continuare con quelle reali (terminare il programma). Il problema è che la maggior parte dei programmi al giorno d'oggi sono compilati con lo stack non eseguibile (come in questo caso). Il programma perciò terminerebbe con un errore di 'Segmentation Fault' e segnale 'SIGSEGV'. Per evitare questo un attaccante potrebbe provare a infettare lo stack in modo da chiamare la system call 'mprotect', dare il permesso di esecuzione allo stack e poi eseguire lo shellcode. Ma in alcuni casi, come in questo, ciò risulterebbe troppo complicato. Per questo motivo l'attaccante deciderebbe ora di seguire un'altra strada: eseguire un attacco di tipo Return-to-Libc. Lo scopo non è più inserire del codice e farlo eseguire dal programma, ma modificare lo stack in modo tale da spingere il programma a eseguire del codice che è stato caricato in memoria in modo lecito e in sezioni già eseguibili, ma in modo tale da ottenere ciò che si vuole. Più precisamente si è scelto di spingere il programma a eseguire la funzione 'system' della libreria Libc, che permette di eseguire un comando di sistema prendendo in input l'intero comando sotto forma di un'unica stringa.

Avendo ora chiaro il tipo di attacco da mettere in pratica, si cerca di ottenere le informazioni mancanti: la posizione e il valore dello Stack Canary; la posizione dell'indirizzo di ritorno, per poterlo sovrascrivere e poter così pilotare il registro RIP (e quindi l'esecuzione del programma target); l'indirizzo della 'system'; e un modo per poter passare una stringa alla 'system', perché su sistemi a 32 bit basterebbe inserire il puntatore alla stringa nel punto giusto dello stack (i parametri in ingresso alle funzioni vengono passati sullo stack), ma su sistemi a 64 bit, come

```
...
        if (strcmp (command, "read") == 0) { // Leggi Dati
            printf ("Inserisci il Numero di Elementi da Leggere: ");
            fflush (stdout);
            scanf ("%d", &el_num);
            if (el_num < 0 || el_num > BUF_LEN) {
                printf ("Il Numero di Elementi deve Essere Compreso tra 0 e
                    8!\n");
                fflush (stdout);
                continue;
            }
            for (i = 0; i < el_num; i++)
                printf ("0x%x ", buffer [i]);
            printf ("\n");
            fflush (stdout);

        } else if (strcmp (command, "write") == 0) { // Scrivi Dati
            printf ("Inserisci il Numero di Elementi da Scrivere: ");
            fflush (stdout);
            scanf ("%d", &el_num);
            \getchar ();
            if (el_num < 0 || el_num > BUF_LEN) {
                printf ("Il Numero di Elementi deve Essere Compreso tra 0 e
                    8!\n");
                fflush (stdout);
                continue;
            }
            for (i = 0; i < el_num; i++) {
                printf ("Inserisci l'Elemento n° %d: ", i + 1);
                fflush (stdout);
                scanf ("%c", &buffer [i]);
                getchar ();
            }
            printf ("\n");
            fflush (stdout);

        } else if (strcmp (command, "exit") == 0) // Esci
            break;

        else { // Errore
            printf ("Il Comando Inserito è Errato!\n");
            printf (command);
            printf ("\n");
            fflush (stdout);
        }
...
```

Figura 5.3. Parte del programma `Programmi/programma_target/vulnerable_program.c` allegato alla tesi.

in questo caso, i primi parametri vengono passati alle funzioni tramite registri e quindi bisogna trovare un modo per inserire il puntatore alla stringa nel registro 'rdi'. Per mettere in pratica quest'ultima parte, bisogna costruire quella che viene definita 'ROP Chain': utilizzando lo stesso meccanismo per la chiamata alla 'system' (sovrascrivere l'indirizzo di ritorno così da cambiare

l'esecuzione del programma), è possibile eseguire un pezzo di codice che finisce con l'istruzione 'ret' in modo da ottenere l'esecuzione di istruzioni utili al malware e poi ritornare a un altro pezzo di codice utile che finisce di nuovo per 'ret' per eseguire altre istruzioni utili... e andare avanti così finché non si ha ottenuto ciò che si vuole (questo verrà approfondito a breve). ROP sta per 'Return Oriented Programming' e ogni pezzo di codice tra un'istruzione di 'ret' e l'altra viene definito un 'ROP gadget'. In questo caso serve unicamente un gadget, quello che permette di inserire il puntatore della stringa da passare alla 'system' nel registro 'rdi': grazie al tool Ropper, che permette di analizzare un eseguibile e di stampare a video tutti i possibili 'ROP gadget', si scopre che ne esiste uno contenente due sole istruzioni, 'pod rdi' e 'ret', perfetto per l'attacco da mettere in pratica. Risulta quindi che le operazioni da fare siano: modificare l'indirizzo di ritorno per chiamare le istruzioni trovate (così da inserire il puntatore alla stringa in 'rdi') e poi sfruttare la nuova istruzione di ritorno (la seconda del gadget) per saltare alla funzione 'system'.

A questo punto mancano ancora i valori necessari alla costruzione dell'exploit (lo Stack Canary, l'indirizzo del gadget e l'indirizzo della 'system'). Per ottenerli un attaccante dovrebbe trovare una vulnerabilità che gli permetta di leggere qualche locazione di memoria utile all'interno del processo target. Nel caso studio questo è proprio ciò che si ottiene se il comando inserito è errato: il comando viene stampato a video, quindi un attaccante può provare a inserire degli identificatori come comando ('%d', '%x', ...) e vedere se la funzione li elabora come veri e propri identificatori. Se sì (come in questo caso), il comando errato che verrà stampato a video non sarà la stringa inserita, ma sarà l'elaborazione della stringa inserita. Perciò, inserendo una stringa come '%p-%p-%p-%p', verranno stampati a video quattro valori esadecimali su 64 bit separati da un trattino, letti dallo stack della chiamata alla 'printf' che stampa a video la stringa di errore. Ma la 'printf' viene chiamata dalla funzione vulnerabile (il main) e lo stack di una funzione segue quello della funzione che l'ha chiamata (o meglio, precede, siccome lo stack cresce da un indirizzo maggiore a uno minore); quindi, inserendo abbastanza '%p', si potrà leggere l'intero stack della 'printf' e l'intero stack del main. Questo attacco viene chiamato Format String Attack e in questo caso permette di ottenere tutti i valori utili alla scrittura dell'exploit.

In figura 5.4 si può notare che inserendo come comando quello in alto, il programma risponde (dopo aver stampato il messaggio di errore) con la stringa rappresentata subito sotto (una serie di valori esadecimali su 8 byte; '(nil)' indica un valore pari a 0). Segue la rappresentazione dello stack visto dal main del programma. Se si paragona quest'ultimo con la risposta ottenuta, si può notare che dopo i primi 5 valori (appartenenti allo stack della 'printf'), lo stack del main è esattamente uguale alla successione dei valori ricevuti in risposta. A questo punto si passa a analizzare lo stack per capire se si hanno tutti i parametri necessari alla creazione della stringa con cui fare overflow e di conseguenza avere un exploit funzionante. In ordine, bisogna innanzitutto fare in modo che il programma esca dal ciclo infinito inserendo in testa al comando da inviare la stringa 'exit' seguita dal byte '0x00'; poi bisogna inserire valori casuali (per esempio '0xAB') fino allo Stack Canary; bisogna fare in modo che quest'ultimo non diventi invalido, ma siccome ora si conosce lo Stack Canary associato a questa esecuzione (perché si trova nella stringa ritornata dal messaggio di errore), basta concatenare al comando quel valore, che andrà a sovrascrivere l'originale con una copia identica e quindi a non invalidarlo; bisogna sovrascrivere il valore a cui punta 'rbp' (il valore del base pointer della funzione precedente) con valori casuali; e poi bisogna sovrascrivere l'indirizzo di ritorno con l'indirizzo del 'ROP gadget' così che l'esecuzione del programma si sposti da quella naturale a quella dell'exploit. Il problema a questo punto è che grazie all'indirizzo contenuto in 'rbp' e all'indirizzo di ritorno, si nota subito che l'eseguibile è un PIE (Position Independent Executable): ogni volta che viene eseguito le sezioni del file verranno caricate in memoria virtuale a indirizzi random. Questo implica che non è possibile utilizzare un indirizzo statico per spingere l'esecuzione verso il 'ROP gadget'. Ma siccome il gadget si trova sempre allo stesso offset all'interno della sezione text dell'eseguibile (la quale si trova a un indirizzo sempre diverso), basta trovare un indirizzo interno alla sezione text e, conoscendone l'offset rispetto al gadget, calcolare l'indirizzo di quest'ultimo. In una funzione qualunque, un indirizzo utile sarebbe quello di ritorno; siccome qui ci si trova nel main, l'indirizzo da utilizzare è quello puntato dal base pointer. Ottenuto l'indirizzo del gadget, lo si concatena al comando. Ora si ha bisogno del puntatore alla stringa da passare alla 'system' (perché il programma a questo punto eseguirà l'istruzione 'pop rdi', la prima del gadget). La stringa verrà inserita sullo stack, quindi bisognerà passare un indirizzo appartenente allo stack, ma di nuovo lo stack si trova in un posto diverso a ogni esecuzione e quindi non si

può utilizzare un valore statico. Per l'ennesima volta si analizza la risposta ricevuta al comando inviato prima e si nota che l'ultimo valore ricevuto è appartenente allo stack (anche se un po' distante); lo si prende, gli si sottrae un offset (solamente per avvicinare la stringa alla posizione corrente nello stack e non dover inviare troppi caratteri nell'exploit) e lo si concatena al comando. A questo punto verrà eseguita la seconda istruzione del gadget ('ret'). Siccome l'intenzione è quella di chiamare la funzione 'system', qui ci vorrà l'indirizzo di questa funzione. Se l'ASLR (Address Space Layout Randomization) non fosse attivo, le funzioni di libreria si troverebbero a indirizzi statici, ma di nuovo non è questo il caso e di nuovo bisogna trovare una soluzione utilizzando lo stesso ragionamento di prima: anche se Libc si trova a ogni esecuzione a un indirizzo diverso, le funzioni al suo interno si troveranno tutte allo stesso offset rispetto alla libreria stessa; trovando quindi un indirizzo interno a Libc sarebbe possibile calcolare l'indirizzo della 'system' semplicemente conoscendo l'offset tra la funzione e l'indirizzo trovato (stesse versioni di Libc, compilate per lo stesso sistema, avranno stessi offset). Analizzando per bene la risposta ricevuta con il primo comando, si può scoprire che il secondo valore (il primo in rosso) è esattamente questo. Avendo ora anche l'indirizzo della 'system' lo si concatena al comando; si finisce concatenando la stringa da passare alla 'system' (che finirà esattamente all'indirizzo inserito in precedenza) e poi si invia il comando al programma.

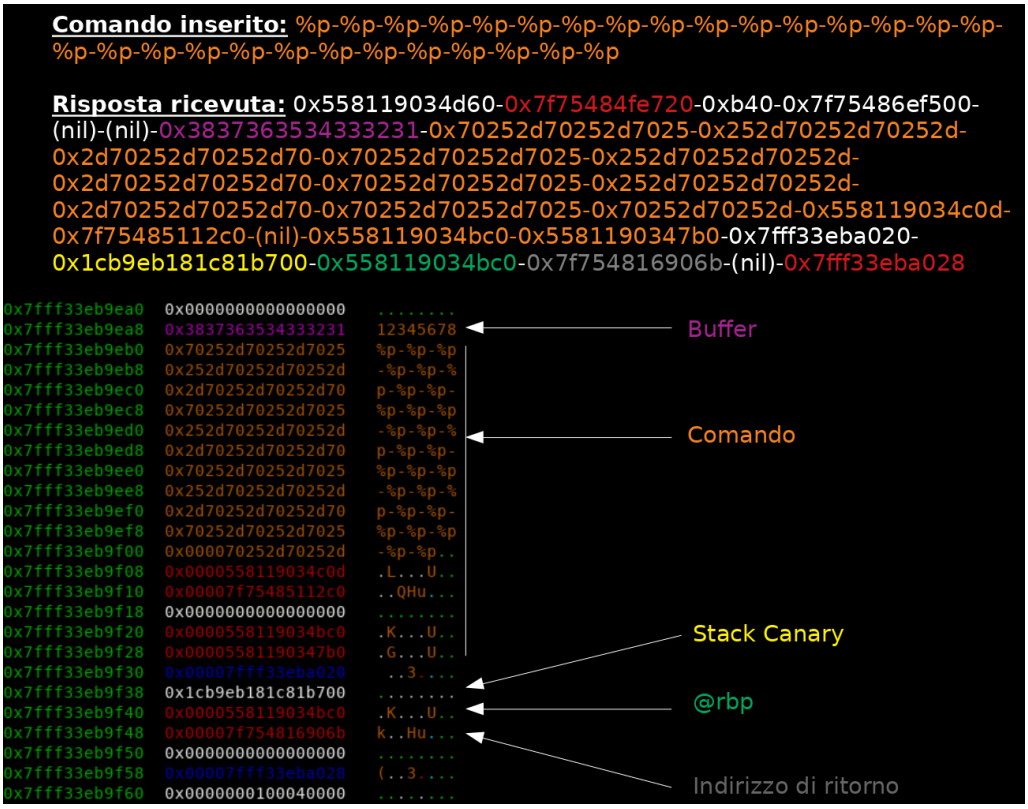

Figura 5.4. Rappresentazione dello stack del programma target prima dell'attivazione dell'exploit.

Tutto ciò che è appena stato descritto è rappresentato in figura 5.5: dopo aver creato il comando (quello in alto) e averlo inviato al programma, lo stack del main contiene la stringa 'exit' seguita dal byte '0x00', una serie di byte casuali, lo stesso Stack Canary di prima, altri caratteri casuali come valore di 'rbp', l'indirizzo di ritorno al 'ROP gadget' selezionato, il puntatore alla stringa che rappresenta il comando di sistema da eseguire, l'indirizzo della funzione 'system' e infine il comando di sistema ('ls' nell'esempio) seguito dal byte '0x00' (per essere sicuri che la stringa sia chiusa nel modo corretto). Per poter mettere in pratica ciò è stato scritto lo script Python `Programmi/programma_target/exploit.py` che esegue al suo interno il processo target e crea una pipe per la comunicazione con esso in modo da poter scrivere sul suo stdin e leggere dal suo stdout. Lo stesso script utilizzato prima per l'esecuzione del programma vulnerabile può essere

usato per attivare l'exploit: con il comando rappresentato in figura 5.6 (eseguito dalla cartella `Programmi/programma_target`) è possibile quindi ottenere il risultato del comando passato come secondo parametro ('ls') eseguendolo all'interno del programma vulnerabile come appena descritto.

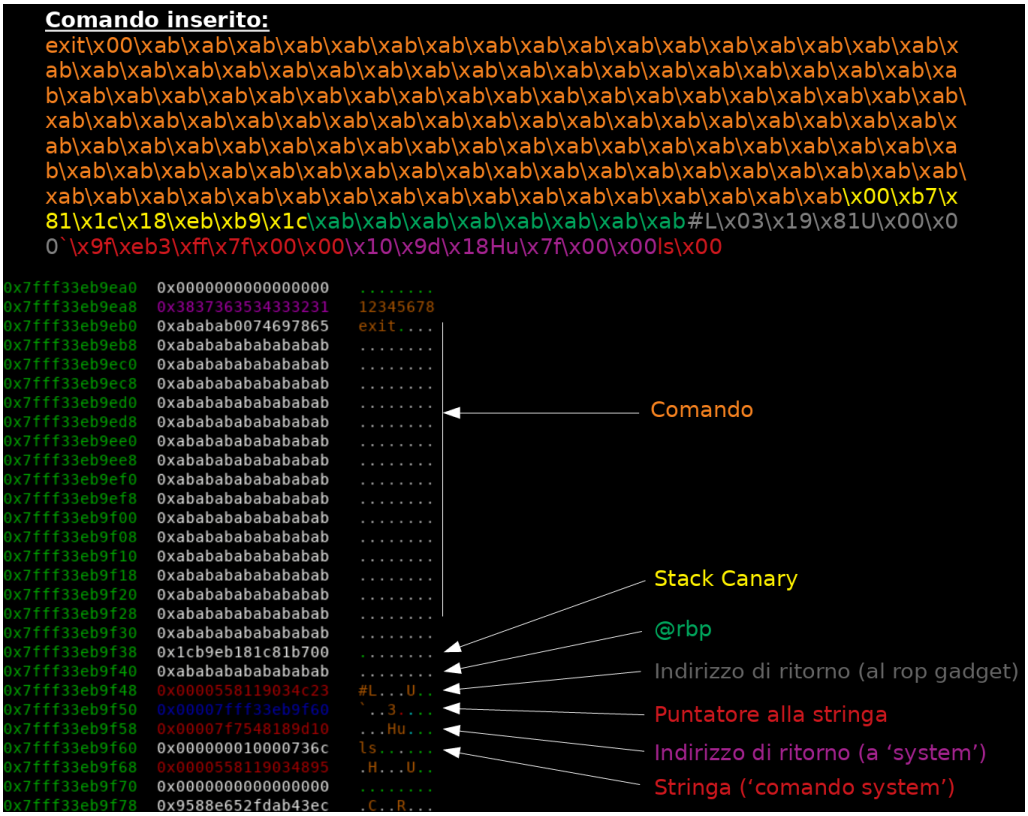

Figura 5.5. Rappresentazione dello stack del programma target dopo l'attivazione dell'exploit.

5.3.1 Ulteriori considerazioni

Come si è visto, nonostante la semplicità del caso studio, esso risulta avere una vulnerabilità che permette l'esecuzione di codice arbitrario. Ovviamente vista la semplicità dell'esempio, questo non sembra essere particolarmente grave, ma se questo programma fosse eseguito dall'utente root e permettesse a tutti gli utenti di chiedergli e inviargli certe informazioni (magari filtrate in base ai permessi dei vari utenti), questa vulnerabilità porterebbe qualunque utente a poter eseguire comandi con permessi di root portando al cosiddetto Privilege Escalation (il passaggio da un utente con permessi minori a un utente con permessi maggiori). O se questo fosse un servizio installato su un server accessibile a qualunque client, un attaccante potrebbe sfruttare la vulnerabilità per entrare nel server stesso. Ovviamente il problema maggiore è l'analisi della vulnerabilità: se non si avesse una copia dell'eseguibile, non sarebbe semplice calcolare tutti gli offset, ma si potrebbero comunque fare diversi test (tramite meccanismi di fuzzing per esempio) e ottenere in ogni caso le informazioni necessarie alla costruzione dell'exploit.

```
bash program_manager 1 ls
```

Figura 5.6. Esecuzione dello script `Programmi/programma_target/program_manager.sh` allegato alla tesi con attivazione dell'exploit.

```
bash program_manager 1 "ls -l"
```

Figura 5.7. Esecuzione errata dello script `Programmi/programma_target/program_manager.sh` allegato alla tesi con attivazione dell'exploit e comando con spazio.

Un'ulteriore considerazione da fare è il fatto che l'exploit viene utilizzato per chiamare la funzione 'system' in un programma che di suo non la chiama. Questo è possibile perché quando un programma viene eseguito, il sistema operativo carica in memoria le librerie che contengono le funzioni che servono al suo funzionamento e poi collega il programma con le funzioni stesse (inserendo gli indirizzi di queste funzioni in un'apposita tabella interna al processo). Di nuovo, il loader carica le librerie, non le funzioni: questo significa che anche se in teoria un processo può chiamare solamente le funzioni di cui conosce gli indirizzi in realtà ha la possibilità di accedere a tutte le istruzioni della libreria che contiene quelle funzioni (e quindi anche a tutte le altre funzioni).

Infine, nell'eseguire l'exploit si dovrebbe notare che non sempre funziona. Infatti ogni tanto capita che il programma vulnerabile venga ucciso per 'Segmentation Fault' (questo accade quando un processo cerca di accedere a una locazione di memoria invalida) o per Stack Canary non corretto. Per spiegarne il motivo, conviene prima analizzare cosa succede quando si chiede all'exploit di eseguire nel processo target un comando con spazio. Se si esegue il comando rappresentato in figura 5.7 dalla cartella `Programmi/programma_target` si può notare che il risultato ottenuto non è quello del comando 'ls -l', ma quello del comando 'ls', come se la parte dopo lo spazio venisse ignorata. Questo è esattamente quello che accade: il programma target riceve la stringa inviata dall'exploit all'interno della funzione 'scanf'; questa si aspetta una stringa in input; per la funzione 'scanf' una stringa è definita come l'insieme di caratteri che precedono uno spazio o un a capo; quindi nel momento in cui incontra lo spazio del comando, smette di leggere, cosa che porta all'inserimento sullo stack, e quindi all'esecuzione, unicamente della prima parte del comando. Questo è anche ciò che succede quando il programma target ritorna errore: se in un qualunque punto della stringa che l'exploit invia al programma vulnerabile c'è un valore che in ASCII rappresenta uno spazio o un a capo, la 'scanf' smetterà di leggere e lo stack non sarà infettato nel modo desiderato. Ovviamente non esiste una soluzione a quest'ultima situazione (se lo Stack Canary contiene uno dei due caratteri 'invalidi' se anche li si sostituisse per far sì che la 'scanf' leggesse l'intera stringa l'esecuzione verrebbe comunque bloccata dal controllo sul canary); si può invece aggirare il problema dell'esecuzione di un comando con spazio grazie alla variabile d'ambiente IFS (Internal Field Separator). Essa è una variabile che descrive quali sono i caratteri che nella shell corrente vengono usati per separare una parola dall'altra. Di default essa contiene uno spazio (infatti se si esegue un comando con spazio in una shell, viene presa la prima stringa e usata come nome del programma da eseguire e la seconda come parametro da passare al programma). Nel caso preso in considerazione quindi un comando con spazio può essere eseguito come rappresentato in figura 5.8 (sempre dalla cartella `Programmi/programma_target`): in questo modo la shell che esegue l'exploit passerà allo script come secondo parametro la stringa 'ls${IFS}-l' (perché il '\' dice alla shell dell'attaccante di interpretare ciò che segue come una stringa e non come una variabile e quindi essa non viene sostituita con uno spazio) il quale lo invierà al programma target che lo passerà alla funzione 'system' che chiederà a una shell di eseguirlo, la quale sostituirà il '${IFS}' con uno spazio ottenendo così il comando 'ls -l'. Ovviamente se nella shell target la variabile IFS fosse stata modificata per contenere un separatore diverso dallo spazio, il tutto funzionerebbe comunque: se IFS contenesse il carattere ',' il comando eseguito dalla shell target verrebbe trasformato in 'ls,-l', la ',' verrebbe letta come separatore e quindi verrebbe eseguito il comando 'ls' con un parametro pari a '-l'.

5.4 Controllare le chiamate a funzione con eBPF

Come visto in precedenza, con eBPF è possibile monitorare le chiamate a funzione e i parametri passati a esse con i tipi di programma KPROBE e TRACEPOINT. Siccome l'obiettivo qui è

```
bash program_manager 1 "ls\${IFS}-l"
```

Figura 5.8. Esecuzione corretta dello script `Programmi/programma_target/program_manager.sh` allegato alla tesi con attivazione dell'exploit e comando con spazio.

controllare quali funzioni vengono chiamate e come vengono chiamate da un certo eseguibile, si potrebbe pensare di attaccarsi a tutte le funzioni associate a tale processo in ingresso e in uscita e analizzarle in tempo reale. Il problema però è che, come visto in precedenza, un attaccante non è solo in grado di chiamare una funzione presente nell'eseguibile con parametri illeciti, ma è anche in grado di chiamare funzioni non associate al processo infettato. Questo implica che nel controllare solo le funzioni presenti, non ci si accorgerebbe delle possibili chiamate a funzione diverse, ma solo di parametri non permessi passati a funzioni permesse. Risulta quindi necessario controllare un insieme maggiore di funzioni. Nel caso delle system call è semplice: esse vengono tutte gestite dall'istruzione 'syscall' che prende come primo parametro l'identificativo della system call da chiamare e come successivi parametri i parametri da passare alla funzione; ciò significa che tutte le system call passano per lo stesso punto (sia in ingresso al che in uscita dal kernel); a differenziarle è semplicemente l'identificativo che dice al kernel quale funzione deve effettivamente chiamare. Nei punti di ingresso e uscita delle varie system call è installato un tracepoint al quale ci si potrebbe attaccare e analizzare così ogni system call effettuata all'interno del sistema. In questo modo sarebbe possibile controllare tutte le system call chiamate da tutti i processi del sistema. Siccome qui si punta a controllare solo le chiamate effettuate da un certo processo, come prima operazione del programma eBPF installato si potrebbe controllare il PID del processo che ha fatto richiesta della system call corrente e decidere se continuare il controllo o meno in base a questo. Ma il caso delle funzioni di libreria invece, risulta più complesso: siccome non hanno un punto di ingresso o di uscita in comune e siccome un processo infettato potrebbe chiamare qualunque funzione di qualunque libreria, in teoria bisognerebbe controllarle tutte. L'ideale a questo punto sarebbe attaccarsi a tutte le funzioni delle librerie caricate dal processo da controllare e fare attenzione al fatto che non vengano caricate altre librerie (a meno che il programma non lo debba effettivamente fare). Attaccarsi però a tutte le funzioni di libreria significa dover creare un programma eBPF per ognuna di esse, operazione che risulterebbe ovviamente troppo lunga; senza contare il fatto che in memoria si avrebbero installate più funzioni di controllo che sezioni di file eseguibili e funzioni di libreria. A questo punto si potrebbe pensare di controllare solamente un sottoinsieme delle funzioni di libreria, tenendo però presente le considerazioni fatte nel capitolo 3 sulla creazione di regole (per evitare che siano troppo pesanti per il sistema o troppo semplici da aggirare). Ma anche questo potrebbe non essere la soluzione ideale. A questo punto bisogna chiedersi se controllare le funzioni di libreria sia davvero necessario: la maggior parte delle funzioni di libreria chiamano al loro interno delle system call e le uniche funzioni che non interagiscono con il kernel sono quelle che eseguono calcoli matematici o istruzioni basilari tra registri e stack; tutto ciò che un attaccante è interessato a fare (a parte in casi molto eccezionali), deve per forza passare per il livello kernel (lettura/modifica/scrittura di file, utilizzo della rete internet, esecuzione di nuovi processi, modifica/infezione di processi in esecuzione, ...). Di conseguenza, anche se un attaccante chiamasse una funzione di libreria illecita, essa chiamerà almeno una system call al suo interno e se si tenessero sotto controllo solamente le system call, il programma eBPF si attiverebbe in ogni caso. Ne consegue che per controllare il comportamento di un processo conosciuto, l'ideale è semplicemente controllare le system call che chiama e i parametri che esse ricevono.

L'ultima cosa da decidere è quale operazione effettuare nel momento in cui il controllo scoprisse qualcosa di inaspettato. Come spiegato in precedenza con eBPF, nel caso di controllo di funzioni, si può solamente effettuare un'operazione di monitoraggio: se succede qualcosa, si avverte il livello utente e esso reagirà in qualche modo. La reazione generalmente è la stampa di un messaggio, ma potrebbe anche essere un'azione di filtraggio: si potrebbe per esempio spingere il programma utente a uccidere il processo che ha effettuato una chiamata illecita. Questo però porterebbe a avere una reazione a livello utente, non kernel: il programma eBPF invia un messaggio al programma utente, il quale lo elabora, capisce che deve uccidere un certo processo, allora chiede al kernel di terminarlo, esso esegue l'azione richiesta e poi manda il risultato al livello utente. Se si volessero

evitare tutti questi scambi tra i due livelli (e avere maggiore permessi), si potrebbe scrivere un modulo del kernel che registri il programma eBPF, riceva il risultato della sua esecuzione e reagisca di conseguenza direttamente da dentro al kernel. Ma questo è esattamente quello che fa Seccomp.

5.5 Seccomp per il filtraggio di system call

Come anticipato nel capitolo 2 Seccomp [12] è un'utilità incorporata nel kernel Linux (fin dalla versione 2.6.12) che permette di decidere quali system call un processo (e tutti i suoi figli) può o non può chiamare. Inizialmente un utente poteva decidere di abilitare questo meccanismo di filtraggio per un processo qualunque andando a settare a 1 il flag contenuto in `/proc/[PID]/seccomp` e da quel momento in avanti quel processo aveva solamente il permesso di chiamare le system call 'read', 'write', 'exit' e 'sigreturn'. Successivamente il file è stato rimosso e al suo posto è stata creata una system call che permetteva di abilitare Seccomp per il processo corrente (e per i suoi figli). Dalla versione 3.5 del kernel, è stata aggiunta una seconda modalità che permetteva a un processo di definire quali system call potevano essere chiamate (o non chiamate) da lui e dai suoi figli e, opzionalmente, con quali parametri in ingresso, tramite un programma BPF. Nello specifico, quando un programma di livello utente voleva generare un filtro Seccomp, doveva innanzitutto inizializzare la libreria corrispondente e poi doveva definire il filtro da passare al kernel scrivendolo direttamente nel linguaggio BPF o usando le funzioni di appoggio definite dalla libreria, le quali permettono di creare un contesto Seccomp contenente un'azione standard da attivare su tutte le system call che non sono presenti nel filtro (per esempio bloccarne o permetterne l'esecuzione) e di definire tutte le regole di filtraggio tramite parole chiave (per esempio una whitelist o blacklist di funzioni che è possibile o non è possibile chiamare in generale o con determinati parametri in ingresso). In questo secondo caso, era la libreria stessa a generare il programma BPF (dato il contesto). Successivamente il programma BPF veniva passato al kernel e il programma utente poteva proseguire con le vere e proprie istruzioni. Durante la sua esecuzione, ogni volta che quel dato processo (o uno dei suoi figli) chiamava una certa system call, il kernel passava il controllo al programma BPF passandogli in ingresso il numero della system call e i parametri di input; il programma analizzava i dati e ritornava un valore diverso a seconda dell'azione definita nel contesto (in fase di programmazione):

- SECCOMP_RET_KILL_PROCESS: il processo che ha chiamato la system call corrente deve essere ucciso con un segnale di SIGSYS (non catturabile dal processo stesso);
- SECCOMP_RET_KILL_THREAD: il thread che ha chiamato la system call corrente deve essere terminato;
- SECCOMP_RET_TRAP: il segnale SIGSYS deve essere inviato al processo che ha chiamato la system call corrente; la system call non deve essere eseguita, ma il processo ha la possibilità di catturare il segnale e reagire di conseguenza;
- SECCOMP_RET_ERRNO: la system call non deve essere eseguita e deve essere ritornato errore al processo che ha chiamato la system call corrente;
- SECCOMP_RET_TRACE: prima dell'esecuzione della system call corrente, bisogna passare il controllo a un debugger (definito precedentemente);
- SECCOMP_RET_LOG: la system call corrente deve essere eseguita, ma solo dopo aver salvato le informazioni relative a essa in un file di log;
- SECCOMP_RET_ALLOW: la system call corrente deve essere eseguita normalmente.

Questo valore di ritorno veniva ricevuto dal modulo del kernel associato a seccomp e la relativa azione veniva eseguita direttamente da dentro al kernel.

Con l'avvento di eBPF, Seccomp si è aggiornato: al posto di utilizzare BPF (e lasciare che il kernel lo traduca automaticamente nel suo successore), ora utilizza direttamente eBPF. Questo significa che nel caso di creazione del filtro tramite le funzioni della libreria descritte sopra, a

livello di programmazione non è cambiato niente (le funzioni da chiamare sono le stesse così come le parole chiave), ma nell'altro caso, bisogna passare alla libreria un programma eBPF sotto forma di puntatore a un vettore di strutture sock_filter, ciascuna delle quali contiene un'istruzione del programma (nella forma descritta nel capitolo 3).

In figura 5.9 si trova un esempio (o meglio, una parte di un esempio) che dimostra come sarebbe possibile utilizzare Seccomp per proteggere un certo programma generando il filtro tramite le apposite funzioni (per maggiore semplicità). Come si può notare, la prima operazione da fare è l'inizializzazione del contesto che permetterà alla libreria di generare il programma eBPF: nell'esempio in questione viene inizializzato con l'operazione SCMP_ACT_KILL che indica che l'azione di default da mettere in pratica nel momento in cui verrà chiamata una system call è uccidere il processo che ha effettuato la chiamata. Altre possibili azioni sono:

- SCMP_ACT_ALLOW per permettere l'esecuzione della system call;
- SCMP_ACT_TRAP per inviare un segnale SIGSYS (catturabile) al thread che ha effettuato la chiamata;
- SCMP_ACT_ERRNO(uint16_t errno) per inviare al thread un valore di ritorno pari a 'errno';
- SCMP_ACT_TRACE(uint16_t msg_num) per notificare il processo (se si trova in fase di debug).

Segue la definizione delle eccezioni e quindi di tutte le system call sulle quali non dovrà essere eseguita l'operazione di default (l'uccisione del processo chiamante). Di conseguenza il filtro creato diventa una whitelist. Infine il filtro viene caricato nel kernel e si passa a definire il vero e proprio programma. Più nel dettaglio, ogni regola di filtraggio viene definita tramite la funzione 'seccomp_rule_add (...)' che prende come primo parametro il contesto Seccomp a cui associare la regola, come secondo parametro l'azione da eseguire su quella system call (in questo caso, siccome si sta definendo una whitelist, l'operazione sarà di permesso di esecuzione per tutte le regole), come terzo il nome della system call e come quarto parametro il numero di valori di input (alla system call) su cui è definito il filtro. Le azioni possibili sulla system call che si sta aggiungendo al filtro sono ovviamente quelle elencate in precedenza (quelle rappresentabili l'azione di default); i parametri passati in input alla system call invece possono essere omessi, ma se presenti vanno definiti dalla posizione (SCMP_A*), dall'operatore da usare per paragonare il valore passato alla system call e quello definito nel filtro (SCMP_CMP_EQ sta per '=', SCMP_CMP_NE sta per '!=', SCMP_CMP_LT sta per '<', ...; una lista completa è presente nel manuale della funzione seccomp_rule_add) e dal valore stesso. Per esempio, nel secondo caso rappresentato nella figura 5.9 si sta dicendo che con qualunque parametro verrà chiamata la system call 'exit_group', essa dovrà essere lasciata passare; nel primo caso invece si sta dicendo che la system call 'read' dovrà essere permessa solamente quando riceve in ingresso un primo parametro casuale (perché manca SCMP_A0), un secondo parametro pari a 'buffer' (più precisamente pari all'indirizzo di 'buffer') e un terzo parametro pari a 'BUF_LEN'.

Come si può notare dall'esempio, definire le regole in questo modo è molto più semplice che scrivere il programma eBPF direttamente, ma ovviamente il controllo dei parametri passati alle system call è limitato dalla libreria, la quale è in grado di analizzare solamente parametri diretti e quindi, in caso di puntatori, si possono solamente paragonare i puntatori, non i valori puntati. A volte questo è abbastanza, ma a volte potrebbe non esserlo; in questo secondo caso bisogna per forza scrivere il programma eBPF direttamente così da poter analizzare anche i valori puntati nel modo più dettagliato e personalizzato possibile. In entrambi i casi ovviamente bisogna fare attenzione a come vengono definite le regole di filtraggio: regole troppo restrittive potrebbero creare problemi all'esecuzione del programma stesso; regole troppo permissive potrebbero essere facilmente aggirabili. Inoltre, fin che si tratta di filtrare in base a quali system call possono essere chiamate e quali no, le regole hanno un livello di difficoltà che potrebbe anche essere elevato, ma non quanto il dover filtrare anche in base ai parametri passati alle funzioni. Infatti questo potrebbe risultare abbastanza semplice nel caso di parametri statici (una certa system call riceve sempre gli stessi parametri in ingresso), ma nel caso di parametri dinamici definire una regola che li controlla potrebbe anche non essere possibile (in generale, non solo con Seccomp): se ogni volta che viene

```
#include <linux/seccomp.h> // Linkare la libreria 'seccomp' in fase di
    compilazione!
...

int main () {
    scmp_filter_ctx seccomp_ctx;
    char buffer [BUF_LEN];
    ...

    /* Inizializza i Dati utili alla Definizione del Filtro */
    if ((seccomp_ctx = seccomp_init (SCMP_ACT_KILL)) == NULL) {
        perror ("[ERROR_init]");
        return 0;
    }
    ...

    /* Definisci le Regole di Filtraggio */
    if (seccomp_rule_add (seccomp_ctx, SCMP_ACT_ALLOW, SCMP_SYS (read), 2,
                    SCMP_A1 (SCMP_CMP_EQ, (scmp_datum_t) buffer),
                    SCMP_A2 (SCMP_CMP_EQ, BUF_LEN)) < 0) {
        perror ("[ERROR_add_read]");
        return 0;
    }
    if (seccomp_rule_add (seccomp_ctx, SCMP_ACT_ALLOW, SCMP_SYS (exit_group),
        0) < 0) {
        perror ("[ERROR_add_exit]");
        return 0;
    }
    ...

    /* Carica il Filtro Definito */
    if (seccomp_load (seccomp_ctx) < 0) {
        perror ("[ERROR_load]");
        return 0;
    }

    /* Programma da Proteggere */
    ...
}
```

Figura 5.9. Esempio di programma protetto da seccomp.

chiamata una system call le si passa sempre dei valori diversi (valori random o letti da tastiera o ricevuti da socket o dipendenti da altri programmi/processi/...) l'unico modo per permettere l'esecuzione del programma senza problemi è quello di permettere l'esecuzione della system call senza analizzare i parametri che riceve in ingresso.

In conclusione, Seccomp risulta un'ottima soluzione per il filtraggio di system call (e di conseguenza delle funzioni che chiamano tali system call) in tempo reale per motivi di sicurezza legata all'esecuzione di un qualunque processo di un sistema. Per questo motivo si ha deciso di utilizzare Seccomp nella parte sperimentale anziché scrivere direttamente programmi eBPF.

```
gcc -Wall protected_program.c -o protected_program -lseccomp
strace ./protected_program
```

Figura 5.10. Compilazione e esecuzione del file `Programmi/programma_target/protected_program.c` allegato alla tesi con il tracciamento delle system call attivo.

```
bash program_manager 2
bash program_manager 2 ls
```

Figura 5.11. Esecuzione senza e con exploit dello script `Programmi/programma_target/program_manager.sh` allegato alla tesi con target il programma protetto da seccomp.

5.6 Protezione del caso studio

Avendo analizzato il caso studio e le funzionalità di seccomp, ora si vuole trovare un modo per proteggere il primo dall'exploit scritto. Per poterlo fare però bisogna innanzitutto capire quali system call il programma ha bisogno per un'esecuzione senza errori o interruzioni. Ci sono vari modi per farlo: si potrebbe per esempio eseguire il programma vulnerabile affiancato da una soluzione per il monitoraggio delle system call (come perf, trace-cmd o eBPF stesso) e scoprire quali system call vengono chiamate in tempo reale. Qui però si è deciso di prendere un'altra strada: si ha scelto di inizializzare la libreria Seccomp per il programma in questione con un'azione di default di uccisione delle system call non permesse e di avere inizialmente una whitelist vuota. A questo punto si ha eseguito il programma che è stato immediatamente ucciso da Seccomp (siccome la whitelist è vuota, nessuna system call è permessa e il programma verrà terminato non appena effettuerà la prima chiamata a una system call). Si ha inserito alla whitelist la system call in questione e poi si ha ricominciato da capo fino a quando il programma non terminava più per filtraggio da parte di Seccomp, ma nel modo atteso/corretto. In questo modo è stato scoperto che le system call necessarie alla sua esecuzione sono: 'fstat', 'read', 'write', 'lseek' e 'exit_group'. Più nel dettaglio, tutto ciò è stato possibile grazie ai comandi rappresentati in figura 5.10 (eseguiti dalla cartella `Programmi/programma_target`): con il primo si compila il programma protetto da Seccomp e con il secondo lo si esegue tracciando tutte le system call chiamate da esso con il programma Strace (di cui non si è parlato nel capitolo 2 perché effettua il monitoraggio in tempo reale delle system call, ma da livello utente tramite del debug automatizzato, appoggiandosi alla libreria Ptrace). Se questi comandi vengono eseguiti con la whitelist completa il programma terminerà con la stringa '+++ exited with 1 +++' (valore di ritorno in caso di successo); se invece prima della compilazione si commentassero/rimuovessero nel codice una o più regole di filtraggio (definite tramite la funzione 'seccomp_rule_add') e si eseguissero di nuovo i due comandi, si otterrebbe la terminazione (in un punto diverso del programma a seconda della system call rimossa) con la stringa '+++ killed by SIGSYS (core dumped) +++' preceduta dalla chiamata alla system call che non fa più parte della whitelist. Tornando all'esecuzione del caso studio e all'exploit, per testare che grazie al filtro Seccomp definito il programma funziona comunque e che è possibile bloccare l'attacco descritto in precedenza, si possono eseguire rispettivamente il primo o il secondo comando rappresentati in figura 5.11 (eseguiti dalla cartella `Programmi/programma_target`). Nel secondo caso l'exploit stamperà a video un messaggio che dimostrerà che il programma target in questo caso non ha tornato il risultato del comando, ma è stato terminato dal segnale SIGSYS (e quindi da Seccomp).

Supponiamo ora di non avere il codice sorgente del programma da proteggere. In questo caso ovviamente non si può aggiungere al codice del programma la definizione del filtro Seccomp. Ma come detto in precedenza, il filtro definito da Seccomp viene ereditato da tutti i figli del processo che lo ha caricato nel kernel. Si può quindi pensare di creare un programma che definisca il filtro necessario a proteggere il programma vulnerabile e poi esegua il programma target e ottenere lo

```
gcc -Wall vulnerable_program.c -o vulnerable_program
gcc -Wall support_program.c -o support_program -lseccomp
strace -f ./support_program
```

Figura 5.12. Compilazione e esecuzione del file `Programmi/programma_target/support_program.c` allegato alla tesi con il tracciamento delle system call attivo.

```
bash program_manager 3
bash program_manager 3 ls
```

Figura 5.13. Esecuzione senza e con exploit dello script `Programmi/programma_target/program_manager.sh` allegato alla tesi con target il programma protetto da seccomp (da un processo padre).

stesso risultato di prima. Ovviamente però bisogna fare attenzione al fatto che il filtro servirà anche al processo padre e quindi dovrà contenere delle regole che permettano anche la sua esecuzione senza interruzioni né problemi. Per scoprire le nuove system call si ha utilizzato la stessa logica di prima, ma con comandi diversi (rappresentati in figura 5.12); la nuova whitelist risulta quindi contenere le system call 'execve', 'brk', 'access', 'openat', 'mmap', 'close', 'mprotect', 'arch_prctl', 'munmap' più quelle definite per il programma vulnerabile. A questo punto si nota subito un possibile problema: la 'execve' deve fare parte delle system call permesse (perché viene utilizzata per eseguire il processo figlio, il programma da proteggere) nonostante sia la stessa che la funzione di libreria 'system' (la funzione che si vuole essere certi di bloccare, oltre possibilmente a tutte le altre illecite) utilizza per l'esecuzione del comando che le si passa. Ma in realtà si scopre che questo non è un problema per due motivi. Prima di tutto se si analizzasse qual'è la system call che viene bloccata da Seccomp durante l'esecuzione dell'exploit con target il programma protetto in uno qualunque dei due modi, si scoprirebbe che non è la system call 'execve' perché prima di chiamare essa la funzione 'system' ne chiama altre che non fanno parte della whitelist (e quindi avere la 'execve' nella whitelist non risulta essere un problema per quanto riguarda questo specifico exploit). Poi come detto in precedenza Seccomp permette non solo di filtrare in base alle system call, ma anche ai parametri che esse ricevono. Senza scrivere direttamente un programma eBPF, come si è deciso di fare qui per semplicità, purtroppo non è possibile filtrare su valori passati per riferimento, ma solo sui puntatori passati alla funzione. Quindi qui non si può inserire nella whitelist la 'execve' specificando che sia permessa solamente se ricevesse in input la stringa per l'esecuzione del programma vulnerabile (esecuzione lecita), ma si può comunque specificare che sia permessa solamente se ricevesse come primo parametro il puntatore a quella stringa. In questo modo se anche un attaccante riuscisse a spingere il programma a chiamare la 'execve', dovrebbe per forza salvare il primo parametro da passare a essa in quella specifica locazione di memoria, cosa che risulterebbe, sempre che sia possibile, altamente improbabile. Per testare il funzionamento di quest'ultima soluzione è possibile eseguire i comandi rappresentati in figura 5.13, dalla cartella `Programmi/programma_target`, ottenendo gli stessi risultati dimostrati per il caso precedente.

Capitolo 6

Risultati ottenuti

6.1 eBPF rispetto a soluzioni simili

Nell'analizzare eBPF si ha capito quanto sia uno strumento potente per l'analisi in tempo reale di un qualunque evento interno a un sistema Linux. Infatti questo è forse uno dei punti migliori di questa tecnologia: se mentre con le soluzioni precedenti a eBPF, accennate nel capitolo 2, si hanno dei programmi in grado di svolgere solo compiti specifici, con esso è possibile monitorare, filtrare e gestire pacchetti di rete e monitorare chiamate a qualunque funzione e system call direttamente e di filtrare chiamate a funzione e system call indirettamente (per esempio tramite Seccomp). Inoltre si ha la possibilità di programmare nel modo più personalizzato possibile il filtro che indica che cosa si vuole o non si vuole monitorare o filtrare. Quest'ultimo punto però in alcuni casi potrebbe anche essere un difetto, per esempio quando si vuole monitorare la rete o le chiamate che un programma fa solamente per motivi di test o debug. In questo caso sembra esagerato dover scrivere un programma eBPF e un programma utente (per quanto sia semplice, soprattutto tramite BCC) e quindi potrebbe essere preferibile utilizzare una delle altre soluzioni, come Wireshark, Trace-cmd o Perf. Bisogna però ricordare che vari programmi utilizzano eBPF sotto la superficie quindi se anche non si avesse il tempo di scrivere il programma eBPF direttamente potrebbe esistere un programma che permette di usare eBPF indirettamente in modo più semplice e veloce (per esempio Tcpdump, Cilium e Seccomp). Infine eBPF è ancora una tecnologia nuova, che cresce insieme al kernel Linux, risultando quindi una soluzione robusta (anche grazie al verifier che controlla i programmi che vengono caricati nel kernel prima di mandarli in esecuzione), fidata (non necessita di moduli di terze parti perché l'intera tecnologia fa parte del kernel stesso), performante (riduce al minimo le comunicazioni tra il kernel e il livello utente) e ben supportata e che ha acquisito nuove funzionalità in un tempo molto breve e soprattutto che potrebbe continuare a acquisirne di nuove nel prossimo futuro.

6.2 Confronto tra i diversi firewall utilizzati

Nel caso specifico della gestione di regole di filtraggio in reti di container, si è visto come Cilium risulti una soluzione molto interessante e utile e come utilizzi eBPF sotto alla superficie, ma trasportando la definizione del filtro a un livello più alto rendendo quindi la sua definizione più semplice, intuitivo e veloce. Esistono però altri firewall utilizzati ormai da anni nel generico networking che possono essere usati in reti di container e come è stato accennato in precedenza nel programma implementato per la gestione delle reti di container e dei loro firewall si ha deciso di inserire anche alcuni di essi. Più precisamente i firewall utilizzati sono:

- Ebtables per il filtraggio a livello 2;
- Tc per il filtraggio a livello 3-4;

- Iptables per il filtraggio a livello 3-4;
- Cilium per il filtraggio a livello 3-4;
- Modsecurity per il filtraggio a livello 7;
- Cilium per il filtraggio a livello 7;
- Ebtables + Modsecurity per il filtraggio a livello 2-7;
- Tc + Modsecurity per il filtraggio a livello 3-4-7;
- Iptables + Modsecurity per il filtraggio a livello 3-4-7;
- Cilium + Modsecurity per il filtraggio a livello 3-4-7;
- Cilium per il filtraggio a livello 3-4-7.

Dalla lista si nota immediatamente che Cilium risulta essere l'unico firewall in grado di filtrare a tutti i livelli necessari (a parte al 2, ma grazie al filtraggio di livello 3 può comunque filtrare basandosi sugli indirizzi IP sorgente e destinazione); per questo motivo sono state definite delle coppie di firewall di livello 2 o 3-4 e di livello 7 così da poter ricoprire tutti i tipi di filtraggio necessari con tutti i firewall. Inoltre è bene sottolineare che uno dei principali vantaggi nell'usare Cilium (anche rispetto all'uso diretto di eBPF) è quello di avere un indirizzamento basato sul nome del container di destinazione (senza bisogno di installare un DNS e di aggiornarne la configurazione nel caso di cambiamento di indirizzo IP del destinatario) e policy di networking basate sulle etichette associate ai container. In tutti gli altri casi se un container cambiasse indirizzo IP bisognerebbe aggiornare come minimo le regole di filtraggio. Lo stesso discorso vale nel caso in cui si aggiungessero nuovi container alla rete basati su regole già esistenti: nuove regole dovrebbero essere aggiunte ai vari firewall, ma non nel caso di Cilium. Inoltre bisogna ricordare il fatto che Cilium (come Iptables), a differenza di Ebtables e Tc, definisce le regole di filtraggio con il meccanismo dello 'Stateful Connection Tracking' che permette di definire meno regole e di conseguenza di avere maggiore filtraggio: se si vuole permettere la comunicazione tra il container1 e il container2 basta creare una sola regola che permetta al primo di comunicare con il secondo, la risposta sarà automaticamente accettata; con Tc e Ebtables invece si è obbligati a definire una regola che permetta al primo di comunicare con il secondo e al secondo di comunicare con il primo, ma questa non è esattamente la stessa cosa perché in questo modo si sta anche dando al container2 il permesso di cominciare la comunicazione. Infine, per quanto riguarda Modsecuity, esso risulta un'ottima soluzione di filtraggio a livello 7, ma bisogna ricordare che si trova a livello utente, cosa che lo rende meno interessante di Cilium (per i motivi già spiegati nell'analisi di soluzioni di livello kernel), che è un modulo di Apache e quindi può essere utilizzato solo nel caso i siti web vengano gestiti tramite esso (altri WAF esistono per altre soluzioni), che va installato all'interno di ogni container che contiene un servizio HTTP(S) (mentre di Cilium ne basta uno per tutti).

Ovviamente però Cilium non ha solo pregi. Un primo svantaggio è il fatto che Cilium necessita di poter controllare l'intera rete di container (per questo motivo viene creata una sotto-rete gestita dal Cilium Daemon a cui connettere i container) mentre le regole degli altri firewall possono essere installate su reti di container così come in sistemi connessi a reti generiche. Poi si è obbligati a scrivere regole di filtraggio in formato JSON, che è certamente più veloce rispetto a scrivere interi programmi eBPF, ma sicuramente scrivere una regola con Iptables risulta più veloce ancora (se si conoscono le varie parole chiave da usare); questo però non è un problema se si utilizza il tool implementato perché le regole vengono definite tutte nello stesso modo, sarà il tool a tradurle in modo diverso a seconda del firewall selezionato (di cui si parlerà meglio nelle guide dell'utente, capitolo 8, e del programmatore, capitolo 9). Inoltre bisogna ricorda che Cilium è un tool molto nuovo e che quindi non possiede ancora tutte le funzionalità per cui è stato progettato (per esempio il filtraggio a livello 7 per il momento è solo utilizzabile tra container e non quando si ha come sorgente o come destinatario un indirizzo IP esterno alla rete). Infine, per quanto riguarda il filtraggio di livello 7, Modsecurity è anche in grado di gestire il protocollo HTTPS, entrando in funzione a livello applicativo, mentre Cilium (almeno per il momento) non utilizza probe per l'analisi delle funzioni di cifratura/decifratura dei pacchetti e quindi è solamente in grado di analizzare il payload dei pacchetti così come si trovano sull'interfaccia di rete (e quindi cifrati nel caso dell'HTTPS).

Per quanto riguarda le prestazioni delle varie soluzioni, sono state svolte delle prove sperimentali su tutti i tipi di firewall precedentemente citati tramite il tool Ab del pacchetto Apache con il quale sono state generate 1000 richieste (con un massimo di 30 richieste in parallelo) verso tutte le pagine del sito web creato. I risultati ottenuti sono contenuti nella cartella `Programmi/Risultati_Calcoli_Prestazioni/rete_container/` allegata alla tesi e le istruzioni su come effettivamente eseguire i test possono essere trovate nella guida dell'utente (capitolo 8). Nelle tabelle 6.1 e 6.2 invece sono stati raccolti i dati principali, in ordine: il firewall attivo sulla rete di container; la pagina richiesta; il tempo impiegato per il test; il numero di richieste effettuate in un secondo; i dati trasmessi al secondo; e un valore positivo/negativo che indica se la richiesta era valida o meno (e quindi se il fireall ha lasciato passare la richiesta/risposta o no). Da essi si può notare immediatamente che Ebtables sembra essere la soluzione più leggera tra tutte quelle che filtrano in base a sorgente/destinazione dei pacchetti (livelli 2 e 3-4) seguita da Cilium di livello 3-4. Spostandosi a livello 7 si nota di nuovo come Cilium sia la soluzione più prestante. Per quanto riguarda invece le soluzioni di filtraggio complete la soluzione Cilium più Modsecurity sembra essere la migliore subito seguita da Ebtables/Iptables più Modsecurity. Le peggiori invece sembrano essere quelle che utilizzano Tc e Cilium a livello 3-4-7 (almeno nel caso preso in considerazione).

Firewall	*Pagina*	*Tempo* [sec]	*N° Richieste* [#/sec]	*Dati trasmessi* [Kbytes/sec]	*Successo*
Nessuno	/	20.321	49.21	24.37	Sì
	/public.php	20.358	49.12	40.20	Sì
	/private.php	21.311	46.92	31.34	Sì
	/vulnerable.php?id=1	21.260	47.04	27.19	Sì
	/vulnerable.php?id='	20.675	48.37	23.57	Sì
ebtables	/	20.964	47.70	23.62	Sì
	/public.php	24.910	40.14	32.85	Sì
	/private.php	25.305	39.52	26.40	Sì
	/vulnerable.php?id=1	25.280	39.56	22.87	Sì
	/vulnerable.php?id='	20.695	48.32	23.55	Sì
tc	/	55.456	18.03	8.93	Sì
	/public.php	52.880	18.91	15.48	Sì
	/private.php	56.529	17.69	11.82	Sì
	/vulnerable.php?id=1	56.073	17.83	10.31	Sì
	/vulnerable.php?id='	57.159	17.50	8.53	Sì
iptables	/	44.320	22.56	11.17	Sì
	/public.php	44.256	22.60	18.49	Sì
	/private.php	43.406	23.04	15.39	Sì
	/vulnerable.php?id=1	44.302	22.57	13.05	Sì
	/vulnerable.php?id='	43.754	22.86	11.14	Sì
cilium lv3-4	/	43.244	23.12	11.63	Sì
	/public.php	38.816	25.76	21.08	Sì
	/private.php	41.661	24.00	16.03	Sì
	/vulnerable.php?id=1	41.460	24.12	13.94	Sì
	/vulnerable.php?id='	40.218	24.86	12.12	Sì
modsecurity	/	36.557	27.35	13.54	Sì
	/public.php	39.041	25.61	20.96	Sì
	/private.php	42.276	23.65	10.76	No
	/vulnerable.php?id=1	39.069	25.60	14.80	Sì
	/vulnerable.php?id='	38.759	25.80	11.82	No
cilium lv7	/	28.349	35.27	4.24	Sì
	/public.php	28.947	34.55	4.15	Sì
	/private.php	28.741	34.79	4.18	No
	/vulnerable.php?id=1	29.072	34.40	4.13	Sì
	/vulnerable.php?id='	28.735	34.80	4.18	No

Tabella 6.1. Prima parte dei risultati dei calcoli sulle prestazioni dei vari firewall analizzati.

Firewall	*Pagina*	*Tempo* [sec]	*N° Richieste* [#/sec]	*Dati trasmessi* [Kbytes/sec]	*Successo*
ebtables + modsecurity	/	32.150	31.10	15.40	Sì
	/public.php	33.971	29.44	24.09	Sì
	/private.php	33.044	30.26	13.77	No
	/vulnerable.php?id=1	34.142	29.29	16.93	Sì
	/vulnerable.php?id='	32.428	30.84	14.12	No
tc + modsecurity	/	40.017	24.99	12.37	Sì
	/public.php	40.964	24.41	19.98	Sì
	/private.php	41.219	24.26	11.04	No
	/vulnerable.php?id=1	41.292	24.22	14.00	Sì
	/vulnerable.php?id='	40.984	24.40	11.18	No
iptables + modsecurity	/	31.320	31.93	15.81	Sì
	/public.php	33.206	30.12	24.65	Sì
	/private.php	32.631	30.65	13.95	No
	/vulnerable.php?id=1	33.278	30.05	17.37	Sì
	/vulnerable.php?id='	32.883	30.41	13.93	No
cilium + modsecurity	/	25.506	39.21	19.69	Sì
	/public.php	27.321	36.60	29.95	Sì
	/private.php	27.563	36.28	16.51	No
	/vulnerable.php?id=1	27.309	36.62	21.17	Sì
	/vulnerable.php?id='	24.828	40.28	18.45	No
cilium lv3-4-7	/	58.699	17.04	2.05	Sì
	/public.php	67.244	14.87	1.79	Sì
	/private.php	56.378	17.74	2.13	No
	/vulnerable.php?id=1	57.673	17.34	2.08	Sì
	/vulnerable.php?id='	57.347	17.44	2.09	No

Tabella 6.2. Seconda parte dei risultati dei calcoli sulle prestazioni dei vari firewall analizzati.

6.3 Il peso dei controlli effettuati da Seccomp

Infine per quanto riguarda il filtraggio delle funzioni interne a un sistema si è visto come controllare l'intero sistema potrebbe essere un problema. Si è quindi passati a analizzare una soluzione per il controllo dei singoli processi e più precisamente delle system call chiamate da essi. Grazie a Seccomp si è visto come questo compito non sia troppo complesso (a parte in fase di definizione del filtro che su programmi molto grossi potrebbe diventare complicato), ma ancora non si ha analizzato quanto questa soluzione pesi sul processo da controllare. Per fare ciò è stato preso il caso studio in tutte e tre le sue forme (vulnerabile, protetto da Seccomp a livello di codice e eseguito da un processo che ne definisce il filtro Seccomp), è stato eseguito, si ha comunicato con esso facendo un paio di richieste di lettura e scrittura per 100.000 volte e poi si ha chiesto al programma di uscire (le istruzioni su come eseguire questo possono essere trovate nella guida dell'utente, capitolo 8). Durante ogni singola esecuzione è stato calcolato il tempo di esecuzione ottenendo i seguenti risultati:

- 6.1345767974853520 secondi per il programma vulnerabile;
- 6.4082818031311035 secondi per il programma protetto da Seccomp a livello di codice;
- 6.4233496189117430 secondi per il programma eseguito tramite il programma di supporto.

Grazie a essi si nota come Seccomp diminuisca le prestazioni del caso studio (cosa ovvia visto che viene effettuato un controllo interno al kernel ogni volta che l'eseguibile chiama una system call), ma anche che il ritardo generato è minimo (almeno in un caso semplice come quello preso in considerazione). Un ulteriore ritardo (anche se molto basso) si ha sull'ultima soluzione probabilmente dovuto al fatto che anziché eseguire un unico programma, ne vengono eseguiti due; questo

è quindi un ritardo che avviene unicamente all'avvio del programma da proteggere: è importante se il target va eseguito molte volte in un tempo molto breve, ma non è da tenere conto se invece va eseguito una volta ogni tanto tempo o comunque se la sua esecuzione dura molto a lungo.

Capitolo 7

Conclusioni

7.1 Considerazioni finali

In questa tesi si è spiegato quanto sia importante integrare in un sistema dei meccanismi di controllo in tempo reale, per scopi di sicurezza. Si hanno analizzato varie soluzioni, arrivando a spiegare come mai eBPF sembri essere la soluzione migliore in alcuni casi. Grazie a esso infatti si ha la possibilità di effettuare ogni tipo di controllo (direttamente o indirettamente) invece di dover ricorrere a programmi diversi per diversi scopi; si ha molta personalizzazione nella definizione delle varie regole, a costo, anche se in minima parte, della velocità e semplicità nella loro definizione; e a volte si arriva anche a avere una migliore ottimizzazione dei controlli stessi. Ma soprattutto si è parlato del fatto che eBPF si trova direttamente nella sorgente del kernel Linux, che è mantenuto e aggiornato insieme a esso. Questo implica che sia una soluzione particolarmente stabile e che si ha molto interesse a continuare a farla evolvere (a differenza di programmi di terze parti che potrebbero cessare di essere aggiornati per vari motivi). Fatto dimostrato anche dalla veloce evoluzione che ha avuto nelle ultime versioni del kernel e che sta avendo tuttora. Ne si deduce che nelle prossime versioni del kernel, eBPF risulterà ancora più interessante di com'è oggi (non solo nel campo della sicurezza) e che di conseguenza lo diventeranno anche i programmi che lo sfruttano per i propri scopi. Inoltre è molto probabile che nasceranno varie altre utilità di livello utente e kernel che sotto la superficie ricorreranno all'utilizzo di eBPF, cosa che renderà più semplice e veloce l'interfacciarsi con esso (a costo ovviamente di una minore programmazione nel filtro eBPF stesso).

7.2 Possibili sviluppi futuri nella sicurezza dei container

Nel caso delle reti di container, si ha parlato delle problematiche che si incontrano nel crearle e soprattutto nel non proteggerle con un firewall (o nel proteggerle solamente nel punto di ingresso dal mondo esterno) e si ha visto che il tool scritto per la gestione di queste reti e della loro protezione sia effettivamente in grado di risolvere questi problemi. Questo tool ha già varie funzionalità, ma è tuttora in una versione basilare. In futuro infatti sarebbe interessante aggiungere ulteriori tipi di firewall sia al comando 'start' (utile all'esecuzione di una rete di container e del suo firewall) che al comando 'install' (utile all'installazione di un firewall su di una rete di container già in esecuzione) o almeno aggiungere ulteriori combinazioni dei firewall già presenti (come per esempio Ebtables a livello 2 e Cilium a livello 7). Potrebbe essere necessario potenziare i firewall già esistenti: il meccanismo di traduzione delle regole definite dall'utente nelle rispettive regole per quel dato firewall potrebbe non gestire tutte le possibili parole chiave del firewall stesso perché per il momento non sembravano abbastanza importanti (come nel caso di Ebtables per esempio). Per quanto riguarda il firewall Cilium invece risulterebbe sicuramente importante aggiornare il meccanismo di traduzione ogni volta che Cilium aggiorna la definizione/gestione delle policy perché essendo un tool molto nuovo e ancora in fase di sviluppo potrebbero essere apportate modifiche in grado di potenziare la definizione del firewall o aumentarne le prestazioni... Infine per quanto riguarda le funzionalità del tool stesso, potrebbe essere molto utile avere delle opzioni di verbose

e debug per avere maggiori informazioni su cosa sta succedendo sotto alla superficie; si potrebbe pensare di portare a un livello più elevato ancora la definizione dei container da eseguire e/o delle regole da tradurre in un firewall (trasformati nei rispettivi file Puppet dal tool stesso); di avere un ulteriore comando che permetta di modificare un firewall già installato (ora per fare questo bisognerebbe disinstallare il firewall e installarlo nuovamente o modificarlo a mano direttamente con i tool ebtables, iptables, cilium, ...); e di aggiungere diverse opzioni per il comando 'exec': per il momento esso è solamente in grado di eseguire file Puppet in modo molto basilare, ma potrebbe essere interessante dare la possibilità all'utente di decidere quando e/o come eseguire un certo file Puppet. Un esempio potrebbe essere uno dei due presi in considerazione per i test di dinamicità in reti protette da firewall Cilium. Nello specifico test si supponeva che i tre container contenenti il sito web fossero troppo carichi e che si avesse la necessità di crearne di nuovi per poter gestire la mole di traffico in arrivo. Questa era una pura supposizione, ma sarebbe interessante dare un opzione al comando 'exec' che permetta di eseguire un certo file Puppet solamente se una certa condizione è verificata o solamente quando una certa condizione si avvererà...

7.3 Possibili sviluppi futuri nel controllo di processi

Per quanto riguarda il controllo delle funzionalità di un sistema, si è visto come convenga controllare i singoli processi anziché l'intero sistema e come Seccomp risulti essere una soluzione più che valida a questo scopo. Purtroppo non si ha avuto la possibilità di implementare un tool per la gestione di regole di filtraggio di questo tipo, ma l'ideale potrebbe essere creare un programma che generi un filtro Seccomp a partire da regole di più alto livello definite dall'utente (anche se Seccomp ha già delle funzioni particolarmente semplici per la definizione del filtro). Ma questo implica comunque che l'utente debba conoscere molto bene il processo da controllare (quali system call necessita di chiamare e con quali parametri statici e/o dinamici), cosa non troppo semplice quando un programma chiama molte system call. Un'alternativa potrebbe essere la scrittura di un programma che cerchi di automatizzare anche la definizione del filtro stesso. Ovviamente finché si tratta di filtrare in base alla system call non è troppo problematico: si analizza in modo statico il programma da controllare, si scoprono quali system call chiama (e quali funzioni di libreria utilizza e le system call che esse chiamano al proprio interno) e si inseriscono nel filtro. Ma quando bisogna definire delle regole sui parametri ricevuti da esse cominciano i problemi: si dovrebbe eseguire un'analisi dinamica con lo scopo di capire quali parametri le varie system call ricevono, se sono valori diretti o puntatori (e in questo caso magari anche analizzare i valori puntati generando il programma eBPF direttamente anziché definire il filtro con le funzioni Seccomp), se sono sempre gli stessi, se sono un certo numero o se sono completamente dinamici, ... Però se l'analisi dinamica fosse effettuata in una sandbox ci si scontrerebbe con i tipici problemi delle sandbox accennati nel capitolo 1, soprattutto su quanto tempo dedicare all'analisi prima di definire il filtro. Infatti, in questo caso si rischierebbe di bloccare l'esecuzione del programma vero e proprio per troppo tempo o si finirebbe per definire un filtro solamente su di una parte del programma, rischiando di bloccarne la reale esecuzione prima che abbia svolto il suo compito. Si potrebbe pensare perciò di mettere in pratica un'analisi più realistica: si analizza staticamente il processo da controllare per poi eseguirlo con il solo filtraggio delle system call; intanto si analizza il suo comportamento dinamico durante il suo reale utilizzo. Questo viene fatto per N volte, alla fine delle quali viene generato il filtro completo e da quel momento in avanti sarà il nuovo filtro a essere associato all'eseguibile. Di nuovo, l'analisi richiederebbe probabilmente dei meccanismi di machine learning e sarebbe tanto difficile quanto interessante.

Comunque, queste sono solamente alcune delle idee che potrebbero tornare utili nel filtraggio delle funzionalità di un sistema, un'operazione particolarmente complessa che necessita certamente ulteriore analisi e sperimentazione.

Capitolo 8

Guida dell'utente

8.1 Introduzione

In questa parte verrà spiegato cosa è necessario installare per poter utilizzare i programmi allegati alla tesi e come eseguirli. Più nel dettaglio, nella prima parte si vedrà come utilizzare i programmi eBPF citati come esempi nel capitolo 3. Nella seconda parte invece si analizzeranno tutte le funzionalità definite nel tool implementato per la gestione di una rete di container e/o di un firewall da installare su di essa. In questo caso si vedrà sia come gestire il caso studio definito a priori, nel capitolo 4, sia come modificare i file di configurazione da passare in ingresso al tool per poter creare reti di container diverse da quella analizzata e per poter personalizzare le regole del firewall da installare (sulla rete definita o su una rete mandata in esecuzione precedentemente, anche non dal tool stesso). Infine, nella terza parte, si vedrà come gestire il programma vulnerabile, protetto o meno da Seccomp, utilizzato come caso studio nel capitolo 5 e del relativo exploit.

8.2 Esempi eBPF

I programmi contenuti nella cartella `Programmi/esempi_eBPF` sono stati scritti appoggiandosi alla libreria BCC per il linguaggio Python. Per poterli eseguire bisogna quindi accertarsi di avere Python installato; bisogna installare la libreria BCC seguendo le istruzioni presenti all'indirizzo https://github.com/iovisor/bcc/blob/master/INSTALL.md; e bisogna installare la libreria Python 'Pyroute2' con il comando 'pip' o direttamente dal Packet Manager (su alcuni sistemi ha il nome 'python-pyroute2', per Python3.X, o 'python2-pyroute2', per Python2.X). In figura 8.1 sono rappresentati i comandi per l'esecuzione degli esempi (da eseguire dalla cartella `Programmi/esempi_eBPF`). Da notare la 'X' dopo a 'python': la libreria BCC, a seconda del sistema utilizzato, di default potrebbe venire installata per Python2 o per Python3; sostituire la X con la versione di Python corretta (e accertarsi di aver installato la libreria Pyroute2 per la stessa versione). Infine, per quanto riguarda i programmi per il monitoraggio/filtraggio della rete ([socket|tc|xdp]_program.py), prima di mandarli in esecuzione potrebbe essere necessario cambiare l'interfaccia di rete su cui installare i programmi eBPF. Così come sono utilizzano l'interfaccia 'eth0'; se necessario modificare il codice dei programmi semplicemente sostituendo 'eth0' con il nome dell'interfaccia da utilizzare.

8.3 Gestione di una rete di container e del suo firewall

Per poter utilizzare il programma implementato per la gestione di reti di container e del loro firewall bisogna prima installare alcune risorse. Nel Packet Manager è possibile trovare Ebtables, per poter utilizzare il firewall Ebtables, Docker, per la gestione dei container, Puppet, per

```
sudo pythonX probe_program.py
sudo pythonX socket_program.py
sudo pythonX tc_program.py
sudo pythonX tracepoint_program.py
sudo pythonX xdp_program.py
```

Figura 8.1. Comandi per l'esecuzione degli esempi eBPF allegati alla tesi (cartella `Programmi/esempi_eBPF`).

```
sudo puppet module install garethr-docker
sudo puppet module install puppetlabs-firewall
```

Figura 8.2. Comandi per l'installazione dei moduli Puppet necessari per il completo funzionamento del programma implementato.

la gestione dei file di configurazione da passare al programma (di cui si parlerà a breve), e Apache2, per il calcolo delle prestazioni dei diversi firewall (attenzione che in alcuni sistemi dopo aver installato Apache2, il servizio associato a esso viene automaticamente lanciato; questo porta a aprire un server sulla porta 80 del sistema; nel momento in cui viene eseguito il caso studio, il 'Load_Balancer' cercherà di mettersi in ascolto sulla porta 80, ma essendo occupata, il container terminerà!). Dopo aver installato Puppet bisognerà eseguire i comandi rappresentati in figura 8.2 per installare due moduli di Puppet utili, in ordine, per la gestione di container tramite Docker e per la creazione di un firewall Iptables. Infine per poter utilizzare il firewall Cilium, bisogna installarne l'architettura. Esistono diversi modi per farlo, qui per semplicità si ha deciso di installare nel sistema principale il Cilium CLI (il client in grado di comunicare con il Cilium Daemon) e di installare il resto dell'architettura Cilium all'interno di una serie di container. Per fare ciò bisogna scaricare l'ultima versione del Cilium CLI da http://releases.cilium.io/v1.2.3/cilium-x86_64 (potrebbe esistere una versione più nuova di quella presente al link citato); per il resto invece bisognerà installare Docker-Compose (dal Packet Manager) che verrà utilizzato per l'esecuzione del file `Programmi/rete_container/cilium/docker-compose.yml` (file scaricato dal Github di Cilium).

Dopo aver installato tutte le risorse necessarie è possibile eseguire il comando rappresentato in figura 8.3 dalla cartella `Programmi/rete_container` per ottenere la risposta raffigurata. Si può notare che se eseguito senza parametri il programma stampa a video un messaggio che ne spiega le varie funzionalità: esso necessita di alcuni parametri, un comando obbligatorio (parentesi graffe) e opzionalmente (parentesi quadre) i parametri per quel dato comando. La lista dei comandi possibili si trova subito sotto, seguiti ciascuno dalla spiegazione del comando stesso (inutile quindi ripeterla qui).

Di seguito verranno spiegati nel dettaglio i vari comandi in ordine alfabetico, supponendo che vengano eseguiti dalla cartella `Programmi/rete_container`. Per ciascuno si vedrà innanzitutto a cosa serve il comando, come utilizzarlo e quali parametri passargli per poter gestire il caso studio preso in considerazione nel capitolo 4; poi si analizzeranno i file di configurazione necessari al suo funzionamento e si daranno istruzioni su come modificarli (o riscriverli) per poter gestire reti di container e firewall diversi da quelli presi in esame.

8.3.1 Comando 'exec' per l'esecuzione di comandi generici

Con il comando 'exec' è possibile eseguire comandi generici su una rete di container o sul firewall a esso associato. Eseguendo il comando senza parametri è possibile ottenere la risposta rappresentata in figura 8.4: il comando prende in ingresso un unico parametro obbligatorio, un file Puppet 'generico'. Questo file deve essere innanzitutto un file Puppet valido [13], deve contenere quindi

```
[nick@host] sudo bash net_manager.sh

       --- Network Manager ---

Utilizzo: ./net_manager.sh {COMANDO} [PARAMETRI_COMANDO]
Per: eseguire/distruggere/gestire una rete di container e/o un firewall

    {COMANDO}:
        exec esegui comandi generici su una rete di container precedentemente
            creata
        install installa un firewall su una rete di container precedentemente
            creata
        start esegui una rete di container e installa su essa un firewall
        stop distruggi una rete di container e il firewall installato su essa
        uninstall disinstalla un firewall precedentemente creato su una rete
            di container
```

Figura 8.3. Messaggio di spiegazione dei diversi comandi del tool `Programmi/rete_container/net_manager.sh`.

```
[nick@host] sudo bash net_manager.sh exec

       --- Network Manager ---

Utilizzo: net_manager.sh exec {file_puppet_generico}
Per: eseguire comandi generici su una rete di container precedentemente creata
```

Figura 8.4. Messaggio di spiegazione del comando 'exec' del tool `Programmi/rete_container/net_manager.sh`.

istruzioni interpretabili dal tool 'puppet', e poi deve essere 'generico' (nel senso che non deve seguire nessuna particolare regola, se non quelle del linguaggio Puppet, a differenza per esempio di file di configurazione necessari agli altri comandi, che dovranno sottostare a alcune condizioni di cui si parlerà nelle apposite sezioni).

Per quanto riguarda il caso studio, nel capitolo 4 questo comando è stato utilizzato per i test legati alla dinamicità del firewall Cilium passandogli in ingresso uno dei tre file Puppet scritti appositamente per questo scopo (di nome `Programmi/rete_container/source_exec*.pp`), a seconda del test da effettuare. In figura 8.5 è rappresentato per esempio il primo dei tre file implementati (`Programmi/rete_container/source_exec1.pp`) utilizzato per riavviare uno dopo l'altro i tre container contenenti le copie del sito web: si può notare un ciclo sui nomi dei tre container per l'esecuzione di due blocchi 'exec' (che nel linguaggio Puppet indicano l'esecuzione di comandi di sistema), uno di riavvio del container e l'altro di attesa. Come si può notare si tratta di un normalissimo file Puppet, con l'unica particolarità di avere la variabile '$system_path' non dichiarata/inizializzata. Per il momento infatti l'unico vantaggio nell'eseguire un file Puppet 'generico' con il tool implementato anziché direttamente con il comando 'puppet' è quello di avere due variabili predefinite, '$system_path' (contiene il contenuto della variabile d'ambiente '$PATH') e '$pwd' (contiene il path assoluto alla cartella corrente). In futuro potrebbe essere interessante avere ulteriori vantaggi sotto forma di opzioni per il comando 'exec' stesso, come spiegato nel capitolo 7. Comunque, per eseguire il file preso in considerazione basta lanciare il comando rappresentato in figura 8.6 (dopo aver eseguito il caso studio).

Per quanto invece riguarda l'utilizzo di questo comando su reti diverse dal caso studio, come già

```
# Riavvia i Servizi dei Vari Server Web (Uno Ogni 30 Secondi) #
["web-server0", "web-server1", "web-server2"].each |Integer $index, String
    $value| {
        exec { "${value}_restart_${index}":
               command => "docker restart ${value}",
               path => "${system_path}",
               before => Exec["${value}_wait_${index}"]
        }
        exec { "${value}_wait_${index}":
               command => "sleep 30",
               path => "${system_path}"
        }
}
```

Figura 8.5. File `Programmi/rete_container/source_exec1.pp`.

```
sudo bash net_manager.sh exec source_exec1.pp
```

Figura 8.6. Esecuzione del comando 'exec' con file di configurazione `Programmi/rete_container/source_exec1.pp`.

detto in precedenza, basta scrivere un file Puppet corretto senza dover seguire regole particolari, con anche la possibilità di appoggiarsi alle due variabili predefinite, e poi basta passare questo file in input al comando 'exec' esattamente come descritto sopra.

8.3.2 Comando 'install' per l'installazione di un firewall

Con il comando 'install' è possibile installare un firewall per la protezione di una rete di container già presente nel sistema. Come questa rete sia stata creata non fa differenza: potrebbe essere stata eseguita dallo stesso tool (tramite il comando 'start') o eseguita con qualche altro programma o direttamente a mano; l'importante è che esista. Eseguendo il comando senza parametri è possibile ottenere la risposta rappresentata in figura 8.7: si può notare che al comando bisogna passare due parametri obbligatori, un file di configurazione Puppet di tipo 'init' e l'ID del firewall da installare (la lista dei possibili ID è stampata subito sotto). Per quanto riguarda il file Puppet, in questo caso oltre a dover essere un file Puppet valido deve anche sottostare alle regole definite dal tool per un file di configurazione di questo tipo, le quali verranno spiegate a breve.

Per quanto riguarda il caso studio, nel capitolo 4 questo comando è stato utilizzato per installare i tre firewall Cilium (uno per volta) sulla rete di container generata in precedenza, con lo stesso tool, senza alcuna protezione. La stessa cosa ovviamente può essere fatta con uno degli altri firewall presenti in lista. Per esempio in figura 8.8 è rappresentato il comando da eseguire per installare il firewall Ebtables sul caso studio precedentemente eseguito senza difesa.

Se invece si volesse utilizzare questo comando per installare un firewall su una rete di container diversa dal caso studio (o per un qualunque altro motivo si volesse installare un firewall diverso da quello definito), bisognerebbe ovviamente passargli in ingresso il relativo file Puppet (di tipo 'init'). Da sottolineare è il fatto che anche se questo tipo di file di configurazione è diverso da quello 'generico' usato per il comando 'exec', anche qui si ha la possibilità di utilizzare le due variabili '$pwd' e '$system_path' senza doverle dichiarare. Comunque in figura 8.9 è rappresentato lo scheletro del file 'init' usato per il caso studio. Si nota innanzitutto che il file è diviso in quattro sezioni: in 'CONTAINERS' vanno definite le regole per la creazione della rete di container; in 'FIREWALL' vanno definite le regole del firewall; in 'CILIUM' va definito il metodo di creazione

```
[nick@host] sudo bash net_manager.sh install

       --- Network Manager ---

Utilizzo: net_manager.sh install {file_puppet_init} {ID_FIREWALL}
Per: installare un firewall su una rete di container precedentemente creata

    {ID_FIREWALL}:
        1 ebtables (lv 2)
        2 tc (lv 3-4)
        3 iptables (lv 3-4)
        4 cilium (lv 3-4)
        5 cilium (lv 7)
        6 cilium (lv 3-4-7)
```

Figura 8.7. Messaggio di spiegazione del comando 'install' del tool `Programmi/rete_container/net_manager.sh`.

```
sudo bash net_manager.sh install source_init.pp 1
```

Figura 8.8. Esecuzione del comando 'install' con file di configurazione `Programmi/rete_container/source_init.pp` e firewall Ebtables.

dell'architettura Cilium; e in 'CILIUM ROUTES' vanno definite le regole di routing che permettono al mondo esterno di interagire con la rete di container (quando protetta da firewall Cilium). Non tutte le sezioni devono per forza essere presenti nel file di configurazione 'init': la prima nel caso di comando 'install' per esempio non serve (infatti questa sezione verrà spiegata solamente per il comando 'start'); le ultime due invece non sono necessarie se non si vuole utilizzare il firewall Cilium. L'importante è che ci siano le sezioni che si vuole utilizzare, che siano definite nell'ordine qui rappresentato e che ciascuna contenga le variabili utilizzate in quella sezione (se ne esistono). Inoltre la sezione 'FIREWALL' è suddivisa in tre sottosezioni: anche qui non è obbligatorio che ci siano tutte (se si vuole utilizzare solo il firewall Ebtables basta scrivere solamente le regole del firewall di livello 2), ma se si decide di inserire regole su due o più livelli, è necessario che siano scritte nell'ordine rappresentato; sarà il tool a prelevare il firewall corretto a seconda dell'ID passatogli da riga di comando. Infine si possono notare delle variabili definite all'interno di alcuni commenti nel firewall di livello 3-4 e nella sezione 'CILIUM': nel primo caso si tratta di specificare che tipo di log attivare per Cilium (tutti i pacchetti catturati, tutti i pacchetti eliminati e/o le richieste/risposte di livello 7) e quanto dettagliato deve essere (0 o 1); nel secondo caso invece si stanno definendo dei parametri importanti per l'installazione del firewall Cilium su una rete già in esecuzione (queste variabili vengono quindi usate unicamente nel comando 'install'). Per spiegare a cosa serve il secondo gruppo di variabili, bisogna ricordare che Cilium crea una sottorete virtuale gestita dal Cilium Daemon alla quale vanno connessi i container da controllare. Siccome qui si sta cercando di installare un firewall su una rete già esistente (e non connessa a Cilium), quello che viene fatto è spostare uno a uno i container all'interno della sottorete Cilium (dopo aver eseguito l'architettura Cilium ovviamente). Ma per fare ciò servono alcune informazioni: 'CONTAINERS' contiene la lista dei nomi dei container da spostare (da notare che si tratta di tutti i container generati dal caso studio); 'SERVICES' contiene la lista dei container che vanno riavviati una volta che tutti i container sono stati spostati nella rete (nel caso studio è necessario riavviare il 'Load_Balancer' perché la sua configurazione dipende dai container contenenti le copie del sito web e quindi bisogna fare in modo che la sua configurazione venga aggiornata); 'ROOT_PATH' contiene il path alla cartella in cui sono definiti i container (qui la cartella corrente), necessario per aggiornare eventuali file di configurazione (per esempio quello del 'Load_Balancer') sostituendo gli

```
##### CONTAINERS ##### (Non cambiare/cancellare questa riga!)
...

##### FIREWALL ##### (Non cambiare/cancellare questa riga!)
...
### Firewall Livello 2 ###
...
### Firewall Livello 3-4 ###
# Inserire qui sotto le informazioni necessarie all'attivazione del log nel
    caso di firewall cilium. #
# Nello specifico inserire la lista dei tipi di evento da salvare (o 'none'
    nel caso in cui non si volesse attivare il log) #
# & definire se si vuole un log basilare o più dettagliato. #
# (Non cambiare/cancellare le parole scritte in maiuscolo!) #
# EVENT_TYPES: capture, drop, l7
# VERBOSE_LOG: 0
...
### Firewall Livello 7 ###
...

##### CILIUM ##### (Non cambiare/cancellare questa riga!)

# Inserire qui sotto le informazioni necessarie all'installazione del
    firewall cilium in una rete di container precedentemente creata. #
# Nello specifico inserire la lista dei container da spostare nella rete
    cilium separati da virgola #
# & il path (assoluto o relativo alla cartella 'Programmi/rete_container/')
    alla cartella root della rete di container #
# & la lista dei servizi da riavviare separati da virgola. #
# (Non cambiare/cancellare le parole scritte in maiuscolo!) #
# CONTAINERS: web-server0, web-server1, web-server2, web-server-db,
    web-server-admin, load-balancer
# ROOT_PATH: .
# SERVICES: load-balancer
...

##### CILIUM ROUTES ##### (Non cambiare/cancellare questa riga!)
...
```

Figura 8.9. Scheletro del file `Programmi/rete_container/source_init.pp`.

indirizzi IP utilizzati in precedenza con i nomi dei container utilizzati per l'indirizzamento nella rete Cilium.

Si passa ora a analizzare più nel dettaglio le varie sezioni. Vista la sua semplicità, la sezione 'CILIUM' non viene riportata qui. Infatti l'unica cosa che viene fatta è definire due blocchi Puppet di tipo 'exec' per lanciare comandi di sistema: 'docker-compose' per l'esecuzione dell'architettura Cilium (appoggiandosi al file di cui si è parlato in precedenza) e 'docker' per la creazione della rete a cui connettere i container. A parte in casi eccezionali questa sezione può anche non essere modificata visto che non è direttamente connessa al caso studio, ma unicamente al modo in cui si ha deciso di generare l'architettura Cilium. Anche la sezione 'CILIUM ROUTES' non viene rappresentata visto che contiene unicamente delle regole di routing che permettono al mondo esterno di connettersi al 'Load_Balancer' e al 'Pannello_Amministratore' scritte allo stesso modo in cui viene scritto il firewall di livello 3 (il motivo verrà spiegato a breve). Ovviamente queste

regole vanno modificate per una rete diversa, ma non vanno per forza scritte in questo modo (è quindi possibile usare meccanismi di routing diversi da quello usato qui), basta che siano scritte in un formato Puppet valido.

Più interessante invece è la sezione 'FIREWALL'. Come visto sopra, questa può contenere fino a tre tipi di firewall: quello di livello 2, quello di livello 3-4 e quello di livello 7. Il primo e l'ultimo dei tre non sono mai interpretabili direttamente da Puppet, ma sono definiti nel modo più simile possibile a quello di livello 3-4 che nel caso di Iptables invece viene interpretato direttamente da Puppet (per questo motivo le regole di routing sono state scritte come firewall di livello 3, perché così Puppet le trasformerà in regole Iptables). Questo perché Puppet contiene un modulo per la definizione di blocchi ciascuno dei quali rappresenta una regola che viene tradotta nel rispettivo comando Iptables. Ovviamente però il modulo così com'è, come Iptables stesso, ha bisogno di definire le regole tramite indirizzi IP. Per portare a un livello più alto la definizione delle regole del firewall, nel file di configurazione è possibile definire le regole tramite i nomi dei container (questo per ogni livello di firewall), sarà il tool a tradurli nell'indirizzo IP/MAC assegnato (così come sarà il tool a tradurre le regole nei comandi necessari per i firewall diversi da Iptables).

In figura 8.10 è rappresentata una parte del firewall di livello 2 definito per il caso studio. Come si può notare ogni regola assomiglia a un blocco Puppet (composto quindi da un nome e da una lista di coppie chiave-valore); le possibili chiavi sono 'source'/'destination' per il filtraggio sul MAC sorgente/destinazione del pacchetto, 'ip_source'/'ip_destination' per il filtraggio sull'IP sorgente/destinazione del pacchetto, 'proto'/'ip_proto' per il filtraggio sul protocollo rispettivamente di livello 2 e 4 del pacchetto, 'ip_sport'/'ip_dport' per il filtraggio sulla porta sorgente/destinazione, 'action'/'jump' per l'azione da eseguire su quel pacchetto e 'chain' per indicare in quale tabella Ebtables andare a inserire la regola; i possibili valori sono invece i valori accettati da Ebtables a parte per i vari sorgente/destinazione dove è possibile utilizzare il nome del container al posto dell'indirizzo MAC. Tornando alla figura, si può notare la definizione di una variabile (che rappresenta l'indirizzo IP dell'amministratore); due regole per permettere la comunicazione tra il 'Load_Balancer' e i primi due 'Web_Server' (nella tabella 'FORWARD', le cui regole vengono attivate per pacchetti scambiati tra container); una regola di log riguardante il 'Pannello_Amministratore' (il tool non solo attiverà il log, ma preleverà le informazioni stampate da Ebtables e le stamperà su un file nella cartella `./logs/`); e la regola che permette all'amministratore di accedere al 'Pannello_Amministratore' (nella tabella 'INPUT', le cui regole vengono attivate per pacchetti in arrivo dall'esterno). Ovviamente ci sono altre regole, quelle definite nel capitolo 4, ma qui non sono state riportate perché molto simili a queste.

In figura 8.11 è invece rappresentata una parte del firewall di livello 3-4 definito per il caso studio. Come anticipato in questo caso si tratta di veri e propri blocchi Puppet definiti nel modulo Puppet Firewall. Di conseguenza qui è possibile usare tutte le coppie di chiave-valore definite dalla documentazione ufficiale [14] con in più la possibilità di usare i nomi dei container al posto degli indirizzi IP nei campi 'source' e 'destination'. Ovviamente nel caso si chieda al tool di installare il firewall Iptables, il firewall verrà generato direttamente dalle regole scritte (dopo aver tradotto i nomi dei container nei rispettivi indirizzi IP); negli altri casi sarà il tool a tradurre ogni blocco nel rispettivo comando. Nel caso di blocchi non validi per un certo firewall (per esempio con Iptables si potrebbe decidere di creare una regole di SNAT/DNAT che non può essere gestita da Cilium), questi vengono ignorati (comunque una comune whitelist verrà accettata da tutti i tipi di firewall); e nel caso di blocchi ridondanti per Cilium, il tool si occupa automaticamente di rimuovere le regole di troppo. Un esempio di questa seconda funzionalità sono le tre regole che permettono la comunicazione tra il 'Load_Balancer' e i tre 'Web_Server': in Iptables servono tre regole, ma in Cilium ne basta una (perché tutti i siti web hanno la stessa etichetta). A questo punto se si utilizza Iptables vanno scritte tutte e tre, se si utilizza Cilium ne basta anche solo una delle tre, e se c'è la possibilità che venga usato uno qualunque dei due (come in questo caso) non c'è alcun problema nello scriverle tutte e tre. Per quanto riguarda ciò che è raffigurato, qui non viene descritto in quanto rappresenta esattamente la stessa parte di firewall vista a livello 2 (con l'unica differenza che si filtra sugli indirizzi IP anziché sui MAC e che la tabella in cui salvare le regole è 'DOCKER', la tabella creata automaticamente da Docker nel momento della creazione della rete di container).

In figura 8.12 è rappresentato l'intero firewall di livello 7 definito per il caso studio. Anche qui non si tratta di blocchi interpretabili da Puppet, ma vengono tradotti dal tool implementato

```
$chosen_ip = "192.168.1.10"
...
firewall_l2 { '000 - load_balancer<->web_server0':
        source => "load-balancer",
        destination => "web-server0",
        action => "accept",
        chain => "FORWARD"
}
firewall_l2 { '001 - load_balancer<->web_server1':
        source => "load-balancer",
        destination => "web-server1",
        action => "accept",
        chain => "FORWARD"
}
...
firewall_l2 { '009 - world<->admin LOG':
        destination => "web-server-admin",
        jump => "LOG",
        chain => "INPUT"
}
firewall_l2 { '010 - chosen_ip<->admin':
        proto => "IPv4",
        ip_source => "${chosen_ip}",
        destination => "web-server-admin",
        action => "accept",
        chain => "INPUT"
}
...
```

Figura 8.10. Parte delle regole di livello 2 contenute nel file `Programmi/rete_container/source_init.pp`.

e siccome filtrare a livello 7 è molto diverso che filtrare a livello 2 e 3-4, si nota subito come le chiavi e i valori siano molto diversi da prima. Con la chiave 'modsecurity_target' (utilizzata solo da Modsecurity) si definisce il path al file di configurazione di Modsecurity in cui andare a salvare la regola definita da quel dato blocco; con le chiavi 'cilium_source' e 'cilium_destination' (utilizzate solo da Cilium) si definiscono i container per cui quella regola è valida (è possibile definire entrambi tramite l'indirizzo IP o il nome del container, ma il secondo deve per forza definire anche la porta di destinazione); e con la chiave 'rule' si definisce la vera e propria regola di filtraggio nel formato definito per Modsecurity [15]. Nella figura si può notare un primo blocco valido per entrambi i firewall (ovviamente il tool elaborerà solo le parti del blocco utili al firewall richiesto) che permette solamente all'indirizzo IP dell'amministratore di fare qualunque richiesta (filtrando sull'header 'X-Forwarded-For', come spiegato nel capitolo 4); due blocchi centrali, uno per Modsecurity e uno per Cilium, che rappresentano le richieste che un client qualunque può fare; e un ultimo blocco, utile solo a Modsecurity, per bloccare tutte le richieste non permesse in precedenza (non necessario a Cilium perché è il comportamento che utilizza di default, come spiegato nel capitolo 4). Il motivo per cui i due blocchi centrali sono stati divisi in due nonostante abbiamo lo stesso scopo è per dimostrare che il Regex di Modsecurity è più potente di quello utilizzato in Cilium. Questo perché mentre il primo utilizza un Regex più vasto, il secondo usa il Regex del linguaggio Go, il quale ha delle limitazioni (per ottenere maggiori prestazioni). In questo caso non è possibile definire una negazione: nella regola associata a Modsecurity si sta dicendo che un client qualunque può chiedere tutte le pagine tranne quella privata; nella regola associata a Cilium si sta dicendo che un client qualunque può solamente chiedere le pagine elencate (e quindi non quella privata); in entrambe si sta dicendo che un client qualunque può richiedere la pagina vulnerabile, ma in modo tale da non

```
$chosen_ip = "192.168.1.10"
...
firewall { '000 - load_balancer<->web_server0':
        proto => "tcp",
        source => "load-balancer",
        destination => "web-server0",
        action => "accept",
        chain => "DOCKER"
}
firewall { '001 - load_balancer<->web_server1':
        proto => "tcp",
        source => "load-balancer",
        destination => "web-server1",
        action => "accept",
        chain => "DOCKER"
}
...
firewall { '008 - all<->admin LOG':
        proto => "tcp",
        destination => "web-server-admin",
        jump => "LOG",
        chain => "DOCKER"
}
firewall { '009 - chosen_ip<->admin':
        proto => "tcp",
        source => "${chosen_ip}",
        destination => "web-server-admin",
        action => "accept",
        chain => "DOCKER"
}
...
```

Figura 8.11. Parte delle regole di livello 3 contenute nel file `Programmi/rete_container/source_init.pp`.

fare scattare nessuna delle vulnerabilità trovate nell'analisi del caso studio (capitolo 4).

8.3.3 Comando 'start' per la creazione di una rete di container

Con il comando 'start' è possibile eseguire una rete di container e opzionalmente proteggerla con un firewall. Eseguendo il comando senza parametri è possibile ottenere la risposta rappresentata in figura 8.13: si può notare che al comando bisogna passare due parametri obbligatori, un file di configurazione Puppet di tipo 'init' (lo stesso utilizzato nel comando 'install') e l'ID del firewall da installare (la lista dei possibili ID è stampata subito sotto). Il numero di firewall installabili in questo caso è maggiore rispetto al comando 'install' perché Modsecurity è un firewall che va installato dentro al container che contiene il sito web da proteggere. Questa operazione è più semplice nel caso in cui il container vada costruito, ma risulterebbe più lunga con il comando 'install' a meno che non ci si trovi in una condizione ideale (avere già un container con Apache e Modsecurity installati e il file di configurazione di Modsecurity contenuto in una cartella che è montata come volume per il container) e quindi per il momento si è deciso di poterlo usare unicamente con il comando 'start'.

Per quanto riguarda il caso studio, nel capitolo 4 questo comando è stato utilizzato per eseguire il caso studio senza alcuna protezione (ID firewall pari a 0). Ovviamente con questo comando è

```
$chosen_ip = "192.168.1.10"
...
firewall_l7 { 'admin_rule':
       modsecurity_target => "${pwd}/web_server/waf_rules/rules.conf",
       cilium_source => "load-balancer",
       cilium_destination => "web-server0:80",
       rule => "SecRule REQUEST_HEADERS:x-forwarded-for \"${chosen_ip}\"
           \"id:100,phase:1,t:none,nolog,pass,ctl:ruleEngine=off\""
}
firewall_l7 { 'public_pages':
       modsecurity_target => "${pwd}/web_server/waf_rules/rules.conf",
       rule => "SecRule REQUEST_URI \"@rx
           ^/$|^/(?!private)[a-z]*\\.php(\\?id=)?[0-9]*$\" \"id:101,allow\""
}
firewall_l7 { 'public_pages':
       cilium_source => "load-balancer",
       cilium_destination => "web-server0:80",
       rule => "SecRule REQUEST_URI \"@rx
           ^/$|^/index.php$|^/public.php$|^/vulnerable.php(\\?id=)?[0-9]*$\"
           \"id:102,allow\""
}
firewall_l7 { 'deny_all':
       modsecurity_target => "${pwd}/web_server/waf_rules/rules.conf",
       rule => "SecRule REQUEST_URI \".\"
           \"id:103,severity:ERROR,deny,status:404\""
}
```

Figura 8.12. Parte delle regole di livello 7 contenute nel file `Programmi/rete_container/source_init.pp`.

possibile eseguire la stessa rete di container direttamente protetta da uno dei firewall presenti in lista. Per esempio in figura 8.14 è rappresentato il comando da eseguire per eseguire la rete di container presa in esame e proteggerla con il firewall Ebtables.

Se invece si volesse utilizzare questo comando per eseguire una rete di container diversa dal caso studio, con o senza firewall, bisognerebbe ovviamente passargli in ingresso il relativo file Puppet (di tipo 'init'). La struttura di questo file è stata quasi interamente spiegata nella sezione dedicata al comando 'install'; in breve: il file è composto da quattro sezioni di cui tre sono state spiegate prima (perché necessarie al comando 'install'). La prima, 'CONTAINERS' è invece l'unica fondamentale in questo caso perché è quella nella quale vanno definiti i comandi Puppet per la creazione della rete di container. Ovviamente se si volesse anche proteggere la rete con un firewall bisognerebbe aggiungere dopo a questa sezione la sezione 'FIREWALL' (seguendo le regole definite in precedenza) e nel caso di firewall Cilium anche le sezioni 'CILIUM' e 'CILIUM ROUTES', nell'ordine definito sopra. Di nuovo, è possibile inserire tutte le sezioni in ogni caso, sarà il tool a preoccuparsi di elaborare solo quelle necessarie (a seconda di come è stato eseguito). In figura 8.15 è presente la prima parte della sezione 'CONTAINERS' per la creazione del caso studio. Come si può notare è un file Puppet contenente dei blocchi 'exec' (potrebbe contenere anche altri generici blocchi Puppet) e dei blocchi per la gestione dei container tramite Docker. Essi sono definiti dal modulo Puppet Docker e possono contenere tutte le coppie chiave-valore presenti nella documentazione ufficiale [16]. Per esempio per la prima parte del caso studio (quella raffigurata) vengono definiti due blocchi per creare le immagini dei 'Web_Server' e del 'Database' a partire dai rispettivi Dockerfile (trovati nelle cartelle `./web_server` e `./database`) e due blocchi per l'esecuzione dei container: viene definito il nome dell'immagine da eseguire, il nome del container, l'etichetta, i volumi da montare (nel primo caso) e una variabile d'ambiente (nel secondo). Infine si può notare che tra l'esecuzione di un container e l'esecuzione del container successivo sono stati inseriti dei comandi di

```
[nick@host] sudo bash net_manager.sh start

       --- Network Manager ---

Utilizzo: ./net_manager.sh start {file_puppet_init} {ID_FIREWALL}
Per: eseguire una rete di container e installare su essa un firewall

   {ID_FIREWALL}:
        0 nessun firewall
        1 ebtables (lv 2)
        2 tc (lv 3-4)
        3 iptables (lv 3-4)
        4 cilium (lv 3-4)
        5 modsecurity (lv 7)
        6 cilium (lv 7)
        7 ebtables + modsecurity (lv 2-7)
        8 tc + modsecurity (lv 3-4-7)
        9 iptables + modsecurity (lv 3-4-7)
       10 cilium + modsecurity (lv 3-4-7)
       11 cilium (lv 3-4-7)
```

Figura 8.13. Messaggio di spiegazione del comando 'start' del tool `Programmi/rete_container/net_manager.sh`.

```
sudo bash net_manager.sh start source_init.pp 1
```

Figura 8.14. Esecuzione del comando 'start' con file di configurazione `Programmi/rete_container/source_init.pp` e firewall Ebtables.

'sleep': questo non è obbligatorio, ma è stato fatto per essere certi che il networking del container appena eseguito finisca di essere elaborato prima che venga elaborato quello del successivo (se così non fosse si rischia che il secondo container ottenga un indirizzo IP prima del primo e quindi un indirizzo diverso da quello atteso). Cambiando la definizione dei container quindi viene cambiata la rete di container da mandare in esecuzione. L'unica particolarità a cui fare attenzione è il fatto che il nome dei tre container contenenti il sito web viene definito tramite una variabile, per questo motivo segue un commento (che non va per forza definito nella stessa riga) che definisce la lista di valori possibili per quella data variabile. Il motivo per cui questo è indispensabile è spiegato nell'apposita sezione della guida del programmatore (capitolo 9).

Un'ultima considerazione: a differenza del comando 'install' che può essere eseguito da una qualunque cartella (facendo attenzione a passare il path, relativo o assoluto, corretto al file di configurazione Puppet), per il momento il comando 'start' deve essere relativo alla cartella contenente tutte le informazioni necessarie alla creazione della rete di container. Questo implica che il tool implementato debba essere inserito nella cartella che descrive la rete di container (come fatto per il caso studio). Anche questo è spiegato nella guida del programmatore (capitolo 9).

8.3.4 Comando 'stop' per la distruzione di una rete di container

Con il comando 'stop' è possibile fermare una rete di container precedentemente eseguita con il comando 'start' e disinstallare il firewall associato a esso (se presente). Eseguendo il comando senza parametri è possibile ottenere la risposta rappresentata in figura 8.16: si può notare che al comando bisogna passare due parametri obbligatori, un file di configurazione Puppet di tipo 'exit' e il livello

```
# Costruisci l'Immagine Docker del Server (se Necessario) #
docker::image { 'web_server':
       docker_dir => "web_server"
}

# Esegui il Server in Diversi Container #
["web_server", "web_server", "web_server"].each |Integer $index, String
    $value| {
       docker::run { "${value}${index}":
              image => "${value}",
              volumes => ["${pwd}/web_server/source:/var/www/html",
                  "${pwd}/web_server/waf_rules:/usr/share/...",
                  "${pwd}/logs/${value}${index}:/var/log/apache2"],
              name => ["web-server${index}"], # web-server${index} =
                  "web-server0", "web-server1", "web-server2"
              labels => ["id=${value}"],
              extra_parameters => ["--rm"],
              before => Exec["${value}${index}_wait"]
       }
       exec { "${value}${index}_wait":
              command => "sleep 3",
              path => "${system_path}"
       }
}

# Costruisci l'Immagine Docker del Database (se Necessario) #
docker::image { 'web_server_db':
       docker_dir => "database"
}

# Esegui il Database #
docker::run { 'web_server_db':
       image => "web_server_db",
       env => ["MYSQL_ROOT_PASSWORD=root_pass"],
       name => ["web-server-db"],
       labels => ["id=web_server_db"],
       extra_parameters => ["--rm"],
       before => Exec["web_server_db_wait"]
}
exec { 'web_server_db_wait':
       command => "sleep 10",
       path => "${system_path}"
}
...
```

Figura 8.15. Prima parte della sezione 'CONTAINERS' del file `Programmi/rete_container/source_init.pp`.

di pulizia richiesto (la lista dei possibili valori è stampata subito sotto). Per quanto riguarda il file Puppet, in questo caso oltre a dover essere un file Puppet valido deve anche sottostare alle regole definite dal tool per un file di configurazione di questo tipo, le quali verranno spiegate a breve.

Per quanto riguarda il caso studio, nel capitolo 4 questo comando è stato utilizzato per fermare la rete di container generata in precedenza, con lo stesso tool. In figura 8.17 è rappresentato lo stesso comando.

```
[nick@host] sudo bash net_manager.sh stop

       --- Network Manager ---

Utilizzo: ./net_manager.sh stop {file_puppet_exit} {LIVELLO_PULIZIA}
Per: distruggere una rete di container e il firewall installato su essa

   {LIVELLO_PULIZIA}:
      0 esegui il primo livello di pulizia
      1 esegui i livelli di pulizia 0 e 1
      2 esegui tutti i livelli di pulizia
```

Figura 8.16. Messaggio di spiegazione del comando 'stop' del tool `Programmi/rete_container/net_manager.sh`.

```
sudo bash net_manager.sh stop source_exit.pp 0
```

Figura 8.17. Esecuzione del comando 'stop' con file di configurazione `Programmi/rete_container/source_exit.pp` e livello di pulizia 0.

Se invece si volesse utilizzare questo comando per rimuovere una rete di container diversa dal caso studio, bisognerebbe ovviamente passargli in ingresso il relativo file Puppet (di tipo 'exit') che come nel caso del file 'init' ha le variabili '$pwd' e '$system_path' predefinite ed è suddiviso in più sezioni, come si può notare dalla figura 8.18. Queste sezioni non sono obbligatorie (ne basta anche solo una, ma se ne viene definita più di una devono essere in ordine), ma si è deciso di dividere il file 'exit' in questo modo per comodità. Prendendo in considerazione il caso studio per esempio, si è deciso di utilizzare il primo livello di pulizia solamente per fermare la rete di container (con i blocchi Puppet 'docker::run' e comandi 'ensure => absent') e disinstallare il possibile firewall associato a esso (quando installato con il comando 'start'); il secondo livello di pulizia per rimuovere le immagini che rappresentano i container da generare (con i blocchi Puppet 'docker::image' e comando 'ensure => absent'); e il terzo livello di pulizia per rimuovere anche le immagini usate come base per creare le immagini che rappresentano i container da generare. Grazie a questa suddivisione è possibile gestire la rete e il suo firewall nel modo più veloce possibile: se si vuole rimuovere la rete e ricrearla è inutile fare una pulizia completa perché questo implicherebbe dover riscaricare le immagini sorgente, ricreare quelle finali e poi eseguire il tutto; se invece si ha finito con la rete è comodo poter ripulire completamente il sistema da ciò che è stato installato. Per quanto invece riguarda i blocchi Puppet utilizzabili, tutto è lecito (blocchi generici, del modulo Puppet Docker, del modulo Puppet Firewall, ...) purché valido per il linguaggio Puppet.

Infine, se un firewall è stato installato sulla rete di container con il comando 'install' conviene prima disinstallare il firewall con il comando 'uninstall' e poi fermare la rete di container con il comando 'stop'.

8.3.5 Comando 'uninstall' per la disinstallazione di un firewall

Con il comando 'uninstall' è possibile disinstallare un firewall precedentemente installato con il comando 'install' (per disinstallarne uno installato con il comando 'start' bisogna per forza utilizzare il comando 'stop' per il momento). Eseguendo il comando senza parametri è possibile ottenere la risposta rappresentata in figura 8.19: si può notare che al comando bisogna passare unicamente un parametro obbligatorio che rappresenta l'ID del firewall da disinstallare (la lista dei possibili

```
##### LIVELLO_PULIZIA_0 ##### (Non cambiare/cancellare questa riga!)

# Ferma tutti i Container & Disabilita i Servizi Associati #
["web_server", "web_server", "web_server"].each |Integer $index, String
    $value| {
        docker::run { "${value}${index}":
                image => "web_server",
                ensure => absent,
                before => Exec["wait"]
        }
}
docker::run { 'web_server_db':
        image => "web_server_db",
        ensure => absent,
        before => Exec["wait"]
}
...

##### LIVELLO_PULIZIA_1 ##### (Non cambiare/cancellare questa riga!)

# Rimuovi Anche le Immagini Finali e Ripulisci i File di Log #
docker::image { 'web_server':
        ensure => absent
}
docker::image { 'web_server_db':
        ensure => absent
}
...

##### LIVELLO_PULIZIA_2 ###### (Non cambiare/cancellare questa riga!)

# Rimuovi Anche le Immagini Sorgente #
docker::image { 'php':
        image_tag => "apache",
        ensure => absent
}
docker::image { 'mysql/mysql-server':
        image_tag => "5.7",
        ensure => absent
}
...
```

Figura 8.18. Parte del file `Programmi/rete_container/source_exit.pp`.

valori è stampata subito sotto). Qui non serve alcun file di configurazione Puppet perché la disinstallazione viene fatta dal tool stesso in modo diverso a seconda del tipo di firewall che si sta rimuovendo.

Per quanto riguarda il caso studio, nel capitolo 4 questo comando è stato utilizzato per disinstallare i tre firewall Cilium (uno per volta) dalla rete di container. La stessa cosa ovviamente può essere fatta con uno degli altri firewall presenti in lista. Per esempio in figura 8.20 è rappresentato il comando da eseguire per disinstallare il firewall Ebtables sul caso studio (ovviamente supponendo di averlo installato in precedenza).

Nel caso si volesse utilizzare questo comando su una rete diversa dal caso studio, non vi è alcuna

```
[nick@host] sudo bash net_manager.sh uninstall

       --- Network Manager ---

Utilizzo: ./net_manager.sh uninstall {ID_FIREWALL}
Per: disinstallare un firewall precedentemente installato su una rete di
    container

    {ID_FIREWALL}:
        1 ebtables (lv 2)
        2 tc (lv 3-4)
        3 iptables (lv 3-4)
        4 cilium (lv 3-4|7|3-4-7)
```

Figura 8.19. Messaggio di spiegazione del comando 'uninstall' del tool `Programmi/rete_container/net_manager.sh`.

```
sudo bash net_manager.sh uninstall 1
```

Figura 8.20. Esecuzione del comando 'uninstall' con firewall Ebtables.

modifica da fare proprio perché non necessita di elaborare alcun file di configurazione Puppet.

8.3.6 Esecuzione dei test per il calcolo delle prestazioni dei diversi firewall

E' possibile effettuare i calcoli sulle prestazioni del caso studio, protetto o meno da uno qualunque dei firewall, eseguendo il comando rappresentato in figura 8.21 (dopo aver eseguito la rete di container e opzionalmente averci installato un firewall) dopo aver modificato le variabili all'inizio del programma stesso rispettivamente per definire qual'è l'indirizzo IP del server che contiene la rete di container, quante richieste fare su ogni pagina del sito web, quante di queste richieste fare in parallelo e il path alla cartella in cui salvare i risultati. Lo script Bash si occuperà automaticamente di creare la cartella dei risultati, se necessario, di lanciare in parallelo un test per ogni pagina del caso studio, di attendere che essi finiscano e di salvare nella cartella dei risultati un file contenente i risultati dei test per ogni pagina richiesta.

8.4 Protezione del programma vulnerabile

Come già anticipato nel capitolo 5, la tesi è stata sviluppata in un sistema Arch Linux a 64 bit, cosa che non fa alcuna differenza nell'esecuzione dei comandi descritti nelle sezioni precedenti, ma potrebbe farne qui. Infatti ciò che viene mostrato in questa sezione in teoria funzionerebbe anche su un sistema diverso, ma come minimo potrebbe essere necessario modificare alcuni parametri dell'exploit creato (il programma che sfrutta le vulnerabilità trovate per mettere in pratica in modo automatico le operazioni spiegate nell'analisi del caso studio, capitolo 5). Per poter utilizzare ciò che è stato implementato in un sistema diverso da Arch Linux senza dover modificare niente, è possibile creare un ambiente Arch all'interno di un container grazie al Dockerfile presente nella cartella `Programmi/programma_target`: innanzitutto bisogna installare Docker (che dovrebbe essere presente nel Packet Manager) e poi eseguire i primi due comandi rappresentati in figura 8.22 dalla

```
bash perf_calculator.sh
```

Figura 8.21. Esecuzione del calcolo delle prestazioni della rete di container protetta o meno da un firewall.

```
sudo docker build -t arch_env .
sudo docker run --rm -it -v ${PWD}:/programma_target arch_env

sudo docker rmi arch_env
sudo docker rmi archlinux/base
```

Figura 8.22. Comandi per la creazione/distruzione dell'ambiente Arch Linux in cui eseguire i comandi successivi.

cartella `Programmi/programma_target` rispettivamente per creare un'immagine Arch Linux completa dei pacchetti necessari per questa sezione e per eseguire l'immagine precedentemente creata. Tutti i comandi descritti successivamente potranno essere eseguiti dalla shell che si ottiene come risultato del secondo comando nello stesso formato descritto (senza dover cambiare path all'interno del container). Una volta terminato con l'ambiente basta inserire il comando 'exit' nella shell e poi rimuovere completamente l'ambiente creato dal sistema principale con gli ultimi due comandi presenti in figura 8.22.

Nel caso in cui si eseguissero i comandi descritti a breve nel sistema principale, assicurarsi che Python sia installato e installare 'gcc' direttamente dal Packet Manager. A questo punto eseguendo il comando rappresentato in figura 8.23 (dalla cartella `Programmi/programma_target`) si può ottenere in risposta la descrizione delle funzionalità dello script che gestisce il programma vulnerabile, protetto o meno da Seccomp, e l'esecuzione dell'exploit. Si può notare che lo script prende in input un primo parametro obbligatorio descritto subito sotto e un secondo parametro opzionale. L'ID del programma definisce quale programma si vuole eseguire: 1 per il programma vulnerabile, 2 per quello protetto da Seccomp direttamente nel codice sorgente del programma stesso e 3 per il programma vulnerabile eseguito da un programma che ne definisce il filtro Seccomp. Il secondo parametro invece rappresenta il comando da fare eseguire all'exploit (dentro al contesto del programma vulnerabile). Quindi nel caso si eseguisse lo script con un solo parametro ci si ritroverà a interagire con il programma desiderato (vulnerabile o protetto in un modo o nell'altro), se invece venisse lanciato con due parametri verrebbe eseguito l'exploit con target il programma richiesto e con comando da eseguire quello passato come secondo parametro. In figura 8.24 sono rappresentati alcuni esempi di esecuzione dello script (da eseguire dalla cartella `Programmi/programma_target`): con il primo comando è possibile interagire con il programma vulnerabile, con il secondo si interagisce con il programma protetto da Seccomp a livello di codice, con il terzo si esegue l'exploit con target il programma vulnerabile per ottenere in risposta l'esecuzione del comando 'whoami' e con l'ultimo si esegue l'exploit con target il programma protetto da Seccomp nel secondo modo per ottenere in risposta l'esecuzione del comando 'uname -a' (il motivo per cui lo spazio va sostituito con la stringa '\${IFS}' è descritto nel capitolo 5).

Infine, è possibile effettuare i calcoli sulle prestazioni del programma vulnerabile, protetto o meno da Seccomp, eseguendo il comando rappresentato in figura 8.25 sempre dalla stessa cartella dopo aver modificato le variabili all'inizio del programma stesso rispettivamente per definire qual'è il target del test ('vulnerable_program.c' per il programma vulnerabile, 'protected_program.c' per il programma protetto da Seccomp a livello di codice o 'support_program.c' per il programma protetto dal programma di supporto) e il numero di cicli di lettura/scrittura da effettuare. Lo script Python si preoccuperà di compilare il target, eseguirlo, comunicare con esso, terminare la comunicazione, ripulire il tutto e stampare a video il tempo impiegato per effettuare il test.

```
[nick@host] bash program_manager.sh

       --- Program Manager ---

Utilizzo: ./program_manager.sh {ID_PROGRAMMA} [CMD_EXPLOIT]

    {ID_PROGRAMMA}:
        1 programma vulnerabile
        2 programma protetto a livello di codice
        3 programma vulnerabile protetto dal programma di supporto
```

Figura 8.23. Messaggio che descrive le funzionalità dello script `Programmi/programma_target/program_manager.sh`.

```
bash program_manager.sh 1
bash program_manager.sh 2
bash program_manager.sh 1 whoami
bash program_manager.sh 3 "uname\${IFS}-a"
```

Figura 8.24. Esempi di esecuzione dello script `Programmi/programma_target/program_manager.sh`.

```
python perf_calculator.py
```

Figura 8.25. Esecuzione del calcolo delle prestazioni del programma vulnerabile, protetto o meno da Seccomp.

Capitolo 9

Guida del programmatore

9.1 Introduzione

In questo capitolo viene mostrato come funziona il programma implementato per la gestione di reti di container e di firewall per la loro protezione introdotto nel capitolo 4. Non verranno affrontati gli argomenti già trattati nella guida dell'utente (capitolo 8), e quindi come eseguire il programma, con quali parametri e qual'è la struttura dei vari file di configurazione Puppet da passare in input la programma, ma verrà descritto come il programma elabora i parametri ricevuti per poter mettere in pratica il suo scopo. Per semplicità, questa guida è stata divisa per argomento: si parlerà innanzitutto di come fa il tool a generare una rete di container, protetta o meno da un firewall, e come fa a distruggerla; come fa a installare un firewall su una rete già in esecuzione e come fa a rimuoverlo; e infine come fa a eseguire file Puppet generici.

9.2 Creazione di una rete di container

Il programma implementato è in grado di costruire una rete di container tramite il comando 'start' che come visto nella guida dell'utente (capitolo 8) necessita in ingresso un file Puppet di tipo 'init' e l'ID del firewall con cui proteggere la rete. Nel caso di ID uguale a 0, la rete viene mandata in esecuzione senza firewall come si può notare dalla figura 9.1: viene innanzitutto fatto un backup delle regole presenti in Ebtables e Iptables prima della creazione della rete; poi viene creato il file `./init.pp` nel quale vengono inserite le due variabili predefinite dal tool ('$pwd' e '$system_path') e viene copiata tutta la prima sezione del file Puppet passato in input; infine viene eseguito il file `./init.pp` appena creato per poi rimuoverlo. Grazie a ciò si può notare il motivo per cui la prima sezione del file Puppet di tipo 'init' debba contenere blocchi Puppet validi (generici e/o di moduli installati, come il modulo Puppet Docker per esempio): quella parte del file viene eseguita direttamente con il comando 'puppet apply' senza alcuna modifica (a parte la definizione delle due variabili predefinite). L'ultima cosa da notare nella stessa figura è che tra la creazione del file `./init.pp` e l'esecuzione del comando 'puppet apply' su di esso, si trovano una serie di blocchi 'if-elif' che controllano l'ID del firewall. Nel caso di ID valido e diverso da 0, si attiverà il blocco di codice riguardante quel dato firewall che servirà a creare la rete di container (prendendo le informazioni necessarie dal file `./init.pp` creato sopra) e a generare un file Puppet contenente le regole del firewall richiesto (tradotte nei comandi da eseguire nel caso di firewall diverso da Iptables) il quale verrà eseguito dal comando 'puppet apply' al fondo della figura. Segue quindi la spiegazione di come i vari firewall vengono gestiti (sempre nel caso di comando 'start').

9.2.1 Installazione del firewall Ebtables

Se si decidesse di creare una rete di container e di proteggerla con il firewall Ebtables (ID firewall uguale a 1) o con la coppia di firewall Ebtables e Modsecurity (ID firewall uguale a 7), si attiverebbe

```
...
# Start #
elif [ "$1" = "start" ]; then

        # Crea un BackUp delle Regole di Filtraggio/Routing Presenti in
            Iptables/Ebtables
        iptables-save > iptables_prestart.bkp
        ebtables-save > ebtables_prestart.bkp

        # Inizializza il File di Configurazione per Puppet
        echo -e "# Inizializza Variabili #" > init.pp
        echo -e "\$pwd = \"$working_dir\"" >> init.pp
        echo -e "\$system_path = \"$PATH\"" >> init.pp
        sed -E '/(##### FIREWALL #####|##### CILIUM)/q' $2 | head -n -1 >>
            init.pp

        # Genera le regole del Firewall Ebtables e Scrivile nel File di
            Configurazione per Puppet (se Necessario)
        if [ "$3" = "1" ]; then
            ...

        # Genera le regole del Firewall TC e Scrivile nel File di
            Configurazione per Puppet (se Necessario)
        elif [ "$3" = "2" ]; then
            ...

        ...

        # Errore
        elif [ "$3" != "0" ]; then
            echo -e "\n[E] Parametri di input errati!\n"
            rm init.pp iptables_prestart.bkp ebtables_prestart.bkp
            exit
        fi

        # Esegui Puppet
        puppet apply init.pp

        # Rimuovi il File di Configurazione Generato (& i Filtri Cilium)
        rm init.pp
        rm -r cilium/cilium_filters 2>/dev/null
...
```

Figura 9.1. Parte del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette l'esecuzione della rete di container senza firewall.

il blocco di codice rappresentato in figura 9.2. Come si può notare, in entrambi i casi vengono fatte le stesse operazioni, a parte la generazione del firewall Modsecurity (la cui relativa funzione verrà spiegata più avanti, nell'apposita sezione): innanzitutto viene eseguita la rete di container tramite il file `./init.pp`, generato nel blocco di codice analizzato in precedenza (per la creazione della rete senza protezione); poi si aspetta per qualche secondo (solo per essere sicuri che l'inizializzazione dei vari container sia andata a buon fine); si chiama una funzione che permette di sostituire nel file Puppet passato in input i nomi dei container usati come sorgente/destinazione delle regole del firewall nei rispettivi indirizzi MAC/IP; si scrivono i comandi necessari per installare le regole

```
...
# Genera le regole del Firewall Ebtables (+ Modsecurity) e Scrivile nel File
    di Configurazione per Puppet (se Necessario)
if [ "$3" = "1" ] || [ "$3" = "7" ]; then
    if [ "$3" = "7" ]; then
        generate_modsecurity_firewall $2
    fi
    puppet apply init.pp
    sleep 10
    translate_firewall_rules $2
    generate_ebtables_firewall $2
    start_logger $3 $2
...
```

Figura 9.2. Parte del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette l'esecuzione della rete di container protetta dai firewall Ebtables o Ebtables + Modsecurity.

definite con Ebtables in un file Puppet; e si avvia il meccanismo di log (se richiesto).

Più nel dettaglio, in figura 9.3 è rappresentata la funzione che traduce i nomi dei container usati per la definizione del firewall nei rispettivi indirizzi MAC/IP. Essa non viene unicamente chiamata nel caso di firewall Ebtables, ma per ovvi motivi viene spiegata una sola volta. Viene innanzitutto fatto un backup del file Puppet passato in input e poi viene modificato l'originale: si vanno a cercare tutti i nomi dei container usati come MAC/IP sorgente/destinazione nelle regole del firewall di livello 2/3-4 (un livello per volta); per ogni nome trovato viene preso l'indirizzo MAC/IP associato a quel dato container e viene inserito quello al posto del nome. A questo punto si il file Puppet passato in input avrà l'intera sezione 'FIREWALL' aggiornata con i correnti indirizzi MAC/IP al posto dei nomi dei container.

Nelle figure 9.4 e 9.5 è invece rappresentata la funzione che interpreta il firewall definito dall'utente e genera i comandi che permettono di installare le regole con Ebtables. Come si può notare viene innanzitutto creato un dizionario contenente la traduzione di tutte le possibili chiavi (definite per il firewall di livello 2) nelle rispettive parole chiave da usare in Ebtables (a parte le chiavi 'action', 'jump' e 'chain' che vengono gestite in modo speciale). Poi si entra in un ciclo che analizza riga per riga la sottosezione della sezione 'FIREWALL' del file Puppet 'init' che contiene il firewall di livello 2 (con i nomi dei container già sostituiti nei rispettivi indirizzi MAC). Più nel dettaglio, se la riga corrente è vuota, contiene un commento o una variabile, la riga viene appesa al file senza alcuna modifica; se la riga contiene l'intestazione di un blocco del firewall viene preso il nome della regola e usato come nome del blocco 'exec' utilizzato dal file Puppet risultante per l'esecuzione di quella data regola, viene inizializzato il comando da eseguire e viene definita l'azione da eseguire come una stringa vuota; se la riga contiene la chiusura di un blocco del firewall viene stampato sul file `./init.pp` il comando generato (il come viene spiegato a breve) dopo averlo chiuso con la relativa azione e, se necessario, viene creata e aggiunta al file anche la regola inversa (se il container1 deve poter parlare con il container2, con Ebtables bisogna per forza creare una regola che permetta anche al container2 di parlare con il container1 altrimenti i pacchetti di risposta verrebbero filtrati); infine se la riga è diversa da tutti i casi precedenti si tratta di una coppia chiave-valore. In quest'ultimo caso viene salvata l'azione nell'apposita variabile, nel caso la chiave fosse 'action' o 'jump', oppure viene inserito nel comando da stampare, al punto giusto, la chain, nel caso la chiave fosse 'chain', oppure viene concatenata al fondo del comando la traduzione della chiave seguita dal valore così com'è. In figura 9.6 è presente un esempio di uno dei blocchi del firewall di livello 2 definiti per il caso studio (prima parte) e del rispettivo blocco, tradotto dal tool, per l'esecuzione di quella stessa regola con Ebtables (seconda parte).

Infine, in figura 9.7 è rappresentata la funzione che gestisce il log dei vari firewall. Essa non viene unicamente chiamata nel caso di firewall Ebtables, ma per ovvi motivi viene spiegata una sola

```
...
# TRANSLATE_FIREWALL_RULES #
function translate_firewall_rules {
    cp $1 $1.bkp

    # Traduci i Nomi dei Container nei Rispettivi Indirizzi MAC nel Firewall
        di Livello 2
    awk '/##### FIREWALL #####/,0' $1.bkp | sed -E "/firewall[ ]?\{/q" - |
        head -n -1 > $1
    names=( $(cat $1 | grep "[^_]source\|[^_]destination" | cut -d '"' -f 2 |
        cut -d "'" -f 2) )
    for name in "${names[@]}"; do
        if [[ $name != *"$"* ]]; then
            mac=$(docker inspect $name | grep -m 1 '"MacAddress"' | cut -d '"'
                -f 4)
            sed -i "s/$name/$mac/g" $1
        fi
    done

    # Traduci i Nomi dei Container nei Rispettivi Indirizzi IP nel Firewall di
        Livello 3-4
    awk '/firewall[ ]?{/,0' $1.bkp | sed '/##### CILIUM #####/q' - | head -n
        -1 >> $1
    names=( $(cat $1 | grep "[^_]source\|[^_]destination" | cut -d '"' -f 2 |
        cut -d "'" -f 2) )
    for name in "${names[@]}"; do
        if [[ $name != *"$"* ]]; then
            ip=$(docker inspect $name 2>/dev/null | grep -m 1 '"IPAddress"' |
                cut -d '"' -f 4)
            if [ "$ip" != "" ]; then
                sed -i "s/$name/$ip/g" $1
            fi
        fi
    done

}
...
```

Figura 9.3. Funzione 'translate_firewall_rules' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di tradurre i nomi dei container usati per la definizione del firewall nei rispettivi indirizzi MAC/IP.

volta. Viene innanzitutto creata la cartella `./logs` e poi nel caso dei firewall Ebtables e Iptables (con o senza Modsecurity) se richiesto dal firewall viene preso il log generato dai tool (una serie di stringhe ricevute come messaggi del kernel) e stampato sul file `./logs/firewall.log`; nel caso invece sia richiesto il log con uno dei firewall Cilium (con o senza Modsecurity) viene analizzata l'apposita 'sezione' del file Puppet di tipo 'init' passato come parametro e poi, per ogni tipo di log richiesto, viene eseguito il comando 'cilium monitor' (con o senza l'opzione di 'verbose') redirigendo l'output (il log appunto) sui file `./logs/firewall_${log_type}.log`. Inoltre, in tutti i casi viene stampato sul file `./logger_kill_support` il PID (o i PID) del processo che sta gestendo il log in background così da poterlo terminare quando non è più necessario (per esempio quando si distrugge la rete creata con il comando 'stop'). Infine, come si può notare, non è gestito il log dei firewall Tc e Modsecurity. Nel primo caso, per il momento il tool non è in grado di gestirne il log; nel secondo caso il log viene fatto automaticamente da Modsecurity, ma all'interno del container in

```
...
# GENERATE_EBTABLES_FIREWALL #
function generate_ebtables_firewall {

    # Definisci le Regole di Traduzione
    declare -A filters
    filters=( ["proto"]="-p" ["source"]="-s" ["destination"]="-d"
        ["ip_source"]="--ip-source" ["ip_destination"]="--ip-destination"
        ["ip_proto"]="--ip-protocol" ["ip_sport"]="--ip-source-port"
        ["ip_dport"]="--ip-destination-port" )

    # Genera le Varie Regole del Firewall e Scrivile nel File di
        Configurazione per Puppet
    echo -e "##### FIREWALL #####\n" > init.pp
    echo -e "\$system_path = \"$PATH\"\n" >> init.pp
    tail -n +2 $1 | sed -E "/firewall[ ]?\{/q" - | head -n -1 | while read
        line; do

        # Commenti, Righe Vuote e Variabili #
        if [ "$line" = "" ] || [[ ${line:0:1} == '#' ]] || [[ ${line:0:1} ==
            '$' ]]; then
            echo $line >> init.pp

        # Intestazione Regole #
        elif [[ "$line" == firewall* ]]; then
            rule_name=$(echo $line | cut -d '"' -f 2 | cut -d "'" -f 2)
            echo -e "exec { '$rule_name':" >> init.pp
            cmd="\tcommand => \"ebtables -A |CHAIN|"
            rule_action=""
...
```

Figura 9.4. Prima parte della funzione 'generate_ebtables_firewall' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di tradurre il firewall definito dall'utente nei comandi necessari all'installazione delle regole richieste con Ebtables.

cui sta lavorando (si ricorda che Modsecurity non è eseguito nel sistema principale, ma dentro a ogni container che lo necessita). Se si volesse avere il log di Modsecurity nel sistema principale si può montare la cartella `./logs` come volume per il container che contiene questo log (e tutti gli altri gestiti da Apache) così come è stato fatto per il caso studio.

9.2.2 Installazione del firewall Tc

Se si decidesse di creare una rete di container e di proteggerla con il firewall Tc (ID firewall uguale a 2) o con la coppia di firewall Tc e Modsecurity (ID firewall uguale a 8), si attiverebbe il blocco di codice rappresentato in figura 9.8. Come si può notare, in entrambi i casi vengono fatte le stesse operazioni, a parte la generazione del firewall Modsecurity (la cui relativa funzione verrà spiegata più avanti, nell'apposita sezione): innanzitutto viene eseguita la rete di container tramite il file `./init.pp`, generato nel blocco di codice analizzato in precedenza (per la creazione della rete senza protezione); poi si aspetta per qualche secondo (solo per essere sicuri che l'inizializzazione dei vari container sia andata a buon fine); si chiama una funzione che permette di sostituire nel file Puppet passato in input i nomi dei container usati come sorgente/destinazione delle regole del firewall nei rispettivi indirizzi MAC/IP (spiegata in precedenza, per il firewall Ebtables); e si scrivono i comandi necessari per installare le regole definite con Tc in un file Puppet.

```
...
        # Chiusura Regole #
        elif [[ ${line:0:1} == '}' ]]; then
            if [ "$rule_action" = "LOG" ]; then
                cmd="$cmd --log --log-ip\""
            else
                cmd="$cmd -j $rule_action\""
            fi
            echo -e "${cmd},\n\tpath => \"\${system_path}\"\n}" >> init.pp
            if [[ "$cmd" == *"-s"* ]] || [[ "$cmd" == *"-d"* ]] || \
                    [[ "$cmd" == *"--ip-source"* ]] || [[ "$cmd" ==
                        *"--ip-destination"* ]]; then
                cmd="${cmd/ -s / -old_s }"
                cmd="${cmd/--ip-source/--old_ip-source}"
                cmd="${cmd/--ip-source-port/--old_ip-source-port}"
                cmd="${cmd/ -d / -s }"
                cmd="${cmd/--ip-destination/--ip-source}"
                cmd="${cmd/--ip-destination-port/--ip-source-port}"
                cmd="${cmd/-old_s/-d}"
                cmd="${cmd/--old_ip-source/--ip-destination}"
                cmd="${cmd/--old_ip-source-port/--ip-destination-port}"
                echo -e "exec { '${rule_name}_opposite':" >> init.pp
                echo -e "${cmd},\n\tpath => \"\${system_path}\"\n}" >> init.pp
            fi

            # Parametri Regole #
            else
                key=$(echo ${line%%=>*})
                value=$(echo ${line##*=>} | cut -d '"' -f 2 | cut -d "'" -f 2)
                if [ "$key" = "action" ] || [ "$key" = "jump" ]; then
                    rule_action=$(echo "$value" | tr a-z A-Z)
                elif [ "$key" = "chain" ]; then
                    cmd="${cmd/|CHAIN|/$value}"
                elif [ "${filters[$key]}" != "" ]; then
                    cmd="$cmd ${filters[$key]} $value"
                fi

            fi
    done

}
...
```

Figura 9.5. Seconda parte della funzione 'generate_ebtables_firewall' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di tradurre il firewall definito dall'utente nei comandi necessari all'installazione delle regole richieste con Ebtables.

Nelle figure 9.9, 9.10 e 9.11 è invece rappresentata la funzione che interpreta il firewall definito dall'utente e genera i comandi che permettono di installare le regole con Tc. Come si può notare viene innanzitutto creato un dizionario contenente la traduzione di tutte le possibili chiavi (definite per il firewall di livello 3) nelle rispettive parole chiave da usare in Tc. Poi si inizializza il file Puppet che conterrà i comandi per l'esecuzione del firewall inserendo subito i comandi necessari per inizializzare Tc sulle interfacce virtuali appartenenti a tutti i container creati in precedenza

```
firewall_l2 { '000 - load_balancer<->web_server0':
    source => "load-balancer",
    destination => "web-server0",
    action => "accept",
    chain => "FORWARD"
}
--------------------------------------------------
exec { '000 - load_balancer<->web_server0':
    command => "ebtables -A FORWARD -s 02:42:ac:11:00:07 -d 02:42:ac:11:00:02
        -j ACCEPT",
    path => "${system_path}"
}
```

Figura 9.6. Esempio di un blocco del firewall di livello 2 e della rispettiva traduzione nel comando Ebtables da eseguire con Puppet.

(questo perché le regole Tc vanno installate su queste interfacce, non sull'interfaccia generica 'Docker0'). Infine si entra in un ciclo che analizza riga per riga l'intera sezione 'FIREWALL' del file Puppet 'init', meno il firewall di livello 7, se presente (con i nomi dei container già sostituiti nei rispettivi indirizzi MAC/IP). Più nel dettaglio, se la riga corrente è vuota, contiene un commento o una variabile, la riga viene appesa al file senza alcuna modifica; se la riga contiene l'intestazione di un blocco del firewall di livello 2 viene settata una variabile che indica che il blocco corrente non va elaborato; se la riga contiene l'intestazione di un blocco del firewall di livello 3 viene preso il nome della regola e usato come nome del blocco 'exec' utilizzato dal file Puppet risultante per l'esecuzione di quella data regola, viene definita l'azione da eseguire come una stringa vuota e viene inizializzato il comando da creare utilizzando l'ID della regola, estratto dal nome del blocco, come valore per indicare la priorità della regola stessa (valori bassi indicano priorità maggiore); se la riga contiene la chiusura di un blocco del firewall viene stampato sul file `./init.pp` il comando generato (il come viene spiegato a breve) dopo averlo chiuso con la relativa azione e, se necessario, viene creata e aggiunta al file anche la regola inversa (se il container1 deve poter parlare con il container2, con Tc bisogna per forza creare una regola che permetta anche al container2 di parlare con il container1 altrimenti i pacchetti di risposta verrebbero filtrati); infine se la riga è diversa da tutti i casi precedenti si tratta di una coppia chiave-valore. In quest'ultimo caso viene salvata l'azione nell'apposita variabile, nel caso la chiave fosse 'action', oppure viene ignorato l'intero blocco, nel caso in cui contenesse una regola di log o una chiave per cui non esiste traduzione, oppure viene concatenata al fondo del comando la traduzione della chiave seguita dal relativo valore. Da notare è il fatto che nella chiusura delle regole, la regola creata (e opzionalmente l'inversa) deve essere installata sia sull'interfaccia del container sorgente che su quella del destinatario; nel caso in cui non fosse specificato né un sorgente né un destinatario, la regola verrebbe installata su tutte le interfacce. In figura 9.12 è presente un esempio di uno dei blocchi del firewall di livello 3 definiti per il caso studio (prima parte) e dei rispettivi blocchi (due per la regola e due per l'inversa), tradotti dal tool, per l'esecuzione di quella stessa regola con Tc (seconda parte).

9.2.3 Installazione del firewall Iptables

Se si decidesse di creare una rete di container e di proteggerla con il firewall Iptables (ID firewall uguale a 3) o con la coppia di firewall Iptables e Modsecurity (ID firewall uguale a 9), si attiverebbe il blocco di codice rappresentato in figura 9.13. Come si può notare, in entrambi i casi vengono fatte le stesse operazioni, a parte la generazione del firewall Modsecurity (la cui relativa funzione verrà spiegata più avanti, nell'apposita sezione): innanzitutto viene eseguita la rete di container tramite il file `./init.pp`, generato nel blocco di codice analizzato in precedenza (per la creazione della rete senza protezione); poi si aspetta per qualche secondo (solo per essere sicuri che l'inizializzazione dei vari container sia andata a buon fine); si chiama una funzione che permette di sostituire nel

```
...
# START_LOGGER #
function start_logger {
    mkdir logs 2>/dev/null

    # Avvia il Log del Firewall Iptables/Ebtables (+ Modsecurity) (se
        Necessario) #
    if [ "$1" = "1" ] || [ "$1" = "3" ] || [ "$1" = "7" ] || [ "$1" = "9" ];
        then
        if [[ $(grep ".*jump.*=>.*LOG.*" $2) != "" ]]; then
            cat /proc/kmsg | stdbuf -oL grep "IN=.*OUT=.*" > logs/firewall.log &
            pid=$!
            (( pid-- ))
            echo $pid > logger_kill_support.tmp
        fi

    # Avvia il Log del Firewall Cilium (+ Modsecurity) (se Necessario) #
    else
        log_types=$(cat $2 | grep "EVENT_TYPE")
        log_types=${log_types#*EVENT_TYPE}
        log_types=${log_types#*:}
        if [[ "$log_types" = *"none"* ]]; then
            return
        fi
        IFS=', ' read -r -a log_types <<< $log_types
        verbose_log=$(cat $2 | grep "VERBOSE_LOG")
        verbose_log=${verbose_log#*VERBOSE_LOG}
        verbose_log=${verbose_log#*:}
        if [[ "$verbose_log" = *"0"* ]]; then
            verbose_log=""
        else
            verbose_log="-v"
        fi
        for log_type in "${log_types[@]}"; do
            cilium monitor -t $log_type $verbose_log >
                logs/firewall_${log_type}.log &
            pid=$!
            echo $pid >> logger_kill_support.tmp
        done
    fi

}
...
```

Figura 9.7. Funzione 'start_logger' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di gestire il log del firewall utilizzato.

file Puppet passato in input i nomi dei container usati come sorgente/destinazione delle regole del firewall nei rispettivi indirizzi MAC/IP; si copiano i blocchi del firewall di livello 3 in un file Puppet (comprese le variabili definite nella sezione 'FIREWALL'); e si avvia il meccanismo di log (se richiesto). A differenza di tutti gli altri casi, come già spiegato in precedenza, qui non è necessario un meccanismo di traduzione delle regole perché Puppet è in grado di gestire direttamente i blocchi scritti per il firewall Iptables (anche per questo motivo non viene mostrato un esempio di un blocco e del rispettivo blocco risultante: l'unica differenza sarebbe avere gli

```
...
# Genera le regole del Firewall TC (+ Modsecurity) e Scrivile nel File di
    Configurazione per Puppet (se Necessario)
elif [ "$3" = "2" ] || [ "$3" = "8" ]; then
    if [ "$3" = "8" ]; then
        generate_modsecurity_firewall $2
    fi
    puppet apply init.pp
    sleep 10
    translate_firewall_rules $2
    generate_tc_firewall $2
...
```

Figura 9.8. Parte del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette l'esecuzione della rete di container protetta dai firewall Tc o Tc + Modsecurity.

indirizzi IP dei container sorgente/destinazione al posto dei nomi).

9.2.4 Installazione del firewall Cilium

Se si decidesse di creare una rete di container e di proteggerla con il firewall Cilium (ID firewall uguale a 4, protezione a livello 3-4, 6, protezione a livello 7, o 11, protezione a livello 3-4-7) o con la coppia di firewall Cilium e Modsecurity (ID firewall uguale a 10), si attiverebbe il blocco di codice rappresentato in figura 9.14. A parte la generazione del firewall Modsecurity (la cui relativa funzione verrà spiegata più avanti, nell'apposita sezione) si può notare che: innanzitutto viene creata una cartella in cui salvare temporaneamente i file JSON generati (le vere e proprie regole del firewall); poi si chiama la funzione che esegue la rete di container e traduce le regole del firewall definito nei suddetti file; si scrivono i comandi necessari per installare le regole create in un file Puppet; e si avvia il meccanismo di log (se richiesto). Per quanto riguarda la funzione 'generate_cilium_firewall' viene chiamata con parametri diversi a seconda dell'ID del firewall da installare: nel caso di ID diverso da 10 e 11 si passa alla funzione direttamente l'ID ricevuto come parametro in ingresso al tool; nel caso di ID uguale a 10 si passa 4 (perché in questo caso si vuole installare il firewall di livello 3-4); nel caso di ID uguale a 11 si chiama due volte la funzione, la prima volta con ID uguale a 6 (per tradurre il firewall di livello 7) e la seconda volta con ID uguale a 11 che verrà interpretato dalla funzione come fosse il 4 (firewall di livello 3), ma con qualche differenza.

Più nel dettaglio, nelle figure 9.15, 9.16, 9.17 e 9.18 è rappresentata parte della funzione che permette di gestire il firewall Cilium. Si può innanzitutto notare che nel caso di comando 'start' viene immediatamente chiamata la funzione 'execute_in_cilium' (spiegata più avanti) che permette di eseguire i vari container nella rete virtuale associata a Cilium; poi viene creato un dizionario per l'associazione dei nomi dei vari container generati nelle rispettive etichette; viene analizzato il firewall definito dall'utente riga per riga per tradurre le regole in file JSON; e infine si rimuovono le regole ridondanti (quelle che hanno stesso sorgente e destinazione, come nel caso delle tre regole del caso studio che permettono la comunicazione tra il 'Load_Balancer' e i tre 'Web_Server', le quali in Cilium risultano uguali visto che i tre server hanno la stessa etichetta). Per quanto riguarda il ciclo, si può notare che nel caso di righe vuote o commenti si copiano le righe così come sono; nel caso di variabili si copiano nel file destinazione solamente se l'ID del firewall è diverso da 11 (perché in quel caso le variabili sono già state copiate, visto che si chiama la funzione con ID 11 solamente dopo averla già chiamata con ID 6) e in ogni caso vengono salvate in un apposito dizionario; nel caso di intestazione di regole di livello 3-4 o 7 vengono generate una serie di stringhe contenenti i pezzi di tutti i possibili file JSON creabili con al posto delle 'variabili' della regola (etichette sorgente/destinazione, protocollo, porta, ...) delle apposite chiavi (|PROTO|, |PORT|, ...); invece nel caso di chiusura del blocco si controlla innanzitutto se esiste già una regola di livello

7 che ricopre ciò che è definito nella corrente regola di livello 3 (questo controllo viene attivato unicamente con ID pari a 11, quando la funzione viene chiamata per la seconda volta) e poi si vanno a analizzare i pezzi del file JSON da creare concatenando quelli che non contengono più parole chiavi da sostituire (la sostituzione è stata fatta in precedenza, il come è spiegato a breve); infine, in tutti gli altri casi, la riga viene trattata come una coppia chiave-valore e quindi si va a prendere il pezzo del file JSON associato a quella data chiave e si sostituisce nel punto giusto la variabile con il valore trovato nella riga. Ricapitolando, il meccanismo di traduzione funziona nel seguente modo: si definiscono una serie di stringhe ognuna delle quali rappresenta un possibile pezzo del file JSON da creare; per ogni coppia chiave-valore definito nel file Puppet ricevuto in input, si prende la chiave, si trova il relativo pezzo del file JSON e si inserisce il valore associato a quella chiave nel punto giusto della stringa (nel caso di firewall di livello 7 il valore viene elaborato in modo da estrarre il filtro definito); una volta analizzate tutte le coppie chiave-valore si vanno a cercare i pezzi del file JSON completati e si concatenano per formare il file JSON. In figura 9.19 è presente un esempio di uno dei blocchi del firewall di livello 3 definiti per il caso studio (prima parte), del rispettivo blocco, tradotto dal tool, per l'esecuzione di quella stessa regola con il Cilium CLI (seconda parte) e del file JSON generato dal tool (terza parte); inoltre è presente un blocco del firewall di livello 7 (quarta parte) e del rispettivo file JSON (quinta parte). Il comando per il caricamento della regola di livello 7 tramite il Cilium CLI non è rappresentato in quanto uguale a quello di livello 3 (a parte per il nome del file JSON).

Nelle figure 9.20 e 9.21 è rappresentata invece la funzione che viene chiamata prima della generazione del firewall Cilium e che serve per inizializzare l'architettura Cilium, creare la rete da dare in gestione al Cilium Daemon e eseguire i vari container. Tutto ciò viene fatto nel seguente modo: viene innanzitutto creato il file `./init.pp` inserendo in esso tutta la sezione 'CILIUM' presente nel file Puppet passato in ingresso; poi viene aggiunta la sezione 'CONTAINERS' modificata in modo tale da avere all'interno di ogni blocco 'docker::run' (utilizzati per l'esecuzione dei container) anche la chiave 'net' con associato come valore la rete creata per Cilium (in questo modo i container verranno installati sulla rete Cilium e non quella di default di Docker); poi viene creato un dizionario che associa tutti i nomi dei container da eseguire ai rispettivi indirizzi IP della rete Docker (il primo container nella rete di default avrà indirizzo pari a 172.17.0.2, il secondo 172.17.0.3, ...); poi si usa questo dizionario per aggiornare tutti gli indirizzi IP statici presenti nei file di configurazione della rete (ricorsivamente a partire dalla cartella corrente!) ai nomi dei container (perché in Cilium gli indirizzi IP saranno diversi da quelli della rete Docker, l'indirizzamento viene fatto tramite il nome del container e quindi nel momento in cui i container verranno eseguiti, dovranno avere dei file di configurazione aggiornati per poter comunicare tra loro); poi si esegue la rete di container con le varie modifiche apportate; infine si sovrascrive il file `./init.pp` con la sezione 'CILIUM ROUTES' andando anche a sostituire nei vari blocchi i nomi dei container con i rispettivi indirizzi IP (perché le route non sono gestite con Cilium e quindi potrebbe essere necessario utilizzare l'indirizzo IP anziché il nome). Il file così creato sarà quello in cui vengono inseriti i comandi per l'installazione del firewall, nel modo descritto in precedenza. Da notare che durante l'associazione dei nomi dei container con i rispettivi indirizzi IP, nel caso di variabili, lo script cerca la definizione della variabile; per questo motivo se la variabile non è direttamente definita all'interno del file, bisogna inserire un commento con la sua definizione (come accennato nella guida dell'utente, capitolo 8).

9.2.5 Installazione del firewall Modsecurity

Se si decidesse di creare una rete di container e di proteggerla con il firewall Modsecurity (ID firewall uguale a 5) verrebbe unicamente chiamata la funzione rappresentata in figura 9.22; come visto prima, la stessa funzione verrebbe chiamata nel caso in cui si decidesse di proteggere la rete creata con una qualunque delle coppie di firewall che comprendono Modsecurity, con l'unica differenza che dopo aver chiamato essa si eseguirebbero le operazioni necessarie per l'installazione dell'altro firewall. La funzione traduce il firewall di livello 7 definito dall'utente nel seguente modo: per ogni riga del firewall di livello 7 presente nel file Puppet passato in ingresso, se la riga contiene il path al file di configurazione di Modsecurity in cui salvare la regola, ne viene salvato il valore (il path appunto); se contiene la regola vera e propria, la regola viene salvata nel file target; altrimenti la riga viene ignorata. In questo modo ogni regola viene salvata nel rispettivo file target nell'ordine di apparizione nel file di input, senza essere elaborata (infatti, come spiegato nella guida dell'utente,

capitolo 8, le regole di livello 7 vanno espresse nel formato Modsecurity e quindi non necessitano alcuna traduzione). Infine non viene inserito alcun comando di caricamento delle regole nel file Puppet da eseguire perché come spiegato Modsecurity agisce direttamente da dentro al container che sta proteggendo, quindi si suppone che il file di configurazione definito come target per una data regola sia in qualche modo associato al container per cui è stata creata quella regola (nel caso studio per esempio, esiste un unico file di configurazione per tutti e tre i 'Web_server' e si trova in una cartella che viene montata come volume a tutti e tre i container contenenti i server; in questo modo una volta modificato il file, la rete di container viene eseguita e nel momento in cui viene mandato in esecuzione Modsecurity all'interno dei container, questo avrà un file di configurazione già aggiornato).

9.3 Distruzione di una rete di container

Il programma implementato è in grado di fermare una rete di container precedentemente creata grazie al comando 'stop' che come descritto nella guida dell'utente (capitolo 8) prende due parametri in input, un file Puppet di tipo 'exit' e il livello di pulizia da effettuare. In figura 9.23 è rappresentato il pezzo di codice che viene eseguito con un comando 'stop' valido: viene creato il file `./exit.pp`, nel quale vengono innanzitutto inserite le variabili predefinite dal tool; poi vengono aggiunte allo stesso file le sezioni necessarie del file Puppet passato in ingresso (solo la prima nel caso di livello di pulizia 0, la prima e la seconda per il livello 1 e tutte e tre per il livello 2); vengono fermati tutti i processi lanciati in background per la gestione del log (se ne esistono); poi viene eseguito il file Puppet creato così da fermare effettivamente la rete e più in generale eseguire tutte le operazioni di pulizia definite nelle sezioni elaborate del file Puppet passato in ingresso; vengono ripristinate le tabelle Ebtables e Iptables; tutti i file di configurazione modificati in precedenza (per esempio nel caso di firewall Cilium) vengono ripristinati alla loro forma originale; e infine vengono ripuliti tutti i file di configurazione per Modsecurity (se ne esistono).

9.4 Installazione di un firewall su una rete di container

Il programma implementato è in grado di installare un firewall per la protezione di una rete di container già in esecuzione tramite il comando 'install' che come visto nella guida dell'utente (capitolo 8) necessita in ingresso un file Puppet di tipo 'init' e l'ID del firewall con cui proteggere la rete. Il codice che gestisce questo comando non viene rappresentato in quanto molto simile a quello che gestisce il comando 'start'. Infatti le uniche differenze tra i due blocchi di codice sono l'avere diversi firewall (il comando 'install' gestisce un sottoinsieme dei firewall gestiti dal comando 'start'), e quindi ID diversi per stessi firewall, e il non dover eseguire la rete di container. Di conseguenza a seconda dell'ID del firewall ricevuto in ingresso, viene chiamato un blocco di codice che non esegue la rete di container, ma installa unicamente il firewall richiesto allo stesso modo del comando 'start'. Per questo motivo non vengono neanche rappresentate le funzioni di gestione dei vari firewall, in quanto sono già state spiegate nella sezione dedicata al comando 'start' (quella riguardante la creazione di una rete di container). L'unica vera differenza si ha nella gestione del firewall Cilium. Infatti in questo caso la funzione 'generate_cilium_firewall' viene chiamata passandogli '1' come ultimo parametro (mentre nel comando 'start' veniva passato 0). Questo parametro è un flag che dice alla funzione se la rete di container è già in esecuzione o meno. All'inizio della suddetta funzione, come visto in precedenza, viene controllato questo flag: nel caso la rete di container non sia attiva (comando 'start') viene chiamata la funzione in grado di creare l'architettura Cilium e di eseguire la rete di container nel modo definito da Cilium; nel caso la rete di container sia già attiva (comando 'install') viene chiamata la funzione che permette di eseguire l'architettura Cilium e di spostare la rete di container dalla rete virtuale in cui si trova alla rete virtuale gestita da Cilium. Questo viene fatto dalla funzione rappresentata nelle figure 9.24 e 9.25: prima viene eseguita l'architettura Cilium come definito nella sezione 'CILIUM' del file passato in input; poi vengono inizializzate tutte le variabili necessarie (le liste dei container e dei servizi da proteggere con Cilium e il path alla cartella contenente la descrizione della rete, dai commenti presenti nella sezione Cilium, spiegati nella guida dell'utente, capitolo 8; il nome della rete virtuale

gestita da Cilium, direttamente dal file Puppet passato in input; e gli indirizzi IP dei vari container da proteggere); poi vengono sovrascritti gli indirizzi IP con i relativi nomi dei container in tutti i file di configurazione della rete (ricorsivamente a partire dalla cartella 'ROOT_PATH'!); vengono staccati tutti i container dalla rete virtuale a cui sono connessi e spostati nella rete gestita da Cilium; vengono riavviati tutti i container per l'associazione con il daemon Cilium, facendo attenzione a riavviare i servizi dopo ai normali container (perché i container contenenti dei servizi potrebbero avere la necessità di essere riavviati solamente dopo che i container da cui dipendono sono tornati in uno stato di esecuzione normale); infine viene creato un nuovo file Puppet contenente la sezione 'CILIUM ROUTES' con i nomi dei container aggiornati ai rispettivi indirizzi IP. Dopo aver eseguito questa funzione il programma prosegue con la generazione del firewall vero e proprio come descritto per il comando 'start'.

9.5 Disinstallazione di un firewall su una rete di container

Il programma implementato è in grado di disinstallare un firewall precedentemente installato con il comando 'install' tramite il comando 'uninstall' che come visto nella guida dell'utente (capitolo 8) necessita unicamente dell'ID del firewall da disinstallare. Il blocco di codice che gestisce questo comando è molto semplice ed è rappresentato in figura 9.26: nel caso di Ebtables vengono ripristinate le sue tabelle a partire dal backup fatto con il comando 'install' prima di modificarle; nel caso di Tc, vengono cancellate tutte le regole associate a tutte le interfacce virtuali dei container; nel caso di Iptables viene fatta la stessa operazione di Ebtables (ma con l'apposito comando Iptables); nel caso di Cilium invece vengono ripristinate le tabelle Iptables, vengono riavviati tutti i container spostati nella rete Cilium per farli tornare al loro stato iniziale e vengono cancellati i container appartenenti all'architettura Cilium (nel caso in cui Cilium fosse stato gestito in modo diverso potrebbe non essere necessario riavviare l'architettura Cilium). Le operazioni omesse dal centro del codice riguardano semplicemente il ripristino dei file modificati durante l'installazione del firewall (ricorsivamente a partire da 'ROOT_PATH') e lo stop del log.

9.6 Esecuzione di comandi di supporto a una rete di container

Il programma implementato è in grado di eseguire comandi generici su una rete di container precedentemente mandata in esecuzione tramite il comando 'exec' che come visto nella guida dell'utente (capitolo 8) necessita unicamente di un generico file Puppet come parametro in ingresso. Il blocco di codice che gestisce questo comando è molto semplice ed è rappresentato in figura 9.27. Come si può notare, l'unica utilità nell'usare questo comando, per il momento, è il poter utilizzare le due variabili predefinite senza doverle dichiarare. Infatti le operazioni fatte sono: creare il file `./exec.pp` inserendo le variabili subito all'inizio e poi copiando l'intero file passato in ingresso; eseguire il file appena creato.

```
# GENERATE_TC_FIREWALL #
function generate_tc_firewall {

    # Definisci le Regole di Traduzione
    declare -A filters
    filters=( ["proto"]="match ip protocol |K| 0xff" ["proto_all"]="match ip
        protocol 0 0x00" ["source"]="match ip src |K|" ["destination"]="match
        ip dst |K|" ["sport"]="match ip sport |K| 0xffff" ["dport"]="match ip
        dport |K| 0xffff" ["action"]="action |K|" )
    declare -A filters_keys
    filters_keys=( ["icmp"]="1" ["tcp"]="6" ["udp"]="17" ["accept"]="pass" )

    # Aggiungi l'Inizializzazione di TC su Tutte le Interfacce Docker al File
        di Configurazione per Puppet
    echo -e "##### FIREWALL #####\n" > init.pp
    echo -e "\$system_path = \"$PATH\"\n" >> init.pp
    ifaces=( $(ip addr | grep veth | cut -d ' ' -f 2 | cut -d '@' -f 1) )
    echo -e "# Inizializza Tc sulle Interfacce di Ogni Container Creato #" >>
        init.pp
    for iface in "${ifaces[@]}"; do
        echo -e "exec { '${iface}_init':\n\tcommand => \"tc qdisc add dev
            $iface ingress\"," >> init.pp
        echo -e "\tpath => \"\${system_path}\"\n}" >> init.pp
    done
    echo "" >> init.pp

    # Genera le Varie Regole del Firewall e Scrivile nel File di
        Configurazione per Puppet
    rule_num=1
    manage_block=0
    sed '/firewall_l7/q' $1 | head -n -2 | while read line; do

        # Commenti, Righe Vuote e Variabili #
        if [ "$line" = "" ] || [[ ${line:0:1} == '#' ]] || [[ ${line:0:1} ==
            '$' ]]; then
            echo $line >> init.pp

        # Intestazione Regole #
        elif [[ "$line" == firewall* ]]; then
            if [[ "$line" == firewall_l2* ]]; then
                manage_block=0
                continue
            fi
            manage_block=1
            action=""
            cmd_name=$(echo $line | cut -d "'" -f 2 | cut -d '"' -f 2)
            rule_id=${cmd_name%% *}
            rule_id=$(( 10#$rule_id ))
            rule_id=$(( rule_id + 1 ))
            cmd="tc filter add dev |I| protocol ip parent ffff: pref $rule_id
                u32"
```

Figura 9.9. Prima parte della funzione 'generate_tc_firewall' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di tradurre il firewall definito dall'utente nei comandi necessari all'installazione delle regole richieste con Tc.

```
...
        # Chiusura Regole #
        elif [[ ${line:0:1} == '}' ]]; then
            if [ $manage_block = "0" ]; then
                continue
            fi
            if [ "$action" != "" ]; then # Aggiunta dell'Azione al Comando
                if [ "${filters_keys[$action]}" = "" ]; then
                    cmd="$cmd action $action"
                else
                    cmd="$cmd action ${filters_keys[$action]}"
                fi
            fi
            n_cycle=1
            while [ $n_cycle -lt 3 ]; do
                i=${cmd##*match ip src} # Filtro su SRC corrente
                if [ "$i" != "$cmd" ]; then
                    i=$(echo $i | cut -d ' ' -f 1)
                    if [[ "$i" == "172.17.0."* ]]; then
                        i=$(( ${i##*.} - 2 ))
                        real_cmd="${cmd/|I|/${ifaces[$i]}}"
                        echo -e "exec { '$cmd_name - tc_$rule_num':\n\tcommand
                            => \"$real_cmd\"," >> init.pp
                        echo -e "\tpath => \"\${system_path}\"\n}" >> init.pp
                        (( rule_num++ ))
                    fi
                fi
                i=${cmd##*match ip dst} # Filtro su DST corrente
                if [ "$i" != "$cmd" ]; then
                    ...
                fi
                cmd="${cmd/match ip src/match ip |S|}"
                cmd="${cmd/match ip dst/match ip src}"
                cmd="${cmd/match ip |S|/match ip dst}"
                (( n_cycle++ ))
            done
            if [ "${cmd##*match ip src}" = "$cmd" ] && [ "${cmd##*match ip
                dst}" = "$cmd" ]; then
                for iface in ${ifaces[@]}; do # Filtro su Tutti i Container
                    real_cmd="${cmd/|I|/$iface}"
                    echo -e "exec { '$cmd_name - tc_$rule_num':\n\tcommand =>
                        \"$real_cmd\"," >> init.pp
                    echo -e "\tpath => \"\${system_path}\"\n}" >> init.pp
                    (( rule_num++ ))
                done
            fi
...
```

Figura 9.10. Seconda parte della funzione 'generate_tc_firewall' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di tradurre il firewall definito dall'utente nei comandi necessari all'installazione delle regole richieste con Tc.

```
...
        # Parametri Regole #
        else
            if [ $manage_block = "0" ]; then
                continue
            fi
            key=$(echo ${line%%=>*})
            value=$(echo ${line##*=>} | cut -d '"' -f 2 | cut -d "'" -f 2)
            if [ "$key" = "action" ]; then
                action="$value"
                continue
            elif [ "$key" = "jump" ] && [ "$value" = "LOG" ]; then
                manage_block=0
                continue
            elif [ "${filters[$key]}" = "" ]; then
                continue
            elif [ "$key" = "proto" ] && [ "$value" = "all" ]; then
                key="proto_all"
            fi
            cmd="$cmd ${filters[$key]}"
            if [ "${filters_keys[$value]}" = "" ]; then
                cmd="${cmd/|K|/$value}"
            else
                cmd="${cmd/|K|/${filters_keys[$value]}}"
            fi
        fi

    done

}
...
```

Figura 9.11. Terza parte della funzione 'generate_tc_firewall' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di tradurre il firewall definito dall'utente nei comandi necessari all'installazione delle regole richieste con Tc.

```
firewall { '000 - load_balancer<->web_server0':
    proto => "tcp",
    source => "load-balancer",
    destination => "web-server0",
    action => "accept",
    chain => "DOCKER"
}
-------------------------------------------------
exec { '000 - load_balancer<->web_server0 - tc_1':
    command => "tc filter add dev vethf73abb7 protocol ip parent ffff: pref 1
        u32 match ip protocol 6 0xff match ip src 172.17.0.7 match ip dst
        172.17.0.2 action pass",
    path => "${system_path}"
}
exec { '000 - load_balancer<->web_server0 - tc_2':
    command => "tc filter add dev vethcfa224e protocol ip parent ffff: pref 1
        u32 match ip protocol 6 0xff match ip src 172.17.0.7 match ip dst
        172.17.0.2 action pass",
    path => "${system_path}"
}
exec { '000 - load_balancer<->web_server0 - tc_3':
    command => "tc filter add dev vethcfa224e protocol ip parent ffff: pref 1
        u32 match ip protocol 6 0xff match ip dst 172.17.0.7 match ip src
        172.17.0.2 action pass",
    path => "${system_path}"
}
exec { '000 - load_balancer<->web_server0 - tc_4':
    command => "tc filter add dev vethf73abb7 protocol ip parent ffff: pref 1
        u32 match ip protocol 6 0xff match ip dst 172.17.0.7 match ip src
        172.17.0.2 action pass",
    path => "${system_path}"
}
```

Figura 9.12. Esempio di un blocco del firewall di livello 3 e della rispettiva traduzione nel comando Tc da eseguire con Puppet.

```
...
# Copia le regole del Firewall Iptables (+ Modsecurity) nel File di
    Configurazione per Puppet (se Necessario)
elif [ "$3" = "3" ] || [ "$3" = "9" ]; then
    if [ "$3" = "9" ]; then
        generate_modsecurity_firewall $2
    fi
    puppet apply init.pp
    sleep 10
    translate_firewall_rules $2
    sed '/firewall_17/q' $2 | head -n -2 > tmp.pp
    cat tmp.pp | grep '\$.*=.*' > init.pp
    awk '/firewall[ ]?{/,0' tmp.pp | head -n -2 >> init.pp
    rm tmp.pp
    start_logger $3 init.pp
...
```

Figura 9.13. Parte del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette l'esecuzione della rete di container protetta dai firewall Iptables o Iptables + Modsecurity.

```
...
# Genera le regole del Firewall Cilium (+ Modsecurity) e Scrivile nel File di
    Configurazione per Puppet (se Necessario)
elif [ "$3" = "4" ] || [ "$3" = "6" ] || [ "$3" = "10" ] || [ "$3" = "11" ];
    then
    mkdir cilium 2>/dev/null
    filters_path="cilium/cilium_filters"
    mkdir $filters_path
    if [ "$3" = "10" ]; then
        generate_modsecurity_firewall $2
        generate_cilium_firewall 4 $2 $filters_path 0
    elif [ "$3" != "11" ]; then
        generate_cilium_firewall $3 $2 $filters_path 0
    else
        generate_cilium_firewall 6 $2 $filters_path 0
        generate_cilium_firewall $3 $2 $filters_path 0
    fi
    echo -e "\n\n\n##### FIREWALL #####\n" >> init.pp
    for rule_name in $filters_path/*; do
        echo -e "exec { '$(basename $rule_name)_import':\n\tcommand =>
            \"cilium policy import \${pwd}/$rule_name\"," >> ini$
        echo -e "\tpath => \"\${system_path}\"\n}" >> init.pp
    done
    start_logger $3 $2
...
```

Figura 9.14. Parte del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette l'esecuzione della rete di container protetta dai firewall Cilium o Cilium + Modsecurity.

```
...
# GENERATE_CILIUM_FIREWALL #
function generate_cilium_firewall {

    # Esegui la Rete di Container senza Firewall
    if [ "$1" != "11" ]; then
        if [ "$4" = "0" ]; then
            execute_in_cilium $2
        else
            move_to_cilium $2
        fi
    fi

    # Genera le Varie Regole del Firewall
    declare -A names_to_labels
    endpoints=( $(cilium endpoint list --no-headers | cut -d ' ' -f 1) )
    docker_ids=( $(docker ps -q) )
    for endpoint_id in "${endpoints[@]}"; do
        endpoint_ip=$(cilium endpoint get $endpoint_id | grep ipv4 | cut -d
            '"' -f 4)
        for docker_id in "${docker_ids[@]}"; do
            if [[ "$(docker inspect $docker_id | grep $endpoint_ip)" != "" ]];
                then
                endpoint_name=$(docker inspect $docker_id | grep -m 1 Name |
                    cut -d '"' -f 4 | cut -d '/' -f 2)
                break
            fi
        done
        endpoint_label=$(cilium endpoint get $endpoint_id | grep -m 1
            "container:" | cut -d ':' -f 2)
        if [ "$endpoint_label" = "" ]; then
            continue
        fi
        names_to_labels+=(["$endpoint_name"]="${endpoint_label::-1}")
    done
...
```

Figura 9.15. Prima parte della funzione 'generate_cilium_firewall' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di tradurre il firewall definito dall'utente nei comandi necessari all'installazione delle regole richieste con Cilium.

```
...
    rule_num=$(ls $3 -1 | wc -l)
    manage_block=0
    declare -A vars
    sed -e '1,/##### FIREWALL #####/d' $2 | sed '/##### CILIUM #####/q' - |
        head -n -1 | while read line; do

        # Commenti e Righe Vuote #
        if [ "$line" = "" ] || [[ ${line:0:1} == '#' ]]; then
            continue

        # Variabili #
        elif [[ ${line:0:1} == '$' ]]; then
            if [ "$1" != "11" ]; then
                echo "$line" >> init.pp
            fi
            name=$(echo ${line%%=*})
            value=$(echo ${line##*=})
            value=$(echo ${value#*\"})
            value=$(echo ${value%\"*})
            vars+=(["\${${name##*$}}"]="$value")

        # Intestazione Regole #
        elif [[ "$line" == firewall* ]]; then
            if ([[ "$line" == firewall_l7* ]] && [ "$1" = "6" ]) || \
                   ([[ "$line" != firewall_l2* ]] && [[ "$line" !=
                       firewall_l7* ]] && [ "$1" != "6" ]); then
                manage_block=1
            else
                continue
            fi
            basic_rule_1="[{\n\t\"labels\": [{\"key\": \"name\", \"value\":
                \"|RULE_NAME|\"}],\n\t\"endpointSelector\": {\"matchLabels\":
                {\"|KEY_DST|\":\"|VALUE_DST|\"}},"
            basic_rule_2_1="\t\"ingress\": [{\n\t\t\"fromEndpoints\":
                [{\"matchLabels\": {\"|KEY_SRC|\":\"|VALUE_SRC|\"}}],"
            ...
            basic_rule_3_1="\t\t\"toPorts\": [{\"ports\": [{\"port\":
                \"|PORT|\", \"protocol\": \"|PROTO|\"}]}]"
            ...
            basic_rule_4="\t}]\n}]"
            label_src=""
            label_dst=""
            rule_name=$(echo $line | cut -d "'" -f 2 | cut -d '"' -f 2)
            (( rule_num++ ))
            basic_rule_1="${basic_rule_1/|RULE_NAME|/$rule_name}"
...
```

Figura 9.16. Seconda parte della funzione 'generate_cilium_firewall' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di tradurre il firewall definito dall'utente nei comandi necessari all'installazione delle regole richieste con Cilium.

```
...
        # Chiusura Regole #
        elif [[ ${line:0:1} == '}' ]]; then
            if [ $manage_block = "1" ] && [ "$1" = "11" ]; then
                source="\\\"${label_src%%=*}\\\":\\\"${label_src##*=}\\\""
                destination="\\\"${label_dst%%=*}\\\":\\\"${label_dst##*=}\\\""
                rules_l7=( $(grep "{\"path\":\|{\"headers\":" $3/* | cut -d ':'
                    -f 1) )
                for rule in "${rules_l7[@]}"; do
                    if [ "$(cat $rule | grep $source)" != "" ] && [ "$(cat
                        $rule | grep $destination)" != "" ]; then
                        (( rule_num-- ))
                        manage_block=0 # Ignora Regola Lv3 se ne Esiste gia' una
                            Equivalente di Lv7
                        break
                    fi
                done
            fi
            if [ $manage_block = "0" ]; then
                continue
            else
                manage_block=0
            fi
            if [[ "$basic_rule_1" != *"|"*"|"* ]]; then
                echo -e ${basic_rule_1} > "$3/rule_$rule_num"
            else
                (( rule_num-- ))
                continue
            fi
            if [[ "$basic_rule_2_1" != *"|"*"|"* ]]; then
                basic_rule_2=${basic_rule_2_1}
            ...
            fi
            if [[ "$basic_rule_3_3" != *"|PATH|"* ]]; then
                echo -e ${basic_rule_2} >> "$3/rule_$rule_num"
                echo -e ${basic_rule_3_3} >> "$3/rule_$rule_num"
            ...
            fi
            echo -e ${basic_rule_4} >> "$3/rule_$rule_num"
...
```

Figura 9.17. Terza parte della funzione 'generate_cilium_firewall' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di tradurre il firewall definito dall'utente nei comandi necessari all'installazione delle regole richieste con Cilium.

```
...
        # Parametri Regole #
        else
            if [ $manage_block = "0" ]; then
                continue
            fi
            key=$(echo ${line%%=>*})
            if [ "$key" = "rule" ]; then
                value=$(echo ${line##*=>})
                value=$(echo ${value#*\"})
                value=$(echo ${value%\"*})
            else
                value=$(echo ${line##*=>} | cut -d '"' -f 2 | cut -d "'" -f 2 |
                    cut -d ',' -f 1)
            fi
            if ([ "$key" = "action" ] && [ "$value" = "drop" ]) || ([ "$key" =
                "jump" ] && [ "$value" = "LOG" ]); then # Azione
                manage_block=0
                continue
            elif [ "$key" = "proto" ]; then # Protocollo (lv 3-4)
                basic_rule_3_1="${basic_rule_3_1/|PROTO|/$value}"
            elif [ "$key" = "dport" ]; then # Porta Destinazione (lv 3-4)
                basic_rule_3_1="${basic_rule_3_1/|PORT|/$value}"
                basic_rule_3_2="${basic_rule_3_2/|PORT|/$value}"
            elif [[ "$key" == *source ]]; then # Sorgente (lv 3-4-7)
                ...
            elif [[ "$key" == *destination ]]; then # Destinazione (lv 3-4-7)
                ...
            elif [ "$key" = "rule" ]; then # Regola Livello 7 (lv 7)
                rule_type=$(echo $value | cut -d ' ' -f 2)
                if [ "$rule_type" = "REQUEST_URI" ]; then
                    uri=$(echo $value | cut -d ' ' -f 4)
                    uri=${uri%\"*}
                    uri=$(echo ${uri//\\/\\\\\\\\\})
                    basic_rule_3_3="${basic_rule_3_3/|PATH|/$uri}"
                elif [[ "$rule_type" == REQUEST_HEADERS* ]]; then
                    ...
                fi
            fi
        fi

    done

    # Elimina le Regole del Firewall Ridondanti
    ...

}
...
```

Figura 9.18. Quarta parte della funzione 'generate_cilium_firewall' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di tradurre il firewall definito dall'utente nei comandi necessari all'installazione delle regole richieste con Cilium.

```
firewall { '000 - load_balancer<->web_server0':
    proto => "tcp",
    source => "load-balancer",
    destination => "web-server0",
    action => "accept",
    chain => "DOCKER"
}
-------------------------------------------------
exec { 'rule_1_import':
    command => "cilium policy import ${pwd}/cilium/cilium_filters/rule_1",
    path => "${system_path}"
}
-------------------------------------------------
[{
    "labels": [{"key": "name", "value": "000 - load_balancer<->web_server0"}],
    "endpointSelector": {"matchLabels": {"id":"web_server"}},
    "ingress": [{
        "fromEndpoints": [{"matchLabels": {"id":"load_balancer"}}]
    }]
}]
-------------------------------------------------
-------------------------------------------------
firewall_l7 { 'public_pages':
    cilium_source => "load-balancer",
    cilium_destination => "web-server0:80",
    rule => "SecRule REQUEST_URI \"@rx
        ^/$|^/index.php$|^/public.php$|^/vulnerable.php(\\?id=)?[0-9]*$\"
        \"id:102,allow\""
}
-------------------------------------------------
[{
    "labels": [{"key": "name", "value": "public_pages"}],
    "endpointSelector": {"matchLabels": {"id":"web_server"}},
    "ingress": [{
        "fromEndpoints": [{"matchLabels": {"id":"load_balancer"}}],
        "toPorts": [{
            "ports": [{"port": "80", "protocol": "TCP"}],
            "rules": {"HTTP": [{"path": "^/$|
                ^/index.php$|^/public.php$|^/vulnerable.php(\\?id=)?[0-9]*$"}]}
        }]
    }]
}]
```

Figura 9.19. Esempio di due blocchi del firewall, uno di livello 3 e l'altro di livello 7, della traduzione nel comando Cilium da eseguire con Puppet del primo e nei rispettivi file JSON generati.

```
...
# EXECUTE_IN_CILIUM #
function execute_in_cilium {

    # Aggiungi la Creazione dell'Architettura Cilium al File di Configurazione
        per Puppet
    cp init.pp init.tmp
    sed '/##### CONTAINERS #####/q' init.tmp | head -n -1 > init.pp
    awk '/##### CILIUM #####/,0' $1 | sed '/##### CILIUM ROUTES #####/q' - |
        head -n -1 >> init.pp
    echo -e "\n\n" >> init.pp
    awk '/##### CONTAINERS #####/,0' init.tmp >> init.pp
    rm init.tmp

    # Aggiungi l'Esecuzione nella Rete Cilium al File di Configurazione per
        Puppet (se Necessario)
    net_name=$(cat init.pp | grep "docker network create")
    net_name=${net_name##* }
    net_name=${net_name%\"*}
    net_name=${net_name%\'*}
    cat init.pp | grep "docker::run" | while read line; do
        sed -i "s/$line/$line\n\tnet => \"$net_name\",/" init.pp
    done

    # Associa i Nomi dei Vari Container agli Indirizzi IP validi per la Rete
        Docker0
    names=( $(grep "name.*=>" init.pp | cut -d '=' -f 2 | cut -d "'" -f 2 |
        cut -d '"' -f 2) )
    ip_id=2
    declare -A ip_dict
    for name in "${names[@]}"; do
        if [[ "$name" == *\${*} ]]; then
            name_values=$(grep "$name.*=" init.pp)
            name_values="${name_values##*=}"
            if [ "$name_values" = "" ]; then
                name=$( echo $name | sed -e 's/{\(.*\)}/\1/')
                name_values=$(grep "$name.*=" init.pp)
                name_values="${name_values##*=}"
            fi
            name_values=( $(echo ${name_values//,/ }) )
            for i in "${!name_values[@]}"; do
                ip_dict+=(["172.17.0.$ip_id"]="${name_values[$i]:1:-1}")
                (( ip_id++ ))
            done
            (( ip_id-- ))
        else
            ip_dict+=(["172.17.0.$ip_id"]="$name")
        fi
        (( ip_id++ ))
    done
...
```

Figura 9.20. Prima parte della funzione 'execute_in_cilium' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di eseguire la rete di container definita nella rete Cilium.

```
...
    # Aggiorna Tutti gli Indirizzi IP dei Container presenti nei Vari File nei
        Rispettivi Hostname (se Necessario)
    files=( $(grep -r --exclude=$(basename "$0") --exclude="*.bkp"
        "172.17.0.*" -m 1 . | cut -d ':' -f 1) )
    for file in "${files[@]}"; do
        cp $file $file.bkp
        for ip in "${!ip_dict[@]}"; do
            sed -i -e "s/$ip/${ip_dict[$ip]}/g" $file
        done
    done

    # Esegui la Rete di Container Senza Firewall
    puppet apply init.pp
    sleep 10

    # Aggiungi le Regole di Routing per Cilium nel File di Configurazione per
        Puppet (se Necessario)
    echo -e "# Inizializza Variabili #" > init.pp
    echo -e "\$pwd = \"$working_dir\"" >> init.pp
    echo -e "\$system_path = \"$PATH\"" >> init.pp
    awk '/##### CILIUM ROUTES #####/,0' $1 >> init.pp
    for name in "${names[@]}"; do
        if [[ "$name" == *\${*}* ]]; then
            name_values=$(grep "$name.*=" $1)
            name_values="${name_values##*=}"
            if [ "$name_values" = "" ]; then
                name=$( echo $name | sed -e 's/{\(.*\)}/\1/')
                name_values=$(grep "$name.*=" $1)
                name_values="${name_values##*=}"
            fi
            name_values=( $(echo ${name_values//,/ }) )
            for i in "${!name_values[@]}"; do
                container_ip=$(docker inspect "${name_values[$i]:1:-1}" | grep
                    -E "\"IPAddress\": \"[0-9].*\"" | cut -d '"' -f 4)
                sed -i -e "s/$name/$container_ip/g" init.pp
            done
        else
            container_ip=$(docker inspect $name | grep -E "\"IPAddress\":
                \"[0-9].*\"" | cut -d '"' -f 4)
            sed -i -e "s/$name/$container_ip/g" init.pp
        fi
    done

}
...
```

Figura 9.21. Seconda parte della funzione 'execute_in_cilium' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di eseguire la rete di container definita nella rete Cilium.

```
...
# GENERATE_MODSECURITY_FIREWALL #
function generate_modsecurity_firewall {

    # Scrivi le Regole del Firewall Modsecurity nei Rispettivi File di
        Configurazione
    target="/dev/null"
    realpath $1 > modsecurity_stop_support.tmp
    sed -e '1,/firewall_l7/d' $1 | sed '/##### CILIUM #####/q' - | head -n -1
        | while read line; do
        key=$(echo ${line%%=>*})
        value=$(echo ${line##*=>})
        value=$(echo ${value#*\"})
        value=$(echo ${value%\"*})
        if [ "$key" = "modsecurity_target" ]; then
            target="${value/\$\{pwd\}\//}"
        elif [ "$key" = "rule" ]; then
            if [[ "$value" == *"\${"*"}"* ]]; then
                var_name=$(echo ${value#*\$\{})
                var_name=$(echo ${var_name%\}*})
                var_value=$(cat $1 | grep "\$$var_name" | cut -d '=' -f 2)
                var_value=$(echo ${var_value#*\"})
                var_value=$(echo ${var_value%\"*})
                value="${value/\$\{$var_name\}/$var_value}"
            fi
            echo $value >> $target
        elif [[ "${line:0:1}" == '}' ]]; then
            target="/dev/null"
        fi
    done
}
...
```

Figura 9.22. Funzione 'generate_modsecurity_firewall' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di gestire il firewall Modsecurity.

```
...
# Stop #
elif [ "$1" = "stop" ]; then

    # Inizializza il File di Configurazione per Puppet
    echo -e "# Inizializza Variabili #" > exit.pp
    echo -e "\$pwd = \"$working_dir\"" >> exit.pp
    echo -e "\$system_path = \"$PATH\"\n" >> exit.pp

    # Genera il File di Configurazione per Puppet
    if [ "$3" = "0" ]; then
        sed '/##### LIVELLO_PULIZIA_1 #####/q' $2 | head -n -1 >> exit.pp
    elif [ "$3" = "1" ]; then
        sed '/##### LIVELLO_PULIZIA_2 #####/q' $2 | head -n -1 >> exit.pp
    elif [ "$3" = "2" ]; then
        cat $2 >> exit.pp
    else
        echo -e "\n[E] Parametri di input errati!\n"
    fi

    # Ferma il Log del Firewall (se Presente) #
    pids=( $(cat logger_kill_support.tmp 2>/dev/null) )
    kill -9 ${pids[@]} 2>/dev/null
    rm logger_kill_support.tmp 2>/dev/null

    # Esegui Puppet
    puppet apply exit.pp

    # Rimuovi il File di Configurazione Generato
    rm exit.pp

    # Ripristina le Regole di Filtraggio/Routing Originali di Iptables/Ebtables
    iptables-restore < iptables_prestart.bkp
    ebtables-restore < ebtables_prestart.bkp
    rm iptables_prestart.bkp ebtables_prestart.bkp

    # Ripristina i file Originali con quelli di Backup (se Necessario)
    ...

    # Ripulisci le Regole di Livello 7 create per Modsecurity
    grep "modsecurity_target" "$(cat modsecurity_stop_support.tmp
        2>/dev/null)" 2>/dev/null | while read target; do
        target=$(echo $target | cut -d '"' -f 2 | cut -d "'" -f 2)
        target="${target/\$\{pwd\}\//}"
        echo "" > $target
    done
    rm modsecurity_stop_support.tmp 2>/dev/null
...
```

Figura 9.23. Parte del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di rimuovere una rete di container precedentemente mandata in esecuzione.

```
...
function move_to_cilium {

    # Inizializza l'Architettura Cilium
    echo -e "\$pwd = \"$working_dir\"" > init.pp
    echo -e "\$system_path = \"$PATH\"\n" >> init.pp
    awk '/##### CILIUM #####/,0' $1 | sed '/##### CILIUM ROUTES #####/q' - |
        head -n -1 >> init.pp
    puppet apply init.pp
    sleep 10
    cat init.pp | grep "ROOT_PATH\|CONTAINERS" > cilium_uninstall_support.tmp

    # Inizializza le Variabili Utili
    containers=$(cat init.pp | grep CONTAINERS)
    containers=${containers#*CONTAINERS}
    containers=${containers#*:}
    IFS=', ' read -r -a containers <<< $containers
    declare -A old_ips
    for container in "${containers[@]}"; do
        old_ip=$(docker inspect $container | grep -E -m 1 "\"IPAddress\":
            \"[0-9].*\"" | cut -d '"' -f 4)
        old_ips+=(["$container"]="$old_ip")
    done
    root_path=$(cat init.pp | grep ROOT_PATH)
    root_path=${root_path#*ROOT_PATH}
    root_path=${root_path#*:}
    services=$(cat init.pp | grep SERVICES)
    services=${services#*SERVICES}
    services=${services#*:}
    IFS=', ' read -r -a services <<< $services
    net_name=$(cat init.pp | grep "docker network create")
    net_name=${net_name##* }
    net_name=${net_name%\"*}
    net_name=${net_name%\'*}

    # Aggiorna Tutti gli Indirizzi IP dei Container presenti nei Vari File nei
        Rispettivi Hostname (se Necessario)
    ...
...
```

Figura 9.24. Prima parte della funzione 'move_to_cilium' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di spostare una rete di container già in esecuzione nella rete virtuale gestita da Cilium.

```
...
    # Sposta i Container nella Rete Cilium
    docker network ls -q | while read net_id; do
        docker inspect $net_id | grep Name | while read line; do
            for container in "${containers[@]}"; do
                if [[ "$line" = *"$container"* ]]; then
                    docker network disconnect $net_id $container
                fi
            done
        done
    done
    for container in "${containers[@]}"; do
        docker network connect $net_name $container
    done

    # Rigenera tutti i Container per l'Associazione al Daemon di Cilium
    for container in "${containers[@]}"; do
        is_service=0
        for service in "${services[@]}"; do
            if [ $container = $service ]; then
                is_service=1
                break
            fi
        done
        if [ $is_service = 0 ]; then
            docker restart $container
        fi
    done
    sleep 30
    for service in "${services[@]}"; do
        docker restart $service
    done

    # Aggiungi le Regole di Routing per Cilium nel File di Configurazione per
        Puppet (se Necessario)
    ...

}
...
```

Figura 9.25. Seconda parte della funzione 'move_to_cilium' del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di spostare una rete di container già in esecuzione nella rete virtuale gestita da Cilium.

```
# Uninstall #
elif [ "$1" = "uninstall" ]; then
    root_path="."

    # Ripulisci le regole del Firewall Ebtables (se Necessario)
    if [ "$2" = "1" ]; then
        error=$(ebtables-restore 2>&1 < ebtables_preinstall.bkp)
        if [ "$?" = "0" ]; then
            echo "[ebtables-restore]: executed successfully"
        else
            echo "[ebtables-restore]: $error"
        fi
        rm ebtables_preinstall.bkp

    # Ripulisci le regole del Firewall TC (se Necessario)
    elif [ "$2" = "2" ]; then
        ifaces=( $(ip addr | grep veth | cut -d ' ' -f 2 | cut -d '@' -f 1) )
        for iface in "${ifaces[@]}"; do
            error=$(tc qdisc del dev $iface ingress 2>&1)
            if [ "$error" = "" ]; then
                echo "[tc del dev $iface]: executed successfully"
            else
                echo "[tc del dev $iface]: $error"
            fi
        done

    # Ripulisci le regole del Firewall Iptables (se Necessario)
    elif [ "$2" = "3" ] || [ "$2" = "4" ]; then
        ...

    ...

    # Ripulisci le regole del Firewall Cilium (se Necessario)
    if [ "$2" = "4" ]; then
        cilium policy delete --all > /dev/null
        containers=$(cat cilium_uninstall_support.tmp | grep CONTAINERS)
        containers=${containers#*CONTAINERS}
        containers=${containers#*:}
        IFS=', ' read -r -a containers <<< $containers
        for container in "${containers[@]}"; do
            systemctl restart docker-$container
        done
        docker stop cilium 2>/dev/null
        docker rm cilium 2>/dev/null
        docker stop cilium-docker-plugin 2>/dev/null
        docker rm cilium-docker-plugin 2>/dev/null
        docker stop cilium-kvstore 2>/dev/null
        docker rm cilium-kvstore 2>/dev/null
        rm cilium_uninstall_support.tmp
    fi
```

Figura 9.26. Parte del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di disinstallare un firewall precedentemente installato con il comando 'install'.

```
...
if [ "$1" = "exec" ]; then

    # Inizializza il File di Configurazione per Puppet
    echo -e "# Inizializza Variabili #" > exec.pp
    echo -e "\$pwd = \"$working_dir\"" >> exec.pp
    echo -e "\$system_path = \"$PATH\"\n" >> exec.pp
    cat $2 >> exec.pp

    # Esegui Puppet
    puppet apply exec.pp

    # Rimuovi il File di Configurazione Generato
    rm exec.pp
...
```

Figura 9.27. Parte del programma `Programmi/rete_container/net_manager.sh` allegato alla tesi che permette di eseguire comandi generici su reti di container precedentemente mandate in esecuzione.

Bibliografia

[1] eBPF - extended Berkeley Packet Filter, http://prototype-kernel.readthedocs.io/en/latest/bpf

[2] Linux Socket Filtering aka Berkeley Packet Filter (BPF), https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/networking/filter.txt

[3] BPF Features by Linux Kernel Version, https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md

[4] BPF Samples, https://github.com/torvalds/linux/tree/master/samples/bpf

[5] BCC Repository, https://github.com/iovisor/bcc

[6] BCC Reference Guide, https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md

[7] Introduction to Linux Traffic Control, http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html

[8] Kernel Probes (Kprobes), https://www.kernel.org/doc/Documentation/kprobes.txt

[9] Using the Linux Kernel Tracepoints, https://www.kernel.org/doc/Documentation/trace/tracepoints.txt

[10] Cilium Repository, https://github.com/cilium/cilium

[11] Cilium's Documentation, http://docs.cilium.io/en/latest

[12] SECure COMPuting with filters, https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt

[13] Puppet 5.4 reference manual, https://puppet.com/docs/puppet/5.4/index.html

[14] Puppet Firewall Module Documentation, https://forge.puppet.com/puppetlabs/firewall

[15] Modsecurity Reference Manual, https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual-%28v2.x%29#SecRule

[16] Puppet Docker Module Documentation, https://forge.puppet.com/puppetlabs/docker