Politecnico di Torino

Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Analisi delle Configurazioni di Sicurezza per Sistemi IDS

Relatori
prof. Riccardo Sisto
prof. Fulvio Valenza
dott. Daniele Bringhenti

Candidato
Riccardo Mazzaroni

Anno Accademico 2024-2025

Sommario

Gli Intrusion Detection Systems (IDS) sono dispositivi, hardware o software, il cui obiettivo è la rilevazione di intrusioni all'interno di computer o reti locali. Sono strumenti che, uniti ad altri dispositivi come i firewall, aiutano a garantire un livello di sicurezza alto all'interno di un sistema. Per poterli utilizzare al meglio, in modo da prevenire attacchi informatici e intrusioni, è necessario studiarne e analizzarne le proprietà, le configurazioni e il funzionamento.
Obiettivo di questa tesi è introdurre un'analisi degli Intrusion Detection Systems in modo che la gestione di questi possa essere, in futuro, implementata all'interno del framework VEREFOO (VErified REFinement and Optimized Orchestrator), che ha come fine quello di automatizzare la configurazione e l'allocazione dei dispositivi di sicurezza in una rete. Per fare questo sono state qui specificate le diverse tipologie di IDS esistenti, poi confrontate tra loro. Questa fase è stata seguita da un'analisi più approfondita di quelli che oggi sono tre diversi modelli di IDS: Snort, Suricata e OSSEC.
Dopo il confronto tra i principali modelli di IDS analizzati, si è proceduto con la costruzione di un modello astratto di IDS, basato su quelli osservati in precedenza, con conseguente creazione di una sua rappresentazione che, attraverso la forma di uno schema, è stata unita alla struttura base su cui si fonda il funzionamento di VEREFOO, poi introdotto in modo da contestualizzare quanto prodotto.
Successivamente è stato illustrato il lavoro svolto per la costruzione di un traduttore che permetta di tradurre la configurazione XML di un IDS, costruita secondo lo schema precedentemente esposto, in configurazioni reali e per la creazione di una classe di parsing che permetta di fornire in input a VEREFOO le configurazioni ottenute tramite l'analisi dell'output generato dall'IDS, in particolare dei log prodotti dallo stesso. In seguito è stato descritto l'ambiente su cui è stata condotta la fase di test dei componenti creati.

Ringraziamenti

Alla mia famiglia.

Indice

Elenco delle figure

Elenco delle tabelle

Listings

Capitolo 1

Introduzione

Con il progresso tecnologico che spinge, ogni giorno di più, verso una necessaria connettività di rete per quello che riguarda ogni aspetto della vita quotidiana, dalle automobili alla domotica per fare alcuni esempi, si diffonde tra le persone l'esigenza di connettersi alle reti in modo sicuro e affidabile.

Per raggiungere un livello di sicurezza adeguato è utile affidarsi a componenti aggiuntivi, fisici e non, quali possono essere i firewall. Per questo nasce VEREFOO (VErified REFinement and Optimized Orchestrator), un framework che può allocare e configurare automaticamente le risorse di sicurezza all'interno di una rete attraverso l'analisi di un grafo creato dagli amministratori della rete.
I firewall non sono l'unico strumento quando si parla di componenti utili a garantire la sicurezza su una rete, un altro è rappresentato dagli Intrusion Detection Systems (IDS).

Obiettivo di questa tesi è lo studio del funzionamento e della configurazione dei vari tipi di IDS tramite un'analisi di quelli che rappresentano lo stato dell'arte, la costruzione di un modello astratto di IDS da aggiungere ai modelli già presenti in VEREFOO per poter in seguito sviluppare un traduttore che permetta di tradurre la configurazione XML di un IDS in configurazioni reali e poi sviluppare una classe di parsing che permetta di passare dall'output generato dall'IDS alle proprietà di input di VEREFOO.

1.1 Struttura del documento

I capitoli successivi sono strutturati nel seguente modo:

- **Capitolo 2**: Studio sugli IDS, sulle diverse tipologie esistenti e sulle differenti modalità in cui operano.
- **Capitolo 3**: Introduzione al framework VEREFOO.
- **Capitolo 4**: Presentazione degli obiettivi della tesi e del contesto in cui è stato svolto il lavoro.

- **Capitolo 5**: Analisi di differenti modelli di IDS.
- **Capitolo 6**: Costruzione del modello astratto di un IDS per iniziare il lavoro di implementazione degli IDS sul framework.
- **Capitolo 7**: Spiegazione del lavoro svolto per la costruzione del traduttore IDS.
- **Capitolo 8**: Descrizione dell'ambiente su cui sono stati svolti i test.
- **Capitolo 9**: Conclusione e lavori futuri.

Capitolo 2

Intrusion Detection System (IDS)

2.1 Tipologie di IDS

Gli Intrusion Detection Systems sono sistemi, hardware o software, in grado di automatizzare il processo di monitoraggio degli eventi che avvengono all'interno di un computer o di una rete, analizzandoli per segnalare eventuali problemi di sicurezza. Questi problemi potrebbero essere tentativi di compromettere la disponibilità, l'integrità o la confidenzialità dei dati o tentativi di superare i meccanismi di sicurezza dei sistemi attaccati. Quando un possibile pericolo viene rilevato, l'IDS genera un alert, un avviso rivolto agli amministratori del sistema.
Esistono tre principali tipi di IDS: Network-Based IDS (NIDS),Host-Based IDS (HIDS),Hybrid IDS (NIDS+HIDS)

- ### 2.1.1 Network-Based IDS (NIDS)

  Sono posizionati sulla rete e, utilizzando diversi sensori, analizzano e monitorano il traffico di rete e sono quindi in grado di segnalare la presenza di attacchi di rete come sniffing, accessi non autorizzati e attacchi Denial-of-Service (DoS). I sensori possono essere in-line o passivi, i primi lasciano che il traffico li attraversi, mentre gli altri monitorano una copia del traffico. Il limite principale dei Network-Based IDS risiede nel fatto che non sono in grado di analizzare il contenuto di pacchetti criptati e potrebbero avere problemi nel gestire grandi quantità di dati.

- ### 2.1.2 Host-Based IDS (HIDS)

  Monitorano l'attività su uno specifico host (endpoint) analizzando i programmi in esecuzione e l'utilizzo delle risorse. Il software che monitora l'host viene anche chiamato agent. Catturando lo stato del sistema in diversi istanti per poter analizzare i cambiamenti, possono segnalare modifiche non autorizzate a file di configurazione, accessi illegittimi alle risorse e tentativi di compromissione dell'integrità dell'host. Sono in grado di analizzare le comunicazioni dell'host che monitorano e possono quindi analizzare il traffico di rete dell'host,

in particolar modo possono avere accesso al contenuto dei pacchetti criptati. Il limite principale degli Host-Based IDS risiede nel fatto che richiedono una notevole quantità di risorse all'host che monitorano.

• 2.1.3 Hybrid IDS (NIDS+HIDS)

Combinano le caratteristiche principali dei NIDS e degli HIDS in modo da poter analizzare lo stato del sistema in un modo più completo, essendo in grado di segnalare la presenza di attacchi di rete e di attacchi all'host. Sono usati per difendersi sia da attacchi che provengono dall'esterno che da attacchi provenienti dall'interno.

Ogni tipologia di IDS può essere trovata sia in formato software sia in formato hardware. Le soluzioni software sono efficienti e più semplici da gestire rispetto a quelle hardware, che richiedono hardware appositi specializzati che, a fronte di prestazioni più elevate rispetto alle loro controparti software, comportano costi maggiori. Il costo elevato è il motivo principale per cui gli hardware IDS sono meno popolari di quelli software, ma sono comunque molto usati in settori specifici (a protezione di data center o server importanti). Esistono poi soluzioni che combinano queste due tipologie di IDS per avere sistemi di monitoraggio semplici come quelli software ma efficienti come quelli hardware.

2.2 Principali metodi di rilevamento

Data la varietà di possibili attacchi che possono colpire un sistema informatico, esistono diverse metodologie usate dagli IDS per rilevarli e avvisare gli amministratori. Le principali sono qui elencate.

• 2.2.1 Signature-Based IDS

Lavorano attraverso l'uso di appositi database contenenti signature di attacchi conosciuti, dove ogni signature è univoca e identifica un particolare attacco. Una signature è l'insieme di alcune delle caratteristiche principali di un certo tipo di attacchi, ad esempio un virus potrebbe lasciare una certa impronta di byte all'interno di un file infetto, questa viene memorizzata nella signature in modo che l'IDS possa generare un alert nel caso in cui trovi qualche corrispondenza. Questa metodologia è particolarmente debole a nuovi tipi di attacchi in quanto l'IDS non sarà in grado di riconoscerli fino a quando le corrispettive signature non saranno inserite nei database. Inoltre, chi sta eseguendo un certo attacco potrebbe essere in grado di offuscare la signature dell'attacco effettuato, rallentando l'intervento dell'IDS. Per questo gli IDS di questo tipo vengono spesso accompagnati da IDS che usano tecniche diverse, come quelli Behavior-Based. Un particolare tipo di Signature-Based IDS sono i Rule-Based IDS, questi IDS sono configurati in modo che possano

usare alcune regole per identificare i pericoli. Con regola si intende la descrizione di un comportamento che si potrebbe verificare durante certi attacchi, qualcosa che li caratterizza. Ad esempio, una regola potrebbe identificare un'elevata quantità di pacchetti ricevuti da un singolo indirizzo IP in modo da segnalare possibili attacchi DoS tramite la generazione di un alert. Questi IDS sono flessibili, per modificarne il comportamento è sufficiente aggiornare o aggiungere regole.

• 2.2.2 Behavior-Based IDS

Questa tecnologia permette di trovare possibili attacchi analizzando il comportamento degli utenti e del sistema. È necessario configurare un punto di partenza per definire i comportamenti corretti in modo che tutto ciò che si discosti da essi possa essere rilevato. Questi IDS sono particolarmente utili per rilevare attacchi che non usano mezzi tecnici ma altri, quali possono essere ad esempio le tecniche di social engineering, si presuppone che questi utenti adottino comportamenti sospetti, che potranno essere rilevati da questi IDS. Un particolare tipo di Behavior-Based IDS sono gli Anomaly-Based IDS, questi IDS, attraverso l'uso di machine learning e analisi statistica, analizzano le deviazioni nel comportamento, in quanto potrebbero indicare la presenza di minacce (ad esempio file modificati senza autorizzazione o scansione delle porte). A differenza degli IDS Signature-Based, questi IDS sono efficaci anche contro attacchi sconosciuti, ma risultano poco efficaci nel caso in cui l'attaccante riesca ad eseguire l'attacco senza modificare il sistema. Inoltre, devono mantenere aggiornata la base che costituisce il normale comportamento del sistema.

2.3 Funzionamento degli IDS

L'obiettivo degli Intrusion Detection Systems è, come già accennato, rilevare un pericolo e avvisare gli amministratori del sistema. Quando un IDS intercetta un pericolo, può generare un alert in base alla configurazione che è stata effettuata. Una classificazione di questa può essere la seguente:

- Alert in tempo reale: Gli IDS inviano gli alert agli amministratori del sistema attraverso canali esterni (come mail o SMS) non appena rilevano un pericolo.
- Alert al superamento di una soglia: Per evitare falsi allarmi, gli amministratori vengono notificati di un certo alert solo quando questo è stato ricevuto un numero di volte superiore ad una certa soglia.
- Prioritizzazione degli alert: è possibile assegnare, ad ogni alert, un certo grado di priorità, in modo che gli amministratori siano notificati solo degli alert ritenuti importanti.

Oltre agli alert, gli IDS dispongono anche di una sezione di reporting. I report possono essere molto dettagliati, fino a contenere tutte le informazioni che l'IDS ha

letto e monitorato. Per questo il loro formato può essere modificato dagli amministratori in modo che contengano solo quello che è ritenuto importante. Alcuni IDS dispongono di sezioni di visualizzazione dati e dashboarding, il tutto per rendere più user-friendly l'analisi dei dati generati dal monitoraggio del sistema.

2.3.1 Falsi positivi e falsi negativi

Uno dei problemi di qualsiasi tipologia di IDS è la generazione di due tipi di errori: falsi positivi e falsi negativi. I falsi positivi si ottengono quando il sistema genera un alert per qualcosa che non è da considerare come pericoloso per il sistema. I falsi negativi si hanno invece quando il sistema non genera un alert per qualcosa che andrebbe considerato come pericoloso per il sistema. I falsi positivi si possono avere in diversi casi:

- Regole o signature configurate male che non permettono all'IDS di riconoscere i pericoli come si vorrebbe.
- Anomalie di rete o del sistema possono causare anomalie anche al corretto funzionamento dell'IDS.

I falsi negativi possono invece essere generati nei seguenti casi:

- Regole o signature non aggiornate.
- Traffico di rete crittografato, in questo modo l'IDS potrebbe avere problemi a rilevare il pericolo.
- Anomalie di rete o del sistema possono causare anomalie anche al corretto funzionamento dell'IDS

È importante ricordare che gli IDS sono sistemi passivi, il loro scopo è quello di segnalare, tramite alert e report, la presenza di pericoli nella rete o nel sistema che monitorano. Non sono in grado di intervenire attivamente per contrastare il pericolo.

Capitolo 3

VEREFOO

3.1 Introduzione a VEREFOO

Le reti moderne diventano ogni giorno più grandi ed eterogenee, composte da componenti diversi tra loro, rendendo difficile la gestione della sicurezza all'interno di esse, in quanto gli errori umani possono comportare gravi problematiche [1].
Appare imminente il passaggio da un tipo di configurazione di sicurezza manuale ad uno automatico [2]. A questa necessità risponde VEREFOO, che è, come indicato precedentemente, un framework progettato per configurare automaticamente le risorse di sicurezza di una rete, quali possono essere i firewall o le VPN.

3.2 Input e Output

Per funzionare correttamente, VEREFOO richiede un Service Graph (SG) e una lista di Requisiti di Sicurezza della rete (NSR) che il grafo dovrebbe soddisfare. In particolare, un Service Graph è una rappresentazione delle relazioni, delle dipendenze e dei flussi di dati che esistono tra i diversi servizi di un sistema distribuito. I requisiti di sicurezza, dovendo indicare la possibilità di bloccare o permettere la comunicazione tra due nodi, sono rappresentati attraverso indirizzi IP, porte e regole.
Fornendo un grafo con la corrispondente lista di NSR, tutto sotto forma di un file XML, il framework elaborerà una corretta configurazione delle risorse che rispetti gli NSR. L'elaborazione del Service Graph porterà alla definizione di un Allocation Graph (AG), una modalità di rappresentazione simile a quella di un Service Graph, ma con più nodi. Sono proprio questi nodi in più a indicare i possibili luoghi di allocazione per le risorse di sicurezza.
Di seguito è fornito un esempio di file XML che rappresenta un Service Graph accettato da VEREFOO. Come si può notare, insieme al grafo, è inclusa la lista di Requisiti di Sicurezza della rete:

Listing 3.1: Esempio di Service Graph accettato da VEREFOO

```
<graphs>
  <graph id="0">
   <node functional_type="WEBCLIENT" name="10.0.0.1">
      <neighbour name="30.0.0.1"/>
      <configuration description="A_simple_description" name="confA">
        <webclient nameWebServer="20.0.0.1"/>
      </configuration>
    </node>
    <node functional_type="FIREWALL" name="30.0.0.1">
      <neighbour name="10.0.0.1"/>
             <neighbour name="20.0.0.1"/>
      <configuration description="A_simple_description" name="conf1">
          <firewall defaultAction="ALLOW">
                    <elements>
                        <action>DENY</action>
                        <source>10.0.0.1</source>
                        <destination>20.0.0.1</destination>
                        <protocol>ANY</protocol>
                        <src_port>*</src_port>
                        <dst_port>*</dst_port>
                    </elements>
                </firewall>
      </configuration>
    </node>
    <node functional_type="WEBSERVER" name="20.0.0.1">
      <neighbour name="30.0.0.1"/>
      <configuration description="A_simple_description" name="confB">
        <webserver>
          <name>b</name>
        </webserver>
      </configuration>
    </node>
  </graph>
</graphs>
<Constraints>
       <NodeConstraints>
       </NodeConstraints>
       <LinkConstraints/>
</Constraints>
<PropertyDefinition>
            <Property graph="0" name="ReachabilityProperty" src="10.0.0.1"
                dst="20.0.0.1"/>
</PropertyDefinition>
```

Nell'esempio sopra si può notare come il grafo sia formato da diversi nodi. Questi sono descritti da alcuni parametri, tra i principali troviamo:

- *functional_type*: indica la tipologia del nodo.
- *name*: indica la l'indirizzo IP del nodo.
- *defaultAction*: indica il modo in cui si dovrebbe comportare il nodo durante l'analisi di traffico non esplicitamente regolato, nell'esempio il traffico verrebbe accettato. L'alternativa ad **ALLOW** è **DENY**. Come si può dedurre dal nome, in questo caso il traffico verrebbe respinto.

Tra i tag più importanti si trovano il tag *neighbour*, che rappresenta la connessione tra due nodi, e il tag *firewall*. Quest'ultimo mostra la rappresentazione nello schema XML di VEREFOO di un firewall.
Nello schema di VEREFOO un firewall è composto da diversi elementi che indicano la configurazione con cui questo dovrebbe essere implementato. Si trovano quindi gli indirizzi IP e le porte di origine e di destinazione del traffico analizzato, il protocollo usato e l'azione che il firewall dovrebbe eseguire su questo traffico. In fondo allo schema XML è possibile osservare i requisiti di sicurezza della rete. Sono compresi tra i tag *PropertyDefinition*. Ogni proprietà è assegnata a un grafo ed è legata a indirizzi IP, campi obbligatori, e porte di origine e di destinazione, campi facoltativi. Quando le porte non sono indicate significa che il range delle porte osservate contiene tutte quelle possibili. Un campo chiave, quando si osservano le proprietà, è il campo *name*, questo può essere uguale a:

- *Reachability Property*: richiede che i due indirizzi siano raggiungibili tra loro.
- *Isolation Property*: richiede che i due indirizzi siano isolati tra loro.

VEREFOO restituirà in output un file di configurazione XML, da utilizzare per l'allocazione automatica delle risorse in rete. Se per qualche motivo non fosse stato possibile rafforzare la sicurezza all'interno del grafo, in output verrà fornito un report, utile per analizzare i problemi trovati.
È importante notare che il file XML ricevuto in output non è direttamente utilizzabile. Per ottenere file di configurazione validi è necessario passare attraverso determinate API offerte dal framework.

3.3 REACT VEREFOO

Per passare da un grafo ad una collezione di configurazioni e corretta allocazione delle risorse e dei servizi, viene usato il problema delle teorie del modulo di massima soddisfacibilità (*maxSMT*) parzialmente pesato. Attraverso la definizione di *hard constraints*, obbligatorie, e *soft constraints*, opzionali e con un peso associato, è possibile, trovando i migliori valori esistenti, cioè quelli che soddisfino tutte le hard constraints e il maggior numero di quelle soft, ottenere la migliore allocazione possibile. Da notare che le constraints vengono definite proprio durante la fase di input del framework.
Attraverso la formulazione e successiva risoluzione di un maxSMT, che riceve in input il grafo e i requisiti di sicurezza della rete, REACT VEREFOO fornisce in output il file di configurazione precedentemente descritto [3]. Per risolvere il problema maxSMT si utilizza un apposito theorem prover open-source, Z3, sviluppato da Microsoft Research.
La versione REACT di VEREFOO (Reactive VErified REFinement and Optimized Orchestration) è caratterizzata dal supporto per la riconfigurazione e riallocazione automatica e dinamica dei firewall. Questi, a differenza di quelli statici, sono implementati su più punti della rete, in modo da garantire un livello di sicurezza più alto e preciso.
Insieme ai principi su cui si fonda VEREFOO (ottimizzazione, automazione e

correttezza formale), la possibilità di implementare firewall dinamici rende questa versione del framework in grado di rispondere a una moltitudine di attacchi informatici.

Capitolo 4

Obiettivo della Tesi

Avendo introdotto, nei capitoli precedenti, gli Intrusion Detection Systems e VEREFOO, questo capitolo è utile per specificare quale sia l'obiettivo di questa tesi. Nel capitolo 3 è stato specificato che l'obiettivo di VEREFOO è quello di allocare funzioni e risorse di sicurezza nella rete e che questa allocazione viene eseguita in base a determinati grafi, rappresentanti la rete, forniti in input. Ma è bene ricordare che al momento VEREFOO supporta un numero limitato di funzioni [4], tra cui vale la pena citare i Firewall [5] e le VPN [6]. L'obiettivo di questa tesi è dare inizio all'implementazione della gestione, della configurazione e dell'allocazione degli IDS all'interno del framework. Di seguito sono elencati i diversi passi che hanno portato verso il raggiungimento dell'obiettivo:

- Studio delle diverse tipologie di IDS e delle differenti funzioni offerte dai vari modelli disponibili. Questo ha portato a un confronto tra alcuni degli IDS più diffusi, necessario per costruire un modello astratto di IDS che potesse essere incluso nello schema che costituisce le regole, i vincoli e gli elementi su cui si fonda il framework. Usando questo schema sono poi stati costruiti dei grafi che rappresentassero reti contenenti i diversi elementi e le diverse funzioni già supportate da VEREFOO, a cui sono stati aggiunti gli IDS. Questo è stato utile per testare le novità introdotte all'interno dello schema.

- La definizione di uno schema per gli IDS ha permesso l'implementazione di un traduttore che, letto un grafo che descrive una rete contenente un IDS, fosse in grado di generare uno script che, se eseguito, potesse configurare le proprietà di sicurezza di un IDS come descritto all'interno del grafo. Il traduttore è stato costruito utilizzando API scritte appositamente, che sono state aggiunte alle tante già offerte dal framework.

- È stata necessaria una fase di test, eseguita su un ambiente apposito che sarà poi descritto in modo da rendere possibile la riproduzione di quanto sopra.

- Infine, dopo aver analizzato le diverse tipologie di log offerte dall'IDS, per permettere l'inserimento in VEREFOO di proprietà ottenute tramite l'estrazione dei log prodotti dall'IDS durante la sua esecuzione, è stata creata una classe di parsing. Attraverso l'estrazione dei log prodotti dall'Intrusion Detection System e utilizzando funzioni scritte appositamente per la loro gestione,

è stata resa possibile la definizione di proprietà all'interno dello schema del framework.

Si può quindi affermare che tra gli obiettivi di questa tesi rientra quello condiviso con altre già esistenti: implementare nuove funzionalità all'interno di VEREFOO.

Capitolo 5

Analisi di differenti modelli di IDS

Vengono ora presentati tre differenti modelli di Software-Based IDS: Snort, Suricata e OSSEC.
L'analisi eseguita sui diversi software si concentra, in particolar modo, sul loro funzionamento e sulle loro possibili configurazioni. Uno dei punti chiave nella scelta di un IDS è la modalità che quest'ultimo utilizza per rilevare le intrusioni e in che modo genera gli avvisi per informare gli amministratori del sistema. Questa analisi è necessaria per poterne valutare i punti di forza e le differenze esistenti.

5.1 Snort

Snort è un Network-Based Intrusion Detection System (NIDS) open-source che usa una tecnologia di tipo Signature-Based, anche ricco di funzionalità che lo rendono un buon packet-sniffer e packet-logger. Quello che è possibile fare con Snort è catturare i pacchetti in rete (packet sniffer), inserirli in appositi container per esaminarli (packet logger) e analizzarli attraverso alcuni specifici criteri per segnalare possibili pericoli (NIDS).
Attraverso l'attivazione di determinati flag durante l'esecuzione di Snort, è possibile eseguire il software in una delle tre possibili modalità:

- **sniffer mode (-v)**: Snort legge i pacchetti sulla rete e stampa i dati letti.
- **packet logger mode (-l)**: Snort logga i pacchetti letti sulla rete in una directory selezionata dagli amministratori del sistema. Questa modalità è più utile della precedente in quanto consente analisi più profonde.
- **network ids mode (-c)**: Snort viene eseguito come IDS, i pacchetti intercettati saranno quindi analizzati e sarà possibile generare alert se dovessero essere trovati riscontri tra le regole.

Snort può operare in due differenti modi:

- **passive**: osservare il traffico su un'interfaccia di rete, senza la possibilità di bloccare il traffico.

- **inline**: per avere la possibilità di bloccare il traffico in caso di necessità, è importante notare che Snort può essere configurato non solo come IDS ma anche come Intrusion Prevention System (IPS).

Snort è composto da otto diversi moduli, configurandoli è possibile abilitare le diverse funzionalità di Snort. Tra i più importanti si possono citare il modulo basic, che gestisce la configurazione di traffico base e l'esecuzione delle regole, e il modulo logger, che coordina gli eventi in output.
Sono presenti librerie di acquisizione dei dati (DAQ) per pacchetti I/O. Queste sostituiscono le chiamate dirette alle funzioni libcap con un layer astratto che facilita le operazioni per garantire la corretta esecuzione di Snort su hardware e software diversi. Snort viene distribuito con una DAQ di default, tuttavia è possibile cambiarla.
Per far sì che il motore di rilevamento delle intrusioni funzioni al meglio, il traffico di rete viene prima analizzato e preparato da alcuni moduli chiamati preprocessori, il cui obiettivo è normalizzare il traffico, decodificare i protocolli per facilitare l'analisi successiva. Questo è necessario in quanto il traffico in rete appena letto potrebbe avere alcune caratteristiche che lo rendono inadatto ad un' ispezione dei pacchetti, come ad esempio la frammentazione. Tra i preprocessori troviamo:

- **Stream**: ricostruisce i flussi di alcuni protocolli (es: TCP).
- **http Inspect**: analizza richieste e risposte http.
- **SSL Preprocessor**: analizza l'handshake SSL/TLS.
- **Portscan**: rileva scansioni di porte.

In modo da garantire un sistema di log completo e preciso, Snort utilizza appositi moduli output, chiamati quando gli alert vengono generati. Grazie a loro, i log possono essere scritti in diversi formati, su file csv, in formato one-line o in server specifici.
Durante la configurazione di Snort è possibile modificare numerose opzioni che riguardano diversi aspetti del software, come i contenuti del logger e la generazione degli alerts. Sono presenti anche impostazioni più specifiche, ad esempio viene offerta la possibilità di scegliere se usare un algoritmo di ricerca "fast pattern" con coda o senza. Con algoritmo di ricerca si intende quello utilizzato per identificare la regola che più si addice al caso analizzato in uno specifico momento da Snort.

5.1.1 Regole Snort

Come già accennato nel paragrafo precedente, Snort, essendo un Signature-Based IDS, utilizza le regole per rilevare possibili pericoli. È possibile trovare set di regole già scritte, ma ogni utente può configurare Snort usando le proprie.
Le regole di Snort si dividono in due sezioni: header e options. Mentre la prima contiene l'azione eseguita dalla regola nel caso in cui trovasse una corrispondenza, i protocolli, gli indirizzi IP e le porte analizzate, la seconda contiene i messaggi mostrati nell'alert generato e altre informazioni riguardanti le modalità di ispezione

dei pacchetti.
L'insieme delle regole può essere considerato come un insieme di elementi separati da OR logici, mentre i singoli componenti della regola sono separati da AND logici.

Listing 5.1: Esempio di regola Snort

```
alert tcp any any -> 192.168.1.0/24 111 (content: |00 01 86 a5|; msg:mountd
    access;)
```

5.1.2 Header

Nell'header vengono definite le azioni compiute dall'IDS nel caso in cui venga trovata una corrispondenza con la regola, il protocollo analizzato, gli indirizzi IP di partenza e di destinazione insieme alle loro porte e la direzione del traffico analizzato.
Esistono sei tipi di azioni configurabili nelle regole:

- **alert**: genera un alert.
- **block**: blocca il traffico.
- **drop**: drop del pacchetto.
- **log**: log del pacchetto.
- **pass**: pacchetto ignorato e segnato come passato.
- **sdrop**: pacchetto bloccato e non loggato.

È inoltre possibile definire nuovi tipi a cui associare azioni formate da quelle già esistenti. Ad esempio, per creare un tipo di azione che scriva su syslog e su un database tcpdump si può scrivere:

Listing 5.2: Esempio di nuovo tipo azione su Snort

```
ruletype redalert
 {
         type alert
         output alert_syslog: LOG_AUTH LOG_ALERT
         output log_tcpdump: suspicious.log
  }
```

Per quanto riguarda i protocolli, quelli supportati da Snort 2.9 sono 4: TCP, UDP, ICMP e IP.
Per gli indirizzi IP la keyword any rappresenta tutti i possibili indirizzi IP, se si desidera invece specificare un indirizzo preciso, è possibile rappresentare la netmask tramite il carattere /.
Le porte possono essere rappresentate tramite la keyword any (tutte le porte), singolarmente (indicando la porta), usando un range di valori (es 1:1024 indica le porte da 1 a 1024, :6000 indica le porte inferiori a 6000, 500: indica le porte superiori a 500).
L'operatore di direzione indica in quale verso il traffico di rete letto è diretto:

- **->**: da sinistra verso destra.
- **<>**: bidirezionale.

5.1.3 Options

Le rule options indicano come verrà analizzato il pacchetto, quale parte e con quali parametri. Sono divise in quattro grandi categorie:

- **General**: forniscono informazioni sulla regola:
  - *sid*: id usato per identificare una regola.
  - *msg*: indica il messaggio da stampare.
  - *priority*: indica il grado di importanza della regola.
- **Payload**: per cercare informazioni tra i dati del pacchetto:
  - *content*: per cercare contenuti del pacchetto specifici. Il contenuto ricercato può essere composto sia da testo che da dati binari (generalmente rappresentati tra due caratteri '|'). Gli unici caratteri che la stringa ricercata non può comprendere sono ';' '\' e '"'. Tramite il carattere '!' è inoltre possibile negare la stringa ricercata. Esistono delle keyword che permettono di modificare il comportamento di content, ad esempio http_client_body restringe la ricerca del content all'interno del solo body del client, un'operazione simile ma sull'header è possibile con http_client_header.
  - *protected_content*: simile a content, ma la stringa ricercata qui viene sostituita dall'hash digest del contenuto ricercato. Qui è quindi necessario anche indicare l'algoritmo usato (MD5, SHA256 o SHA512), l'offset e la lunghezza dei dati.
  - *pcre*: per le regular expression.
  - *base64_decode*: per decodificare dati criptati con base64.
  - *byte_math*: per eseguire operazioni matematiche sui byte letti.
- **Non-Payload**: per cercare informazioni che non riguardano il payload:
  - *ttl*: per controllare il valore del time-to-live.
  - *dsize*: per controllare la grandezza del payload del pacchetto.
  - *flags*: per controllare i valori dei flag tcp.
- **Post-Detection**: per eseguire determinate azioni dopo che una regola è stata eseguita:
  - *logto*: indica un file a cui loggare i pacchetti che hanno causato l'attivazione della regola.
  - *react*: abilita una risposta (una pagina web o altri contenuti) al client prima di chiudere la connessione.

Questi sono solo alcuni esempi delle options utilizzabili durante la definizione di una regola.

5.1.4 Eventi

Gli eventi su Snort avvengono quando una regola trova una corrispondenza e genera un alert. Sono quattro i modi in cui è possibile gestirli:

- **Detection Filters**: viene definita una soglia da superare prima che una regola possa generare un alert.
- **Rate Filters**: quando il numero di corrispondenze trovate supera una certa soglia, che può dipendere da diversi fattori come l'IP sorgente o di destinazione, la regola cambia l'azione eseguita. Ad esempio, una regola che si limitava a loggare i pacchetti, dopo un certo numero di match potrebbe generare un alert.
- **Event Filters**: utili a ridurre il numero di eventi loggati da una certa regola, per avere log più ordinati e puliti, in modo da agevolare l'analisi degli eventi.
- **Event Suppression**: utili a sopprimere completamente certi eventi loggati, la soppressione potrebbe dipendere da diversi fattori quali l'IP sorgente o di destinazione. Il loro obiettivo è simile a quello degli event filters, qui raggiunto tramite una tecnica più brutale.

5.1.5 Configurazione di Snort

Snort è configurabile attraverso l'editing di un file di configurazione chiamato snort.conf.
Qui è possibile editare le variabili usate da Snort durante la sua esecuzione, come ad esempio il range di indirizzi della rete che si intende difendere, settare la scheda di interfaccia di rete (NIC) in modalità promiscua in modo che Snort possa catturare tutto il traffico della rete e non solo quello diretto alla macchina su cui è installato.
La configurazione più importante rimane comunque l'inserimento delle regole definite dall'utente, che devono essere inserite secondo il formato illustrato precedentemente.
Secondo la convenzione le regole vengono scritte in uno o più file esterni, che vengono poi linkati nel file *snort.conf*.

5.2 Suricata

Suricata è, come Snort, un Network-Based Intrusion Detection System (NIDS) open-source che usa una tecnologia di tipo Signature-Based. Il suo funzionamento è molto simile a quello di Snort. Tra le novità introdotte da Suricata [7], però, troviamo:

- **multi-threading**: Suricata è capace di sfruttare al meglio i sistemi multi-core.

- **deep packet inspection (DPI)**: Suricata è in grado di ispezionare i pacchetti fino ai livelli applicativi (HTTP, TLS,FTP,ecc. . .).

Suricata può essere eseguito in diverse modalità, tra cui quelle citate precedentemente per Snort: sniffer mode, packet logger mode e network IDS mode.
Per la cattura dei pacchetti, Suricata utilizza librerie come libpcap, ma supporta anche tecnologie più performanti, in modo da migliorare la scalabilità e il throughput.
Suricata viene distribuito come software da lanciare da riga di comando (per ascoltare una specifica interfaccia si usa il comando: **suricata -i eth0**), più sensori di Suricata possono trovarsi nella stessa rete per generare alert distribuiti (per ascoltare più interfacce in contemporanea è possibile usare il comando **suricata -i eth0 -i eth1**), non è compresa un'interfaccia di lettura degli alert. Le informazioni prodotte da un singolo sensore possono essere inviate e contenute in un server o possono rimanere salvate nel sensore. Esistono comunque tool esterni, sviluppati appositamente per Suricata, che permettono una lettura più agevole per l'utente.
Durante la configurazione di Suricata è possibile modificare numerose opzioni che riguardano diversi aspetti dell'IDS attraverso l'editing di uno specifico file *YAML*, tra queste troviamo:

- Impostazioni di logging dettagliato.
- Ottimizzazioni per l'uso della CPU e della memoria.
- Scelta di algoritmi di pattern-matching.

5.2.1 Regole di Suricata

Una caratteristica particolare di Suricata risiede nella sua capacità di supportare l'uso delle regole scritte su Snort per Snort, questo semplifica la transizione da un sistema all'altro.
Suricata legge le regole una volta sola, all'avvio. Nel caso in cui queste cambiassero o venissero disabilitate, sarebbe necessario riavviare Suricata. Per risolvere questo problema, a partire dalla versione 1.3dev è stata introdotta una nuova funzionalità che prende il nome di Live Rules Swap. Questa funzionalità obbliga Suricata a rileggere le regole durante la sua esecuzione.
Le regole sono divise in tre parti:

- **action**:quale azione viene eseguita dalla regola.
- **header**:definisce protocolli, indirizzi IP, porte e direzione della regola.
- **options**:definisce le specifiche della regola.

Listing 5.3: Esempio di regola Suricata

```
alert tcp any any -> 192.168.1.0/24 111 (content:"|00 01 86 a5|"; msg:"mountd
    access"; sid:1000001; rev:1;)
```

5.2.2 Action

Qui viene definita una delle possibili azioni eseguite da Suricata nel caso in cui la regola generi un match con un pacchetto analizzato. Esistono sette tipi di azioni configurabili:

- **alert**: genera un alert quando trova una corrispondenza.
- **pass**: interrompe l'analisi del pacchetto, viene ignorato.
- **drop**: il pacchetto viene bloccato e un alert viene generato, opzione eseguibile solo se Suricata viene eseguito come Intrusion Prevention System (IPS).
- **reject**: invia un errore RST o ICMP al mittente dei pacchetti per cui viene trovato un match.
- **rejectsrc**: come reject.
- **rejectdst**: invia un errore RST o ICMP al destinatario dei pacchetti per cui viene trovato un match.
- **rejectboth**: invia un errore RST o ICMP sia al mittente che al destinatario dei pacchetti per cui viene trovato un match.

5.2.3 Header

I protocolli disponibili su Suricata sono TCP, UDP, ICMP e IP. Casi particolari di TCP possono essere segnalati con tcp-pkt (per indicare il singolo pacchetto tcp) o con tcp-stream (per indicare lo stream tcp). Sono inoltre disponibili diversi protocolli del livello applicazione, come ad esempio HTTP, FTP, DNS, DHCP, SSH, SMTP, IMAP, POP3, WEBSOCKET, ecc... Questi protocolli risultano essere disponibili solo se abilitati nel file di configurazione di Suricata.
Per indicare il mittente e il destinatario, si utilizzano gli indirizzi IP (IPv4 e IPv6 sono entrambi supportati). È possibile indicare un singolo indirizzo o un range di indirizzi. Di seguito alcuni esempi:

- **!1.1.1.1**: tutti gli indirizzi tranne 1.1.1.1.
- **![1.1.1.1, 1.1.1.2]**: tutti gli indirizzi IP tranne 1.1.1.1 e 1.1.1.2.
- **[10.0.0.0/24, !10.0.0.5]**: 10.0.0.0/24 tranne 10.0.0.5.
- **any**: ogni indirizzo IP.

I valori che seguono gli indirizzi IP indicano le porte. Possono essere rappresentate in modo analogo a quello utilizzato per gli indirizzi IP:

- **[80, 81, 82]**: porte 80,81 e 82.
- **1024:** : tutte le porte a partire dalla 1024.

- **!80**:tutte le porte tranne le 80.
- **[1:80,![2,4]]**:]]: porte da 1 a 80 tranne le porte 2 e 4.
- **any**: ogni porta.

L'operatore di direzione indica in quale verso il traffico di rete è diretto:

- **->**: da sinistra verso destra.
- **<>**: bidirezionale.

5.2.4 Options

Le options sono contenute tra le parentesi e divise da ';'. Alcune options possono avere degli argomenti da impostare, altre invece sono composte solamente dalla keyword che le identifica. L'ordine con cui vengono scritte ha importanza, cambiarlo significa cambiare il significato della regola.
Le keyword delle options si dividono tra *content modifiers*, che posizionano prima le keyword e dopo il buffer, e *sticky buffer*, che posizionano prima il nome del buffer e solo dopo le keyword da applicare ad esso.
Di seguito un esempio di content modifiers: *(content:"index.php"; http_uri; sid:1;)*: il pattern "index.php" è ricercato all'interno del buffer http uri.
Le rule options si dividono in numerose categorie:

- **Meta Keywords**: influiscono solamente sul modo in cui Suricata genera eventi/alerts:
  - *sid*: fornisce un id alla regola, obbligatorio in tutte le regole.
  - *priority*: indica il grado di importanza di una regola, da 1 a 255, dove 1 è il massimo. Regole con una priorità più alta saranno utilizzate prima di quelle con una priorità più bassa.
- **IP Keywords**: riguardano vari aspetti dell'IP:
  - *ttl*: controlla il campo time-to-live nell'header del pacchetto
  - *ipopts*: controlla se specifiche opzioni IP sono settate.
  - *geoip*: controlla il paese di origine di un indirizzo IP.
- **TCP Keywords**: riguardano vari aspetti del protocollo TCP:
  - *tcp.flags*: controlla i valori di specifici flag.
  - *seq*: controlla il valore di sequenza TCP del pacchetto.
  - *ack*: controlla il valore dell' ack TCP.
- **UDP Keyword**: riguardano vari aspetti del protocollo UDP:
  - *udp.hdr*: buffer utilizzato per controllare l'header UDP.

- **ICMP Keyword**: riguardano vari aspetti del protocollo ICMP:
  - *itype*: ICMP ha diversi tipi di messaggi, questi messaggi dipendono da valori numerici su cui viene eseguito un check tramite questa keyword.
  - *icode*: simile alla keyword itype, ma controlla il campo ICMP Code.
  - *icmpv4.hdr e icmpv6.hdr*: controllano rispettivamente gli header di ICMPv4 e ICMPv6.
- **Payload Keyword**: ispezionano il contenuto del payload del pacchetto o dello stream:
  - *content*: il contenuto da trovare all'interno del payload. Possono essere cercati caratteri dalla a alla z, maiuscole, minuscole e caratteri speciali. È inoltre possibile eseguire la ricerca di bytes, scritti tra due caratteri '|'. La ricerca è case sensitive, quindi è necessaria precisione.
  - *nocase*: per disattivare una ricerca di tipo case sensitive, è possibile utilizzare questa keyword.
  - *depth*: quanti byte saranno controllati a partire dall'inizio del payload.
- **Integer Keyword**: per usare i valori interi e controllare diversi valori nel traffico di rete, ad esempio le bitmask.
- **Transformation Keyword**: trasformano i dati negli sticky buffer in qualcos'altro.
  - *compress_whitespace*: comprime tutti gli spazi bianchi consecutivi in un unico spazio bianco.
  - *to_uppercase*: converte il contenuto del buffer in maiuscolo.
  - *from_base64*: il buffer viene decodificato usando valori opzionali per mode, offset e lunghezza del contenuto.
- Esistono molte altre categorie di keywords, il cui titolo rappresenta bene la loro funzione, per citarne alcune: Prefiltering, Flow, HTTP, File, DNS, SSL/-TLS, SSH, DHCP, Base64, NFS, SMTP, WebSocket, Alert, Thresholding.

5.2.5 Configurazione Suricata

Le impostazioni che riguardano i diversi aspetti della configurazione di Suricata sono accessibili attraverso l'editing di un file chiamato *suricata.yaml*. È sempre in questo file che andranno indicati i file contenenti le regole che si intende utilizzare.

5.3 OSSEC

Ossec sta per Open Source HIDS SECurity, a differenza dei due IDS illustrati precedentemente, questo è un Host-Based IDS, scalabile, multipiattaforma e open source.[8]

Essendo multipiattaforma, OSSEC può raccogliere e ispezionare i log presenti su diversi sistemi. L'agente OSSEC, presente sui dispositivi, è il componente che si occupa di raccogliere le informazioni utili e mandarle all'OSSEC manager, il server, che contiene i log, gli eventi, le regole e tutto quello che serve agli agenti per poter lavorare in maniera adeguata. È importante sottolineare che su un dispositivo possono essere presenti più agenti.
Per sistemi che non permettono l'installazione di agenti, esistono componenti agentless, che permettono comunque l'esecuzione di controlli d'integrità. Editando il file *ossec.conf*, è possibile configurare le varie impostazioni che permettono la corretta esecuzione degli agenti OSSEC.
Tra i punti di forza dell'IDS troviamo:

- **File Integrity Checking**: L'agente OSSEC esegue un iniziale scansione di file e directory specifiche, indicate nelle configurazioni dell'IDS, e invia le informazioni di checksum al server OSSEC, che le salva. Se in futuro gli agenti OSSEC dovessero ottenere checksum che non corrispondono a quelli contenuti nel server, gli amministratori del sistema verrebbero allertati. Questo è importante nel caso in cui siano presenti file o directory di cui non è prevista la modifica o la cancellazione.
- **Log Monitoring**: Gli agenti sono in grado di raccogliere qualsiasi tipo di log generato, errori di autenticazione, errori di installazione o altre attività che potrebbero costituire un pericolo per il sistema. I log sono poi analizzati attraverso l'uso di specifiche regole e, se dovessero essere trovate delle corrispondenze, gli amministratori potrebbero ricevere un alert.
- **Rootkit Detection**: Il rootcheck (rootkit detection engine) viene eseguito ogni due ore, gli amministatori hanno la possibilità di cambiare il lasso di tempo che passa da un esecuzione all'altra. Per trovare possibili rootkit installati, l'IDS esegue diverse procedure, elencate nella documentazione ufficiale di OSSEC[8]:
  - esecuzione di diverse system call su un elenco di file spesso usati da software malevoli.
  - scansione di numerose directory per trovare possibili file nascosti o file problematici, come i file con problemi di permessi.
  - ricerca di processi nascosti e scansione dei pid.
  - scansione delle porte, per trovare quelle nascoste, e delle interfacce del sistema per rilevare eventuali anomalie
- **Active Response**: OSSEC contiene al suo interno una serie di script che possono essere eseguiti quando un certo tipo di attacco viene rilevato, in questo modo si può limitare il pericolo. Questa caratteristica è utile soprattutto nei casi in cui gli amministratori non siano immediatamente disponibili.

5.3.1 Regole di OSSEC

L'analisi dei log, attraverso cui OSSEC è in grado di segnalare la presenza di possibili threat, viene eseguita dal logcollector, che li raccoglie, e dall'analysisd, che li

decodifica, filtra e classifica. Questa analisi è eseguita per rilevare gli attacchi alla rete, al sistema o alle applicazioni tramite l'uso dei log, punto chiave delle operazioni di OSSEC. Quello che le regole OSSEC fanno per segnalare tempestivamente possibili attacchi o intrusioni è confrontare i messaggi contenuti nei log con casi predefiniti. Le regole usate da OSSEC si presentano con un formato nettamente differente da quelle usate da Suricata e Snort, sono infatti scritte in formato XML, hanno, quindi, una struttura ad albero.

Listing 5.4: Esempio di regola OSSEC

```
<rule id="100000" level="7">
  <list lookup="match_key" field="srcip">path/to/list/file</list>
  <description>Checking srcip against cdb list file</description>
</rule>
```

Analizzando la regola, è possibile notare come la prima riga venga usata per definire un id e il livello di gravità del caso descritto.
La seconda riga indica la ricerca effettuata dalla regola, una ricerca di tipo match_key che confronterà il valore di srcip con quelli contenuti in path/to/list/file. Questi potrebbero essere indirizzi IP bannati, whitelist o blacklist. La terza riga indica semplicemente una descrizione di quello che è il compito della regola.

I messaggi dei log passano prima dai decoder, la cui funzione è tradurre il messaggio in modo da raccogliere solamente le informazioni più importanti, ovvero quelle da usare. I decoder si possono definire come nel seguente esempio:

Listing 5.5: Esempio di decoder OSSEC

```
<decoder name="widget-transaction">
<parent>widget-processor</parent>
<regex>ID: (\d+) </regex>
<order>id</order>
<accumulate />
</decoder>
```

La prima riga definisce il nome del decoder. Con il tag parent è invece possibile indicare il decoder che precede quello qui definito nell'analisi dei log, il decoder widget-transaction effettuerà le sue analisi in base ai risultati ottenuti dal decoder widget-processor.
Il tag regex è usato per indicare la stringa cercata mentre il tag order assegna un nome alla stringa ricercata tramite la regex, in modo che questo campo possa poi essere usato in una regola (es: *field="id"*). Con il tag accumulate si indica il formato in cui gli eventi verranno stampati sul log. Tutti i match verranno accumulati in un unico evento.

5.3.2 Conclusione analisi OSSEC

Questa analisi di OSSEC è utile per mostrare il modo in cui lavora e funziona un Host IDS, che appare immediatamente diverso da quello di altri IDS di tipo Network, quali possono essere Suricata e Snort.
Si può affermare che OSSEC sia un ottimo prodotto, con grandi potenzialità che

dipendono soprattutto dalla sua configurazione, ma risulta meno adatto a quello che è l'obiettivo di questa tesi rispetto alle sue controparti NIDS. I NIDS sono infatti più adatti a lavorare su una rete per garantirne l'integrità e evitare possibili threats. Una loro implementazione su VEREFOO è quindi più appropriata.

Capitolo 6

Costruzione del modello astratto di un IDS

Per poter avviare l'implementazione degli IDS all'interno di VEREFOO, si rende necessaria la costruzione di un modello astratto di IDS. Da qui nasce l'esigenza di confrontare i due modelli sui quali si baserà quello astratto, in modo da evidenziarne le similitudini e le differenze per una maggiore generalizzazione del modello.

6.1 Confronto Snort Suricata

Avendo analizzato alcuni dei principali IDS, di seguito è presentato un confronto tra Suricata e Snort, i due NIDS la cui configurazione è candidata a essere implementata su VEREFOO.

6.1.1 Similitudini e differenza nella fase di installazione, configurazione e nell'architettura

Entrambi gli IDS richiedono una fase precedente all'installazione del software in cui sarà necessario eseguire il download e quindi installare una libreria utilizzata per catturare il traffico in rete, come *Npcap*.
Mentre su Linux sono disponibili per l'installazione le ultime versioni di entrambi i Network IDS, questo non è vero su Windows per le ultime versioni di Snort.
Analizzando la fase di configurazione si nota che il processo è molto simile tra i due software, entrambi utilizzano un file di configurazione attraverso il quale è possibile impostare le varie funzioni dei due ids, dalle impostazioni di rete alle operazioni di log. Una differenza sostanziale durante questa fase risiede nel diverso formato dei due file, Snort utilizza un file *.conf*, mentre Suricata uno *.yaml*. Quest'ultimo appare più user-friendly e intuitivo.
Entrambi i file permettono di scrivere le regole all'interno di file salvati su directory esterne e poi puntati da questi, in modo da garantire una lettura e una modifica delle regole più agevole.
L'architettura di Snort è single-thread, l'analisi del traffico avviene attraverso l'uso di un solo thread che lavora in modo sequenziale. Questo processo è ottimale in

ambienti con traffico medio o basso, ma limita la scalabilità se il volume dei dati aumenta, in quanto hardware più prestanti non possono essere sfruttati al meglio. Suricata è invece progettato su un'architettura multi-thread e multi-core. Con un conseguente aumento dell'uso delle risorse, l'IDS riesce a distribuire il carico di lavoro in modo da elaborare l'analisi del traffico in parallelo, migliorando le prestazioni soprattutto in ambienti con traffico intenso.

6.1.2 Differenza nelle regole

Suricata supporta l'uso delle regole di Snort, questo non significa però che non ci siano differenze tra le regole utilizzate dai due IDS.
Uno dei punti di forza di Suricata è il rilevamento automatico dei protocolli dell'application layer, tra cui DNS, HTTP, IMAP, POP3. FTP, SSH, TLS, ecc... Il rilevamento dei protocolli, a differenza di Snort, è indipendente dalla porta. Se, ad esempio, su Snort *http_inspect* deve essere configurato su una porta, su Suricata non è necessario, è quindi possibile eseguire un controllo su qualsiasi porta senza preoccuparsi dell'impatto sulle performance che si avrebbe con Snort se si volesse eseguire la stessa operazione.
Suricata esamina il traffico di rete come pacchetti individuali e, nel caso TCP, come parte di uno stream. Esistono quindi keywords che vengono applicate solo all'ispezione dei pacchetti (*dsize*, *ttl*) e altre solo all'ispezione degli stream (*http_**) per garantire il corretto funzionamento di Suricata. Snort invece permette l'uso combinato di queste keywords, sarà quindi possibile usare *dsize* con *http_**. È importante notare, inoltre, che molte keywords presentano leggere differenze nel comportamento tra i due ids, soprattutto nella gestione dei buffer. Su Suricata i buffer vengono spesso normalizzati automaticamente, cosa che non avviene su Snort. Queste non impediscono a Suricata di poter usare rule set scritti per Snort, ma sono il motivo per cui ci si può aspettare leggere differenze nei risultati. Un'altra differenza che è importante osservare risiede nell'utilizzo del fast pattern matcher. Quello di Suricata è, a differenza di quanto avviene su Snort, case sensitive.
Per quanto riguarda gli alert, Snort permette l'impostazione di un numero massimo di alert generati da una regola. Su Suricata questo limite non è configurabile ed è pari a 15.

6.1.3 Conclusione del confronto e scelta dell'IDS da implementare.

Nella tabella sottostante vengono riassunte le principali differenze tra i due software.

Caratteristica	**Snort**	**Suricata**
Architettura e Prestazioni	Single Thread; Ideale per traffico medio - basso; Bassa scalabilità	Multi Thread e Multi Core; Progettato per scalabilità orizzontale; Capacità di gestire il traffico in parallelo
Supporto per protocolli e decodifica	Supporta una vasta gamma di protocolli: TCP, UDP, HTTP, DNS, FTP, SMTP, ecc...	Supporta una vasta gamma di protocolli: TCP, UDP, HTTP, DNS, FTP, SMTP per citarne alcuni ed è in grado di analizzare e decodificare più protocolli contemporaneamente
Regole	Le regole di Snort sono un punto di riferimento per molti IDS; Vasta community che ha prodotto un gran numero di regole pronte da implementare	Supporta le regole di Snort, rendendo agevole la migrazione
Funzionalità aggiuntive	Focalizzato principalmente sul suo compito di IDS,la rilevazione di intrusioni nel sistema; Generazione di log e integrazione con altre soluzioni	Funzionalità avanzate come il rilevamento di pacchetti al livello applicazione. Deep Packet Inspection, File Extraction e Output JSON nativo
Implementazione e documentazione	Molto documentato e supportato da una vasta community	Essendo più giovane di Snort, la documentazione e la community risultano essere più esigue rispetto a quelle di Snort. Si sta comunque diffondendo molto in fretta
Supporto per protocolli emergenti	Aggiornamenti lenti per nuovi protocolli	Maggiore rapidità nell'adozione e supporto di protocolli emergenti
Supporto per analisi forense	Logging dettagliato, ma meno strutturato	Output strutturato (JSON) facilita l'analisi forense automatizzata
Consumo delle risorse	Più leggero, adatto a hardware meno potenti	Maggiore consumo di risorse per le funzionalità avanzate ma maggiore efficienza con multi-core

Tabella 6.1: Confronto tra Snort e Suricata

Per scegliere l'IDS da implementare su VEREFOO sono stati presi in considerazione i punti di forza e di debolezza di Suricata e Snort. Tra questi è importante considerare la scalabilità, superiore quella di Suricata rispetto a quella di Snort grazie alla sua architettura parallela che ne aumenta l'efficienza. Suricata inoltre offre molte funzioni aggiuntive rispetto a quelle offerte da un IDS di base e questo potrebbe facilitare l'implementazione futura di queste nuove funzionalità su VEREFOO.
Suricata è inoltre in grado di generare log in formato JSON, questi log sono formattabili secondo la preferenza degli amministratori del sistema. Questo è un altro

punto di forza di Suricata in quanto l'output generato dal sistema dovrà essere fornito in input al framework e la possibilità di ottenere un file JSON formattato secondo le proprie preferenze facilita il processo. Vale la pena notare che Snort è più diffuso, ha una documentazione completa, è molto stabile e dispone di una grande community che fornisce rule-set già pronti. Queste regole sono però utilizzabili anche da Suricata.
A seguito di queste osservazioni, Suricata risulta essere il software più adatto a un'implementazione su VEREFOO.

6.2 Modello astratto IDS – basato sull'analisi di Snort e Suricata

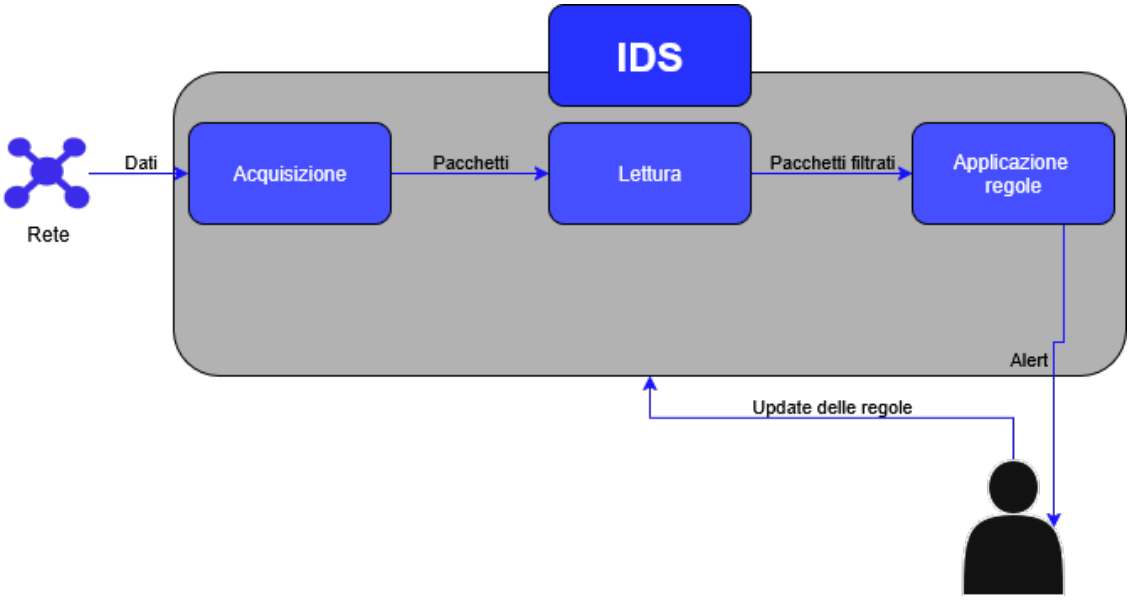

Figura 6.1: Modello astratto di IDS

Per rendere possibile l'implementazione degli IDS su VEREFOO è necessario prima costruire un modello astratto che andrà a costituire, insieme agli altri componenti già presenti, lo schema del framework.

6.2.1 Funzionalità

Le funzionalità dell'IDS possono essere ridotte a quelle elencate di seguito:

1. **Acquisizione dei dati**:
   - Cattura dei pacchetti sulla rete monitorata: Per permettere all'IDS l'analisi del traffico, devono essere usate librerie apposite, come npcap.
   - Filtraggio dei pacchetti secondo i criteri predefiniti, ad esempio indicando gli indirizzi IP e le porte da monitorare.

2. **Lettura del pacchetto**:
   - Lettura dell'header del pacchetto (header IP, TCP o UDP).
   - Lettura del payload del pacchetto e decodifica in base al protocollo usato.
3. **Applicazione delle regole**:
   - Uso delle regole create per rilevare possibili attacchi.
4. **Generazione degli alert e archiviazione per Report**:
   - Se una regola trova una corrispondenza, viene generato un alert.
   - L'alert verrà inviato ad un sistema monitorato da chi di dovere, salvato in un log, in un file o in altri modi impostati durante la configurazione dell'IDS, in questo modo sarà possibile produrre dei report riguardo lo stato del sistema.
5. **Update delle regole**:
   - Per garantire un funzionamento corretto dell'IDS, le regole andranno costantemente aggiornate in modo da prevenire un maggior numero di attacchi.

6.2.2 Regole

Snort e Suricata sono entrambi IDS di tipo Signature-Based e anche le loro regole mostrano una sintassi molto simile. La differenza principale risiede nel fatto che Suricata offre un numero di opzioni più elevato, soprattutto per quanto riguarda la gestione dei protocolli più recenti.
Suricata divide le regole in tre parti: Action, Header e Options. Mentre Snort divide le regole in due parti: Header e Options.
Le regole del modello astratto possono quindi essere viste come regole formate da una sezione header, che corrisponde alla sezione Action più Header di Suricata e alla sezione Header di Snort, e da una sezione options, per la quale non vi sono distinzioni troppo marcate tra i due IDS.
Per quanto riguarda la sintassi della regola, può essere tenuta quella già presente su Snort e Suricata. Un esempio di regola generica del modello astratto può essere quindi il seguente:

Listing 6.1: Sintassi regola generica del modello astratto di IDS

```
alert tcp $rete1 $porte1 -> $rete2 $porte2 (msg: avviso generato dalla
    regola; $OPTIONS)
```

L'header, anche qui, mostra l'azione generata dalla regola. Sia Snort che Suricata offrono come possibili azioni alert, drop e pass. Una differenza importante da notare è l'azione di blocco del traffico che viene chiamata block su Snort. Questa azione non è presente su Suricata, in compenso però esiste un'azione simile chiamata reject che, nel caso di match, oltre a bloccare il traffico genera un errore RST/ICMP che viene inviato al mittente o al destinatario dei pacchetti.

Successivo all'azione, si trova il protocollo analizzato. Snort accetta solamente protocolli di livello trasporto nell'header, quali TCP, UDP, ICMP e IP, mentre Suricata accetta anche protocolli applicativi, come HTTP, FTP, TLS, ecc... In coda all'header si trovano le porte e gli indirizzi IP del mittente e del destinatario da monitorare, insieme alla direzione del traffico. Questa parte non presenta differenze sostanziali tra i due IDS e può quindi essere considerata uguale a come è già stata mostrata precedentemente.
Un caso diverso è la sezione Options, seppure appaia simile nei due NIDS, presenta contenuti piuttosto diversi. Se è vero che Suricata garantisce la compatibilità con le regole di Snort, non è vero il contrario. Come già detto in precedenza, Suricata propone un numero di funzionalità maggiore rispetto a quello offerto da Snort. Inoltre, Suricata richiede in modo obbligatorio la presenza di un id per ogni regola che si intende usare, pena l'esclusione di tale regola dall'esecuzione. Snort non presenta quest'obbligo.

Configurando in modo adeguato le regole durante la fase di configurazione, è possibile intercettare e rilevare i diversi attacchi che Suricata e Snort sono in grado di rilevare:

- **SQL Injection**,**Cross-Site scripting**, **Command Injection**: Sono attacchi il cui obiettivo è inserire codice (SQL o JavaScript) o comandi dannosi e malevoli con lo scopo di ottenere informazioni riservate alterando il normale comportamento del sistema attraverso l'esecuzione di tale codice. Questi attacchi possono essere rilevati filtrando determinate stringhe presenti all'interno dei pacchetti ricevuti.

- **Scansione delle porte**: Attacco che prevede la rilevazione, da parte di un attaccante, delle porte di un sistema, in modo da trovare quali sono aperte o chiuse. Avviene atraverso l'invio di pacchetti alle porte in modo da poterne analizzare le risposte. L'uso del comando threshold, rilevando quanti pacchetti di un certo tipo sono stati inviati dallo stesso IP, può essere utile per la rilevazione di questo attacco.

- Tentativi di accesso non autorizzato (**Brute force** e **Dictionary attacks**): Sono attacchi che, attraverso tentativi sistematici o tramite l'uso di liste chiamate dizionari, provano a indovinare password o chiavi in modo da ottenere informazioni riservate o rovinare l'integrità di un sistema. L'uso del comando threshold, anche qui, può essere utile per la rilevazione di questi attacchi, contando quanti pacchetti di un certo tipo vengono inviati dallo stesso IP in un certo lasso di tempo. Nel caso specifico di HTTP si potrebbe in particolar modo specificare, nella regola, la porta 80 come destination port, quella usata da HTTP.

- **Attacchi DoS e DDoS**: Attacchi Denial of Service e Distributed Denial of Service, hanno lo scopo di rendere un servizio o una rete non disponibile saturando risorse come CPU o memoria. Possono essere rilevati dall'IDS se questo monitora volumi di traffico elevati verso un sistema, in questo caso l'IDS potrebbe generare allarmi quando il traffico supera limiti impostati dagli amministratori.

6.2.3 Implementazione dello schema XSD

Alla conclusione di questa analisi di un NIDS astratto, basato sui modelli offerti da Suricata e Snort, è possibile definire uno schema XSD da integrare a quelli già esistenti su VEREFOO e che ne rappresentano la struttura.
Si procede di seguito alla definizione dell'elemento network_ids:

Listing 6.2: Schema XSD: network IDS

```
<xsd:element name="network_ids">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="nids_rules" minOccurs="0"
maxOccurs="unbounded" />
        </xsd:sequence>
        <xsd:attribute name="defaultAction" type="ActionTypes"/>
    </xsd:complexType>
</xsd:element>
```

network_ids è un elemento complesso in quanto è formato dalla collezione di diversi elementi, le nids_rules. Il passo successivo alla creazione dell'elemento che rappresenta il network ids è la definizione dello schema delle regole, ovvero dell'elemento nids_rules, mostrato qui di seguito:

Listing 6.3: Schema XSD: regola del network IDS

```
<xsd:element name="nids_rules">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="nids_action" type="NIDS_ActionTypes" />
            <xsd:element name="protocol" type="NIDS_ProtocolTypes" />
            <xsd:element name="source_ip" type="xsd:string"/>
            <xsd:element name="source_port" type="xsd:string" />
            <xsd:element name="destination_ip" type="xsd:string" />
            <xsd:element name="destination_port" type="xsd:string" />
            <xsd:element name="direction" type="xsd:string" />
            <xsd:element name="message" type="xsd:string" />
            <xsd:element name="option" type="xsd:string" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Anche l'elemento nids_rules è un elemento complesso, le regole sono infatti formate da diversi elementi:

- **nids_action**:l'azione svolta dall'ids quando un match viene trovato. Questo elemento è di un tipo specifico, definito all'interno dello schema, che verrà mostrato successivamente.
- **protocol**: il protocollo analizzato. Anche questo elemento è di un tipo specifico, definito all'interno dello schema, che verrà mostrato successivamente.
- **source_ip**, **source_port**, **destination_ip**, **destination_port**, **direction**: gli indirizzi IP di origine e di destinazione insieme alle loro porte e la direzione del traffico. Questi elementi sono semplici stringhe.

- **message** e **option**: il messaggio restituito dall'alert e le varie opzioni che è possibile definire. Anche questi elementi sono semplici stringhe. Il numero di possibilità risulta essere pressoché illimitato, definire un tipo apposito perde di significato.

Come menzionato in precedenza, per realizzare in modo adeguato alcuni elementi utilizzati nello schema è stato utile definire due nuovi tipi di elementi, dedicati esclusivamente ai due elementi usati nello schema delle regole del NIDS. NIDS_ActionTypes contiene i possibili valori assegnabili al campo action dell'header delle regole ed è rappresentato attraverso il seguente schema:

Listing 6.4: Schema XSD: tipi di azione della regola

```
<xsd:simpleType name="NIDS_ActionTypes">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="ALERT" />
        <xsd:enumeration value="REJECT" />
        <xsd:enumeration value="DROP" />
        <xsd:enumeration value="PASS" />
    </xsd:restriction>
</xsd:simpleType>
```

Qui sono elencate le quattro possibili scelte.
L'elemento NIDS_ProtocolTypes contiene invece i possibili valori assegnabili al campo protocol presente nell'header delle regole, anche qui le scelte possibili sono quattro e sono rappresentate attraverso il seguente schema:

Listing 6.5: Schema XSD: tipi di protocolli accettati

```
<xsd:simpleType name="NIDS_ProtocolTypes">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="TCP" />
        <xsd:enumeration value="UDP" />
        <xsd:enumeration value="IP" />
        <xsd:enumeration value="ICMP" />
    </xsd:restriction>
</xsd:simpleType>
```

Per concludere la procedura che ha portato alla definizione dello schema di un modello NIDS astratto, è utile poter vedere la rappresentazione di un nodo della rete che rappresenta un NIDS attraverso lo schema appena esposto.

Listing 6.6: Schema XSD: esempio di NIDS

```
<node functional_type="NETWORK_IDS" name="40.0.0.1">
    <neighbour name="30.0.0.1"/>
    <neighbour name="20.0.0.1"/>
    <configuration description="A␣simple␣description" name="confA">
      <network_ids defaultAction="ALLOW">
    <nids_rules>
    <nids_action>ALERT</nids_action>
    <protocol>TCP</protocol>
    <source_ip>10.0.0.1</source_ip>
    <source_port>ANY</source_port>
    <destination_ip>20.0.0.1</destination_ip>
    <destination_port>ANY</destination_port>
```

```
        <direction>-&gt;</direction>
        <message>SQL Injection Attempt</message>
        <option>content:&quot;OR 1=1&quot;; nocase;</option>
      </nids_rules>
      </network_ids>
        </configuration>
    </node>
```

L'elemento di cui sopra rappresenta un network IDS con indirizzo IP *40.0.0.1*. Il nodo si trova tra altri due elementi, rappresentati dagli indirizzi IP *30.0.0.1* e *20.0.0.1*. Il NIDS contiene nella sua configurazione una sola regola, la quale genererà un alert nel caso in cui un pacchetto proveniente da qualsiasi porta della macchina con indirizzo IP *10.0.0.1*, indirizzato verso qualsiasi porta della macchina identificata dall'indirizzo IP *20.0.0.1*, contenga la stringa *";OR 1=1"*. La keyword *nocase* indica una ricerca case-insensitive. Il messaggio generato dall'alert sarà *"SQL Injection Attempt"*. Vale la pena notare che il simbolo >, usato per indicare la direzione del traffico, è qui sostituito da *>*, così come il carattere dell'apice ' è stato sostituito da *"* in quanto entrambi i caratteri risultano problematici se inseriti all'interno dei tag XML.
Si può quindi affermare che questo modello è in grado di rappresentare un NIDS Signature-Based, quali sono Suricata e Snort, il suo funzionamento e la configurazione delle sue regole.

Capitolo 7

Traduttore IDS

7.1 Pattern Model-View-Controller (MVC)

In modo da comprendere in modo chiaro il lavoro svolto, vale la pena soffermarsi brevemente sul pattern MVC, un pattern di progettazione adottato da molti framework, tra cui Spring, usato da VEREFOO. Questa tecnica permette di rendere un'applicazione scalabile e facilmente manutenibile, separando le responsabilità e i doveri all'interno della stessa, dividendola in tre componenti.

- **Controller – Outer Layer**: Gestisce le richieste HTTP, i parametri e il routing. Indirizza le richieste verso le operazioni adeguate.
- **Servizi – Service Layer**: Contiene la logica dell'applicazione, ovvero le funzioni chiamate dal controller.
- **Repository – Data Persistence Layer**: Questo componente opera da intermediario tra l'applicazione e il database. Crea, aggiorna e cancella i dati.

7.2 Traduttore per IDS

In seguito alla definizione dello schema utile a rappresentare i network IDS, ha avuto inizio il lavoro sul traduttore, utile per la configurazione dell'IDS in base al grafo ricevuto in input.
Obiettivo di questo traduttore è la generazione di uno script che, se eseguito, sia in grado di configurare le regole di Suricata. In quanto questo è il modo principale per configurare le proprietà di sicurezza dell'IDS.
Tramite l'uso di JAXB (Java Architecture for XML Binding), una delle API offerte dalla Java Enterprise Edition, che permette di effettuare il mapping tra schemi XML e classi Java, viene garantito l'unmarshalling. In questo modo, a partire dallo schema XML vengono generate le classi NetworkIds, NidsRules, NIDSActionTypes e NIDSProtocolTypes all'interno dell'apposito pacchetto.
La struttura di queste classi rispecchia quanto contenuto nel corrispettivo schema:

- **NIDSProtocolTypes** è una enum che può assumere i seguenti valori: *TCP*, *UDP*, *IP* e *ICMP*.
- **NIDSActionTypes** è una enum che può assumere i seguenti valori: *ALERT*, *REJECT*, *DROP* e *PASS*.
- **NidsRules** è una classe composta da diverse variabili di tipo stringa che rappresentano gli IP sorgente e di destinazione, le porte sorgente e di destinazione, la direzione del traffico, il messaggio generato dall'alert e le option della regola. Si trovano, inoltre, una variabile di tipo NIDSActionTypes e una di tipo NIDSProtocolTypes. Essendo queste variabili protette, sono dichiarati qui anche i metodi getter e setter.
- **NetworkIds** è una classe composta da due parametri: una lista di NidsRules e una variabile di tipo ActionTypes, che rappresenta la defaultAction dello schema.

Queste classi permettono di rappresentare il grafo letto su Java, rendendo possibile una prima implementazione degli IDS su VEREFOO. Inizia da qui, infatti, la creazione del traduttore.

7.2.1 Controller

La generazione dello script di configurazione dell'IDS dovrà essere offerta dal framework tramite API. Per questo, come prima cosa, è stata creata la funzione getNetworkIDS all'interno del controller (DeployerController.java). Questa può essere chiamata con:

Listing 7.1: Esempio chiamata API per traduttore IDS

```
GET BASEURL/verefoo/fwd/deploy/getNetworkIds/{nid}
```

Dove BASEURL è la base URL di VEREFOO, ad esempio http://localhost:8085. La funzione riceve l'id di un nodo del grafo di rete (nid) e invoca una funzione del servizio (FDWService.java) chiamata *getFile*.

7.2.2 Service

Alla funzione getFile vengono passati come parametri il nid e la stringa *nids*. La variabile nid contiene l'id del nodo che rappresenta l'ids contenuto nel grafo di rete originale.
La funzione getFile è la seguente:

Listing 7.2: Service - getFile

```
public File getFile(long nid, String type) {
            File resource = null;
           switch(type)
           {
           case "fortinet":
                  resource = db.getFortinetFirewall(nid);
```

```
                break;
                case "ipfw":
                        resource = db.getIPFWFirewall(nid);
                        break;
                case "iptables":
                        resource = db.getIptablesFirewall(nid);
                        break;
                case "opnvswitch":
                        resource = db.getOpnvswitchFirewall(nid);
                        break;
                case "bpf_iptables":
                        resource = db.getBPFFirewall(nid);
                        break;
                case "nids":
                        resource = db.getNetworkIDS(nid);
                        break;
                default:
                        return null;
                }
        return resource;
}
```

Il file resource viene inizializzato in modo diverso in base alla stringa *type* che la funzione riceve come parametro.
Per implementare fino a qui la gestione degli IDS, è stato sufficiente aggiungere il caso in cui la stringa type fosse uguale a *nids*. Se così sarà, verrà invocata la funzione *getNetworkIDS*, che riceve come unico parametro l'id del nodo ricercato.

7.2.3 Classe per la scrittura del file

Un passaggio la cui esecuzione è necessaria prima della generazione del file di configurazione è la lettura del nodo del grafo salvato nel database. Bisogna quindi creare una classe per contenere i valori del nodo.
La classe creata per questo scopo si chiama NetworkIDS, contiene le seguenti variabili:

- **id**: id del nodo del grafo di rete.
- **node**: nodo del grafo di rete.
- **rules**: regole dell'IDS, contenute nel nodo.
- **filename**: nome del file di configurazione che viene generato.
- **configurationWriter**: usato per scrivere lo script.

Il costruttore richiede come parametri il nodo per cui è richiesta la configurazione di sicurezza IDS e il suo id. Qui viene inizializzato il nome del file che conterrà lo script di configurazione, viene anche eseguito un controllo sul nodo: se questo non dovesse rappresentare un Network IDS, verrebbe lanciata un'eccezione.
Dopo aver settato e inizializzato le variabili, all'interno del costruttore viene eseguita la funzione *getConfigurationFile*, che si occupa della scrittura vera e propria

dello script di configurazione.
Si ricorda che nei network IDS Rule-Based analizzati, le configurazioni di sicurezza si basano su regole precise scritte all'interno di determinati file. Per questo, lo script di configurazione ha l'obiettivo di scrivere su quei file le regole contenute nel nodo del grafo.
Vengono quindi estratte le regole dal nodo e, attraverso uno specifico ciclo, inserite all'interno dello script.
È importante notare che alcuni caratteri non possono essere rappresentati nei file XML, tra questi possiamo trovare i simboli di maggiore (>), che viene scritto come *&gt*, e minore (<), scritto come *&lt*. Viene quindi effettuata anche una sostituzione di questi caratteri, in modo che la regola scritta all'interno del file di configurazione dell'IDS sia comprensibile e corretta.

7.2.4 Data Layer

Dopo aver illustrato il funzionamento di questa classe, è importante citare il punto in cui viene utilizzata. La funzione getNetworkIDS, chiamata all'interno del service layer, è contenuta all'interno della classe FWDDatabase. Viene mostrata qui:

Listing 7.3: Data Layer - getNetworkIDS

```
    public File getNetworkIDS(long nid) {
                Node n = nodes.get(nid);
                if (n == null)
                        return null;
                File file=null;
                try {
                        NetworkIDS nids= new NetworkIDS(nid,n);

                        file = new File(nids.getFilename());

                } catch (Exception e) {
                return null;
                }
          return file;
  }
```

Dopo aver estratto il nodo con id uguale a *nid*, lo stesso viene usato per l'inizializzazione della variabile di classe NetworkIDS *nids*, grazie alla quale viene poi ricavato il file.

7.3 Test ed esempi

I seguenti test sono stati svolti con l'ausilio di Postman.

7.3.1 Upload del grafo su VEREFOO

Per poter testare la nuova funzionalità offerta dal traduttore implementato su VEREFOO, è stato necessario prima caricare un grafo che rappresentasse una rete contenente almeno un nodo di tipo network IDS. Il grafo è stato caricato usando:

Listing 7.4: Chiamata API per upload grafo su VEREFOO

```
POST http://localhost:8085/verefoo/fwd/nodes/addnfv
```

Il grafo scelto è stato quindi il seguente:

Listing 7.5: Esempio di grafo caricato su VEREFOO

```
<NFV xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../../xsd/nfvSchema.xsd">
  <graphs>
    <graph id="0">
     <node functional_type="WEBCLIENT" name="10.0.0.1">
     <neighbour name="30.0.0.1"/>
        <configuration description="A␣simple␣description" name="confA">
          <webclient nameWebServer="20.0.0.1"/>
        </configuration>
      </node>

      <node functional_type="FIREWALL" name="30.0.0.1">
        <neighbour name="10.0.0.1"/>
               <neighbour name="40.0.0.1"/>
        <configuration description="A␣simple␣description" name="conf1">
            <firewall defaultAction="ALLOW">
                      <elements>
                          <action>DENY</action>
                          <source>10.0.0.1</source>
                          <destination>20.0.0.1</destination>
                          <protocol>ANY</protocol>
                          <src_port>*</src_port>
                          <dst_port>*</dst_port>
                      </elements>
                  </firewall>
        </configuration>
      </node>

      <node functional_type="NETWORK_IDS" name="40.0.0.1">
        <neighbour name="30.0.0.1"/>
                  <neighbour name="20.0.0.1"/>
        <configuration description="A␣simple␣description" name="confA">
          <network_ids defaultAction="ALLOW">
                      <nids_rules>
                             <nids_action>ALERT</nids_action>
                             <protocol>TCP</protocol>
                             <source_ip>any</source_ip>
                             <source_port>any</source_port>
                             <destination_ip>any</destination_ip>
                             <destination_port>any</destination_port>
                             <direction>-&gt;</direction>
                             <message>Injection Attempt</message>
                             <option>http.uri; content:&quot;OR 1=1&quot;;
                                 nocase; sid:100;</option>
                      </nids_rules>
                </network_ids>
        </configuration>
      </node>

      <node functional_type="WEBSERVER" name="20.0.0.1">
        <neighbour name="40.0.0.1"/>
```

```
        <configuration description="A␣simple␣description" name="confB">
          <webserver>
 <name>b</name>
          </webserver>
        </configuration>
      </node>
    </graph>
   </graphs>
   <Constraints>
          <NodeConstraints>
          </NodeConstraints>
          <LinkConstraints/>
   </Constraints>
   <PropertyDefinition>
              <Property graph="0" name="ReachabilityProperty" src="10.0.0.1"
                  dst="20.0.0.1"/>
     <Property graph="0" name="ReachabilityProperty" src="20.0.0.1"
         dst="10.0.0.1"/>
   </PropertyDefinition>
   <ParsingString></ParsingString>
</NFV>
```

Analizzando la risposta che questa chiamata ritorna, è possibile trovare l'id con cui è stato salvato il nodo che rappresenta il network ids. Durante il test svolto, l'id assegnato a questo nodo è stato pari a 2.
Di seguito si trova la parte interessata:

Listing 7.6: Sezione risposta ottenuta a seguito del caricamento del grafo

```
<List>
    ---
        <id>2</id>
        <name>40.0.0.1</name>
        <functionalType>NETWORK_IDS</functionalType>
    ---
</List>
```

7.3.2 Generazione del file di configurazione

Per testare il traduttore, è necessario chiamare la seguente API:

Listing 7.7: Chiamata API per esecuzione del traduttore

```
GET http://localhost:8085/verefoo/fwd/deploy/getNetworkIds/2
```

In risposta è stato fornito quanto mostrato qui sotto:

Listing 7.8: Traduttore: Script ottenuto

```
#!/bin/sh
cmd="sudo"
RULES_FILE="/var/lib/suricata/rules/suricata.rules"
if [ -f "$RULES_FILE" ]; then
  : > "$RULES_FILE"
else
${cmd} touch "$RULES_FILE"
```

```
fi
echo 'alert tcp any any -> any any (msg:"Injection Attempt";http.uri;
    content:"OR 1=1"; nocase; sid:100;)' >> "$RULES_FILE"
```

7.3.3 Test con id errato

Una richiesta di generazione dello script di configurazione formulata in modo errato, ad esempio indicando come id uno diverso da quello del network IDS, genera un comportamento atteso. Si ottiene, infatti, il seguente errore:

Listing 7.9: Traduttore: Errore ricevuto

```
{
"timestamp": 1758494796085,
"status": 404,
"error": "Not Found",
"message": "not found",
"path": "/verefoo/fwd/deploy/getNetworkIds/1"
}
```

7.3.4 Esecuzione dello script

Una volta ottenuto lo script, la sua esecuzione configura le regole dell'IDS installato (Suricata). Nel test svolto, la regola impostata è la seguente:

Listing 7.10: Traduttore: Regola generata

```
alert tcp any any -> any any (msg:"Injection Attempt";http.uri;
    content:"OR 1=1"; nocase; sid:100;)
```

Dopo aver lanciato in esecuzione Suricata su una macchina con IP 172.22.51.243, da un'altra macchina, con IP 172.22.48.1, è stato inviato un pacchetto che potesse generare una corrispondenza e quindi un alert.
Il funzionamento osservato è stato quello atteso, in quanto l'alert generato è stato il seguente:

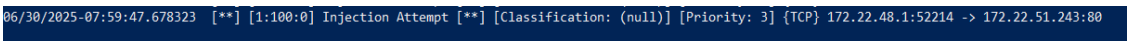

Figura 7.1: Alert generato da Suricata

Una descrizione più dettagliata dell'ambiente di test si può trovare nei capitoli successivi.

7.4 Gestione delle modalità di log di Suricata

Il traduttore illustrato finora permette di passare dalla configurazione di un IDS su VEREFOO a quella di un IDS reale. Il passo successivo alla sua creazione è stata la definizione di una classe di parsing che permettesse di configurare le properties di VEREFOO in base all'output generato dall'IDS, nello specifico da Suricata.
Le proprietà di VEREFOO sono gli elementi con cui è possibile inserire i requisiti di sicurezza, costituiscono, infatti, uno dei possibili input di REACT-VEREFOO[9], utili alla configurazione automatica delle risorse[10] . In particolare, le proprietà di tipo *IsolationProperty* indicano limitazioni alle comunicazioni tra due o più nodi di rete.

7.4.1 Output di Suricata

Gli alert di Suricata, prodotti dalle corrispondenze trovate dalle regole, vengono salvati in specifici file di log.
I diversi file di log contenuti in Suricata corrispondono alle numerose modalità con le quali questi output possono venire generati.
Il primo punto da affrontare per iniziare il lavoro sulla classe di parsing è stata l'analisi delle diverse modalità di output offerte da Suricata. Tra queste sono state prese in considerazione le due principali modalità di log: Fast Mode e EVE Mode.

- **Fast Mode**
  Questa modalità genera log composti da una singola linea, il che significa che ogni alert genererà una sola linea all'interno del file fast.log, il file dedicato a questa modalità di output.
  Di seguito è fornito un esempio di fast.log, presente sulla documentazione ufficiale di Suricata:

Listing 7.11: Esempio di log generato in fast mode

```
    10/05/10-10:08:59.667372 [**] [1:2009187:4] ET WEB_CLIENT ACTIVEX
        iDefense
  COMRaider ActiveX Control Arbitrary File Deletion [**]
      [Classification: Web
  Application Attack] [Priority: 3] {TCP} xx.xx.232.144:80 ->
      192.168.1.4:56068
```

  È importante notare che i log generati con questa modalità non sono personalizzabili.

- **EVE Mode**(Extensible Event Format)
  Questa tipologia di output permette di produrre log con un alto livello di dettaglio. Ogni alert generato produrrà infatti dei log in formato JSON, facilmente integrabili con altri software. Di seguito un esempio di eve.json, presente sulla documentazione ufficiale di Suricata:

Listing 7.12: Esempio di log generato in EVE mode

```
{
  "timestamp": "2017-04-07T22:24:37.251547+0100",
```

```
  "flow_id": 586497171462735,
  "pcap_cnt": 53381,
  "event_type": "alert",
  "src_ip": "192.168.2.14",
  "src_port": 50096,
  "dest_ip": "209.53.113.5",
  "dest_port": 80,
  "proto": "TCP",
  "metadata": {
    "flowbits": [
      "http.dottedquadhost"
    ]
  },
  "tx_id": 4,
  "alert": {
    "action": "allowed",
    "gid": 1,
    "signature_id": 2018358,
    "rev": 10,
    "signature": "ET HUNTING GENERIC SUSPICIOUS POST to Dotted Quad
        with Fake Browser 1",
    "category": "Potentially Bad Traffic",
    "severity": 2
  },
  "app_proto": "http"
}
```


A differenza dei log generati in Fast Mode, questi sono altamente personalizzabili. Modificando il file di configurazione .YAML di Suricata, è possibile modificare il formato con cui questi JSON vengono generati. Si può, ad esempio, includere il payload del pacchetto rilevato o estendere il log con maggiori informazioni riguardo al protocollo usato.

Il punto chiave da considerare per scegliere quale modalità sia quella più adatta allo scopo è il modo in cui le proprietà di VEREFOO vengono configurate e come sono composte. Le proprietà di VEREFOO sono formate da:

- **nome**: può essere uno tra IsolationProperty, ReachabilityProperty e CompleteReachabilityProperty. Non può essere ricavato dal log di Suricata.
- **id** del grafo a cui questa proprietà viene assegnata. Non può essere ricavato dal log di Suricata.
- **indirizzi IP** e **porte** di destinazione e sorgente.
- **protocollo** usato: può essere uno tra ANY, TCP, UDP e OTHER.

Dato che gli indirizzi IP, le porte e il protocollo usato sono contenuti in entrambi i formati, si può proseguire con la modalità EVE. Quest'ultima risulta più consona alla procedura di input.
Il log dovrà, infatti, essere fornito in input all'interno del body delle richieste API utili alle configurazioni delle proprietà. Disporre di un file JSON rende questo processo più veloce ed immediato.

Inoltre, la possibilità di poter formattare il contenuto del file di log attraverso determinate modifiche al file di configurazione di Suricata rende questa modalità la più adatta anche in vista di future implementazioni e aggiornamenti di VEREFOO.

7.4.2 Definizione della classe di parsing

Per eseguire il parsing del log in formato JSON prodotto da Suricata, è stata creata una classe chiamata *NIDSAlertInfo.java*. Questa classe è composta da cinque variabili: gli indirizzi IP sorgente e di destinazione, le porte sorgente e di destinazione e il protocollo utilizzato dalla comunicazione. Questa classe offre un metodo particolare, oltre al costruttore e ai vari getter e setter, chiamato *convertToProperty*, che ritorna un oggetto *Property* e richiede come parametri l'id del grafo per cui questa proprietà deve essere settata.

7.4.3 Creazione API dedicate

Il passo successivo alla definizione della classe è stata l'implementazione di nuove funzioni all'interno del controller che contiene le API che gestiscono la creazione e la modifica delle proprietà.
Sono quindi stati creati i seguenti metodi:

- **createRequirementsSetNIDS**:

Listing 7.13: Chiamata API per creare un set di requisiti

```
POST BASEURL/verefoo/adp/requirements/FromNIDSAlert
```

Come parametri richiede l'id del grafo a cui assegnare le proprietà e la collezione di NIDSAlertInfo generata attraverso il parsing del log.json. Crea il set di requisiti contenente la proprietà desiderata. Si noti che un set di requisiti è una collezione di proprietà.

- **updateRequirementsSetNIDS**:

Listing 7.14: Chiamata API per aggiornare un set di requisiti

```
PUT BASEURL/verefoo/adp/requirements/{rid}/FromNIDSAlert
```

Aggiorna il set di requisiti con l'id pari a *rid*. Come parametri richiede l'id del grafo a cui assegnare i requisiti aggiornati e la collezione di NIDSAlertInfo generata attraverso il parsing del log.json che sostituirà le vecchie proprietà.

- **createPropertyNIDS**:

Listing 7.15: Chiamata API per creare una proprietà

```
POST BASEURL/verefoo/adp/requirements/{rid}/properties/FromNIDSAlert
```

Aggiungela proprietà al set di requisiti con l'id pari a *rid*. Come parametri richiede l'id del grafo a cui assegnare la proprietà contenuta nel file log.json ricevuto in input.

- **updatePropertyNIDS**:

Listing 7.16: Chiamata API per aggiornare una proprietà

```
PUT BASEURL/verefoo/adp/requirements/{rid}/properties/
{pid}/FromNIDSAlert
```

Sostituisce la proprietà con l'id pari a *pid* contenuta nel set di requisiti con id pari a *rid* con una nuova proprietà contenuta nel file log.json ricevuto in input.

7.4.4 Fase di test

Per testare la gestione dei log prodotti da Suricata è stato necessario, come primo step, generare un grafo attraverso la chiamata alla seguente API:

Listing 7.17: Chiamata API per creare un grafo

```
POST http://localhost:8085/verefoo/adp/graphs
```

La struttura del grafo non è rilevante per il test, per tanto non viene qui riportata. Dalla risposta che si ottiene, si può ricavare l'id del grafo. In questo caso l'id ottenuto è pari a 192. Successivamente è stato estratto il seguente file di log, di tipologia EVE Mode:

Listing 7.18: Estrazione log in modalità EVE durante i test


```
{
"timestamp":"2025-08-03T18:12:15.627076+0200",
"flow_id":2190206884362222,
"in_iface":"eth0",
"event_type":"alert",
"src_ip":"172.22.48.1",
"src_port":56727,
"dest_ip":"172.22.51.243",
"dest_port":80,
"proto":"TCP",
"pkt_src":"wire/pcap",
"tx_id":0,
"alert":
       {
       "action":"allowed",
       "gid":1,
       "signature_id":100,
       "rev":0,
       "signature":"Injection Attempt",
       "category":"",
       "severity":3
       },
"http":
       {
       "hostname":"172.22.51.243",
       "url":"/login?username=admin'%20OR%201=1%20--",
       "http_user_agent":"Mozilla/5.0 (Windows NT; Windows NT 10.0; it-IT)
           WindowsPowerShell/5.1.19041.6093",
       "http_content_type":"text/html",
```

```
        "http_method":"GET",
        "protocol":"HTTP/1.1",
        "status":404,
        "length":275
        },
  "app_proto":"http",
  "direction":"to_server",
  "flow":
        {
        "pkts_toserver":4,
        "pkts_toclient":3,
        "bytes_toserver":399,
        "bytes_toclient":610,
        "start":"2025-08-03T18:12:15.575483+0200",
        "src_ip":"172.22.48.1",
        "dest_ip":"172.22.51.243",
        "src_port":56727,
        "dest_port":80
        }
  }
```


Questo log è formato dall'alert generato da una corrispondenza trovata con la seguente regola:

Listing 7.19: Regola di Suricata usata per la generazione del log

```
alert tcp any any -> any any (msg:"Injection Attempt";http.uri;
    content:"OR 1=1"; nocase; sid:100;)
```

Il file JSON è stato poi inserito all'interno del body della seguente chiamata API:

Listing 7.20: Chiamata API per inserire un set di requisiti usando l'output di Suricata

```
POST http://localhost:8085/verefoo/adp/requirements/
FromNIDSAlert?graph=192
```

Nella risposta fornita dalla funzione, si può individuare l'id del requirement creato, in questo caso è pari a 259. Per visualizzare la proprietà appena generata, è possibile eseguire la seguente chiamata API:

Listing 7.21: Chiamata API per ottenere un set di requisiti

```
GET http://localhost:8085/verefoo/adp/requirements/259
```

In risposta, si ottiene la proprietà così definita:

Listing 7.22: Proprietà creata

```
<property>
    <name>ISOLATION_PROPERTY</name>
    <graph>192</graph>
    <src>172.22.48.1</src>
    <dst>172.22.51.243</dst>
    <lv4Proto>TCP</lv4Proto>
    <srcPort>56727</srcPort>
    <dstPort>80</dstPort>
</property>
```

Nell'esempio vengono illustrati solo i campi definiti dalle funzioni appena definite. Le operazioni di aggiornamento e creazione dei requisiti usando file JSON contenenti log generati da più regole e di creazione e aggiornamento delle singole proprietà, essendo simili a quelle riportate qui sopra, non verranno illustrate. Basti sapere che i risultati ottenuti corrispondono alle attese.

Capitolo 8

Ambiente di Test

Viene qui descritto l'ambiente utilizzato per svolgere i test, in modo che questi siano facilmente replicabili.
La macchina utilizzata per i test dispone di Windows 10, per poter simulare la comunicazione tra due diversi terminali, in modo che l'IDS possa intercettare i pacchetti scambiati, è necessario installare una macchina virtuale. Quella scelta è stata WSL (Windows Subsystem for Linux) in quanto permette di utilizzare facilmente il terminale Ubuntu, da cui verrà operato l'IDS.
Per installare WSL, è sufficiente eseguire i seguenti comandi sul terminale:

Listing 8.1: Installazione WSL

```
wsl --install
```

Per eseguire WSL il comando è quello riportato qui sotto:

Listing 8.2: Esecuzione WSL

```
wsl
```

Dopo aver completato l'installazione della macchina virtuale, è necessario installare Suricata, ovvero l'IDS scelto. Per farlo, si possono eseguire i seguenti comandi:

Listing 8.3: Installazione Suricata

```
sudo apt update && sudo apt upgrade -y
sudo apt install suricata -y
```

Per verificare che l'installazione sia andata a buon fine:

Listing 8.4: Controllo versione Suricata

```
suricata --version
```

La versione di Suricata utilizzata durante lo svolgimento di questa tesi è la 7.0.3.[7]
Il file di configurazione di Suricata, durante lo svolgimento di questa tesi, si trovava in:*etc/suricata/suricata.yaml*, mentre il file che conterrà le regole utilizzate si trovava in:*var/lib/suricata/rules/suricata.rules*. Per testare le API è stato usato Postman, installato sulla macchina Windows. Per evitare problemi di formattazione, gli script forniti dal traduttore, prima di essere eseguiti sulla macchina Ubuntu, sono stati formattati attraverso il seguente comando:

Listing 8.5: Formattazione script generato dal traduttore

```
dos2unix executableScript.sh
```

Naturalmente, se dos2unix non dovesse essere già disponibile, è possibile installarlo con:

Listing 8.6: Installazione dos2unix

```
sudo apt install dos2unix -y
```

Per eseguire lo script su WSL, il comando utilizzato è il seguente:

Listing 8.7: Esecuzione script su WSL

```
sudo ./executableScript.sh
```

Dopo aver eseguito lo script e quindi aver settato le regole di Suricata, quest'ultimo può essere avviato con:

Listing 8.8: Esecuzione Suricata su WSL

```
sudo suricata -i eth0 -c /etc/suricata/suricata.yaml
```

Dopo aver ricavato l'indirizzo IP della macchina WSL, da quella Windows sono stati inviati messaggi che potessero risultare in un match, come ad esempio:

Listing 8.9: Esempio di messaggio scambiato durante i test

```
curl "http://172.22.51.243/login?username=admin' OR 1=1 --"
```

Per monitorare la generazione degli alert, è possibile eseguire il seguente comando:

Listing 8.10: Comando per monitorare il fast log

```
sudo tail -f /var/log/suricata/fast.log
```

Questo mostrerà i cambiamenti che avvengono all'interno del log in modalità fast. L'installazione e la configurazione di VEREFOO non vengono qui illustrate, in quanto già facilmente reperibili nella repository ufficiale.

Capitolo 9

Conclusione e lavori futuri

Il lavoro di questa tesi si è concentrato inizialmente sul confronto e sull'analisi delle diverse tipologie di Intrusion Detection Systems, in particolare sono stati confrontati gli IDS la cui gestione può essere implementata all'interno del framework VEREFOO.
L'obiettivo principale era infatti compiere uno studio che possa permettere, in futuro, l'integrazione degli IDS all'interno del framework, il quale non offre al momento soluzioni che li comprendano.
Il primo contributo di questa tesi consiste nell'esposizione delle principali funzioni offerte da questi componenti, dei loro punti di forza e di debolezza, insieme a un'analisi delle loro configurazioni di sicurezza. Il punto principale affrontato durante lo svolgimento di questo lavoro è stato, infatti, l'analisi delle modalità con cui le diverse tipologie di IDS possono aumentare il livello di sicurezza all'interno di una rete.
Il primo passo è stato la scelta dell'IDS su cui basare l'implementazione di questi su VEREFOO. Quello designato è stato Suricata, per la sua versatilità e per le sue prestazioni.
In base ai dati raccolti è stato quindi creato un modello astratto di Network-Based IDS, in seguito implementato nello schema strutturale di VEREFOO. Questo modello rende possibile rappresentare questi componenti all'interno del framework VEREFOO.
Su questo modello si è basato il lavoro svolto per la costruzione del traduttore, che rappresenta il vero primo approccio verso gli IDS da parte del framework, che fino ad ora non comprendeva nel proprio perimetro gli Intrusion Detection Systems.
In seguito ad un'analisi delle modalità di output dell'IDS, è stata poi implementata la lettura dei log di Suricata con l'obiettivo di configurare determinate proprietà all'interno dei grafi presenti su VEREFOO.
Tutte le nuove funzioni sono poi state testate su un ambiente dedicato.
Insieme alla costante evoluzione che le reti e la sicurezza informatica attraversano ogni giorno, le metodologie per garantire quest'ultima si espandono. VEREFOO, nato con l'obiettivo di automatizzare i processi che aiutano a garantire un livello adeguato di sicurezza all'interno di una rete, deve poter crescere in modo da offrire configurazioni eterogenee e complete.
Il contributo principale di questo lavoro è quello di porsi come punto di partenza per lavori futuri che possano estendere le funzioni che il framework già offre per

altri componenti, quali i firewall e le VPN, anche agli IDS, in quanto, al momento, l'implementazione degli Intrusion Detection Systems all'interno di VEREFOO non è ancora completa. L'obiettivo dei lavori futuri dovrà quindi essere l'automatizzazione della configurazione degli IDS.
Inoltre, lo schema degli IDS definito durante questa tesi è nato da un'analisi effettuata, in particolare, su Suricata, le cui funzioni non si limitano a quelle di un semplice IDS. Suricata può, infatti, anche svolgere il ruolo di Intrusion Prevention System (IPS), un sistema di prevenzione che interviene attivamente per bloccare possibili attacchi. Una futura implementazione di questi componenti all'interno di VEREFOO, basata su quanto prodotto durante questo lavoro, permetterebbe al framework di ampliare le sue possibilità.

Bibliografia

[1] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, "Automation for network security configuration: State of the art and research trends," *ACM Computing Surveys*, vol. 56, no. 3, pp. 1–37, 2023.

[2] D. Bringhenti, S. Bussa, R. Sisto, and F. Valenza, "Atomizing firewall policies for anomaly analysis and resolution," *IEEE Transactions on Dependable and Secure Computing*, vol. 22, no. 3, pp. 2308–2325, 2025.

[3] D. Bringhenti, F. Pizzato, R. Sisto, and F. Valenza, "Autonomous attack mitigation through firewall reconfiguration," *International Journal of Network Management*, 2025.

[4] D. Bringhenti, R. Sisto, and F. Valenza, "A novel abstraction for security configuration in virtual networks," *Computer Networks*, vol. 228, p. 109745, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128623001901

[5] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated firewall configuration in virtual networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1559–1576, 2023.

[6] D. Bringhenti, R. Sisto, and F. Valenza, "Automating vpn configuration in computer networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 22, no. 1, pp. 561–578, 2025.

[7] O. I. S. Foundation, "Suricata user guide." [Online]. Available: https://docs.suricata.io/en/suricata-7.0.3/

[8] O. Project, "Ossec hids's documentation." [Online]. Available: https://ossec-documentation.readthedocs.io

[9] D. Bringhenti, F. Pizzato, R. Sisto, and F. Valenza, "A looping process for cyberattack mitigation," in *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*, 2024, pp. 276–281.

[10] F. Pizzato, D. Bringhenti, R. Sisto, and F. Valenza, "Automatic and optimized firewall reconfiguration," in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, 2024, pp. 1–9.