UNIVERSITÀ DEGLI STUDI DI URBINO CARLO BO
Corso di Laurea in Informatica Applicata

Monitoraggio in tempo reale della rete locale con visualizzazione web

Relazione del progetto d'esame per il corso di
Reti di Calcolatori

Sessione Autunnale a.a. 2024/25

Balducci Milena Matricola 321791

Indice

1 Introduzione

Negli ultimi anni la sicurezza delle reti locali è diventata un tema di importanza cruciale. La crescente diffusione di dispositivi connessi, la pervasività di servizi online e l'elevata complessità degli attacchi informatici hanno reso necessario lo sviluppo di strumenti capaci di monitorare e analizzare in tempo reale il traffico di rete. Una rete locale, anche se di dimensioni ridotte e apparentemente "protetta" da un perimetro fisico, è in realtà vulnerabile a una molteplicità di minacce: attacchi di tipo spoofing, flood di pacchetti, tentativi di intrusione e canali nascosti di esfiltrazione dati.

Il progetto descritto in questa relazione nasce dalla volontà di sviluppare un sistema capace non solo di osservare il traffico di rete e fornire statistiche utili su di esso, ma anche di individuare potenziali anomalie che possano indicare un attacco informatico in corso. Per questo motivo, accanto al monitoraggio, è stato realizzato un simulatore di attacchi capace di produrre traffico malevolo, utile a verificare la robustezza dei meccanismi di rilevamento.
L'idea di fondo è realizzare uno strumento didattico ma al tempo stesso pratico, che unisca funzionalità di packet sniffing, analisi del traffico e un'interfaccia web intuitiva per la visualizzazione dei dati, oltre a permettere di testare scenari realistici di attacco.

Dal punto di vista tecnologico, il progetto integra più componenti:

- **Python** è stato scelto per la fase di packet sniffing e analisi, grazie alla libreria Scapy, che consente di catturare e manipolare pacchetti di rete in modo semplice ed efficace. Python si è rivelato ideale anche per l'implementazione della logica di elaborazione e aggregazione dei dati.
- **Node.js** è stato utilizzato per la gestione del backend in tempo reale, in particolare tramite **WebSocket**, così da garantire un flusso costante di informazioni tra il sistema di monitoraggio e la dashboard web.
- **Docker** ha permesso di containerizzare i vari componenti del progetto, offrendo un ambiente di esecuzione isolato e facilmente replicabile, che semplifica la fase di deploy su macchine diverse.
- **Dashboard Web**: realizzata con HTML, CSS e JavaScript, arricchita da librerie come Chart.js per la visualizzazione grafica e DataTables per la gestione interattiva delle tabelle. Questa interfaccia è pensata per presentare i dati in maniera chiara, accessibile e immediatamente interpretabile anche da utenti non esperti.

Gli obiettivi principali del progetto possono essere così riassunti:

- Sviluppare uno sniffer di rete in grado di catturare pacchetti dall'interfaccia di rete locale e raccogliere informazioni rilevanti (IP sorgente/destinazione, porte, protocollo, dimensione, ecc.).
- Implementare il rilevamento in tempo reale di possibili attacchi, distinguendo traffico legittimo da comportamenti sospetti e generando alert immediati.

- Costruire un simulatore di attacchi, utile sia a scopo didattico che per testare l'efficacia del sistema. Tra gli attacchi simulati figurano ARP spoofing, SYN flood, ICMP flood, DNS tunneling e tentativi di DDoS.
- Fornire una dashboard interattiva che permetta di monitorare la rete locale con grafici e tabelle aggiornati in tempo reale, dando all'utente una panoramica sempre aggiornata dello stato della rete.
- Garantire portabilità e semplicità di deploy, grazie all'utilizzo di ambienti virtualizzati e containerizzati, che consentono di eseguire il sistema su macchine diverse senza particolari problemi di compatibilità.

Dal punto di vista metodologico, il lavoro si è sviluppato in diverse fasi: in primo luogo è stato implementato il motore di cattura pacchetti e di analisi, sfruttando le potenzialità di Python e definendo le strutture dati necessarie a raccogliere le statistiche. Successivamente, si è passati alla creazione del modulo di comunicazione in tempo reale, con un server Node.js capace di ricevere e inoltrare i dati attraverso socket. Una fase successiva ha riguardato la realizzazione della dashboard web, progettata per essere responsiva e user-friendly, con un'interfaccia in grado di mostrare sia dati istantanei che storici.

La fase di deployment è stata affrontata utilizzando **Docker** e **ambienti virtuali Python**, con l'obiettivo di isolare le dipendenze e garantire una configurazione stabile e riproducibile. Questo approccio consente di semplificare sia i test in fase di sviluppo sia l'eventuale installazione su infrastrutture più ampie.

2 Architettura del Sistema

Il progetto è stato organizzato in modo modulare, così da separare chiaramente i diversi compiti e facilitare sia lo sviluppo che la manutenzione. In particolare, l'architettura si articola in tre componenti principali:

- **Server e Dashboard Web**, sviluppati con `Node.js`, `HTML`, `CSS` e `JavaScript`, responsabili della gestione della comunicazione in tempo reale tramite WebSocket e della presentazione dei dati all'utente. La dashboard consente di monitorare graficamente il traffico di rete attraverso grafici interattivi e tabelle aggiornate dinamicamente, rendendo immediata la lettura delle informazioni.
- **Network Sniffer**, implementato in `Python`, che ha il compito di catturare i pacchetti direttamente dall'interfaccia di rete locale. Questo modulo non si limita alla raccolta dei pacchetti, ma include anche funzioni di analisi e rilevamento di possibili attacchi, inviando i risultati al server per la successiva visualizzazione.
- **Simulatore di Attacchi**, anch'esso sviluppato in `Python`, progettato per generare traffico malevolo realistico e riprodurre diversi scenari di attacco (ad esempio ARP spoofing, SYN flood o DNS tunneling). Questo componente è fondamentale per testare l'efficacia dei meccanismi di rilevamento integrati nello sniffer.

La struttura delle cartelle del progetto riflette questa suddivisione logica, come mostrato in figura:

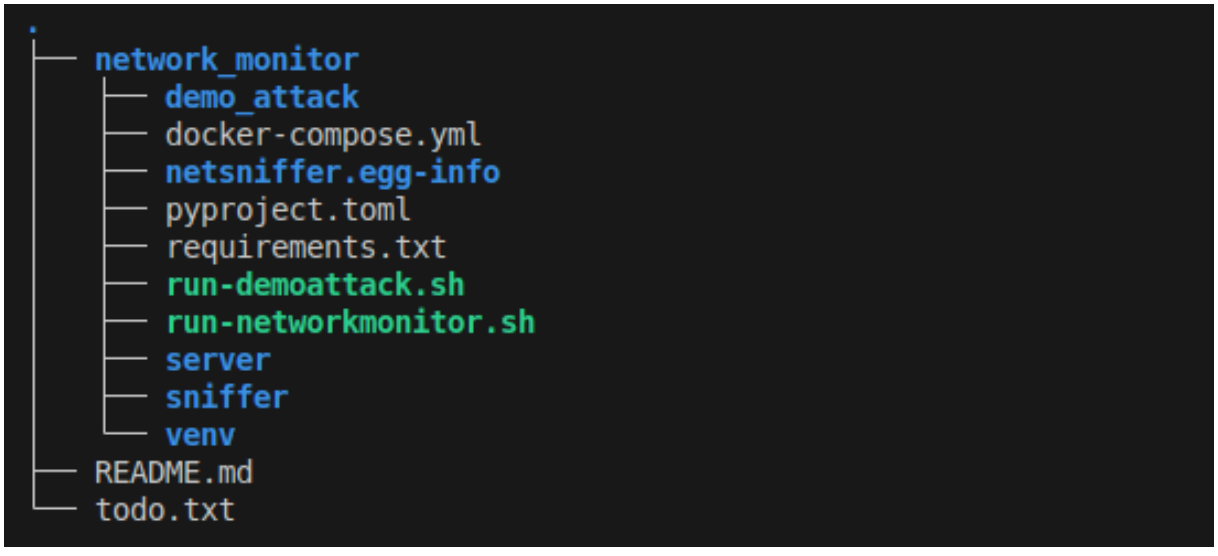

Figura 1: Struttura delle cartelle

- `demo_attack/` raccoglie gli script per lanciare i vari attacchi simulati, organizzati in modo da poter essere avviati singolarmente o tutti insieme.

- `server/` ospita il server Node.js e i file necessari alla dashboard, compresi gli script JavaScript per la gestione dei grafici (Chart.js) e delle tabelle (DataTables).
- `sniffer/` contiene il cuore dell'applicazione: lo sniffer Python, le librerie di supporto, i moduli per l'elaborazione dei pacchetti e le logiche di rilevamento degli attacchi.
- `venv/` rappresenta il virtual environment Python, indispensabile per isolare le dipendenze e garantire portabilità su macchine diverse senza conflitti di librerie.

Infine, per semplificare l'avvio delle componenti principali, sono stati predisposti due script helper: `run-demoattack.sh`, che esegue il simulatore di attacchi con un menù interattivo, e `run-networkmonitor.sh`, che avvia in parallelo il network sniffer e il server con dashboard web, permettendo all'utente di monitorare immediatamente gli effetti del traffico generato.

2.1 Network Sniffer

Lo sniffer costituisce il cuore del sistema ed è stato realizzato in `Python` sfruttando la libreria **Scapy**, una delle più diffuse e potenti per la manipolazione e l'analisi del traffico di rete. Scapy consente infatti di catturare pacchetti direttamente dall'interfaccia di rete, decodificarne i contenuti e, se necessario, generarne di nuovi.
Grazie a questa libreria è stato possibile implementare un componente versatile, capace non solo di monitorare il traffico in tempo reale, ma anche di estrarne informazioni utili per successive analisi statistiche e per il rilevamento di attacchi.

2.1.1 Funzionalità principali

Lo sniffer è progettato per svolgere una serie di compiti fondamentali:

- **Cattura pacchetti** dall'interfaccia di rete specificata, garantendo la raccolta di tutti i dati necessari per l'analisi.
- **Estrazione dei metadati** più rilevanti dai pacchetti, tra cui:
  - Indirizzi IP sorgente e destinazione.
  - Indirizzi MAC dei dispositivi coinvolti nella comunicazione.
  - Protocollo a livello di rete, trasporto e applicazione.
  - Dimensione del pacchetto e, in caso di TCP, i flag utilizzati nella comunicazione.
- **Mantenimento di statistiche di traffico**, come la classifica degli IP più attivi, la distribuzione del traffico per protocollo e l'andamento dell'input/output della rete.
- **Rilevamento di attacchi** attraverso euristiche e soglie predefinite, che permettono di individuare comportamenti anomali o sospetti.
- **Invio in tempo reale** dei dati raccolti e degli eventuali alert di sicurezza al server Node.js, tramite comunicazione WebSocket.

2.1.2 Moduli principali

Per garantire chiarezza e modularità, il codice dello sniffer è stato suddiviso in più file, ciascuno con responsabilità ben definite:

- `sniffer.py`: rappresenta il punto di ingresso principale, inizializza la cattura dei pacchetti e gestisce il callback che richiama le funzioni di analisi.
- `analyzer.py`: contiene le logiche di analisi dei protocolli principali (IP, TCP, UDP, DNS), con funzioni dedicate all'estrazione dei metadati e all'aggiornamento delle statistiche.
- `security.py`: implementa gli algoritmi di rilevamento degli attacchi, tra cui ARP spoofing, ICMP flood, SYN flood, TCP reset, UDP amplification, DNS tunneling e attacchi DDoS simulati.
- `socket_client.py`: si occupa della comunicazione con il server Node.js tramite WebSocket, inviando pacchetti, statistiche e alert in tempo reale.

Questa suddivisione rende il sistema più leggibile e manutenibile, oltre a facilitare eventuali estensioni future (ad esempio, l'aggiunta di nuovi protocolli o meccanismi di rilevamento).

2.1.3 Comunicazione WebSocket

Un aspetto cruciale del progetto è la capacità dello sniffer di inviare in tempo reale i dati analizzati al server Node.js, in modo che possano essere immediatamente visualizzati nella dashboard web. Per realizzare questa funzionalità è stato utilizzato **Socket.IO Client**, una libreria che permette di instaurare una connessione WebSocket stabile ed efficiente.

Gli eventi principali trasmessi al server sono i seguenti:

- `"packet_log_data"`: pacchetti catturati e statistiche di dettaglio.
- `"ip_log_data"`: lista degli IP più attivi.
- `"protocol_traffic_data"`: distribuzione del traffico per protocollo.
- `"io_traffic_data"`: andamento temporale del traffico in entrata e uscita.
- `"security_alert_listener"`: alert generati dai moduli di rilevamento attacchi.

Questa architettura permette una comunicazione bidirezionale costante tra il backend Python e il frontend Node.js, riducendo la latenza e consentendo un monitoraggio che sia effettivamente in tempo reale.

2.2 Server Node.js e Dashboard Web

Il server rappresenta il punto di contatto tra il backend di analisi e l'utente finale, gestendo la visualizzazione in tempo reale dei dati attraverso una dashboard web interattiva. Questa componente è stata implementata in **Node.js**, sfruttando il framework **Express** per la gestione delle rotte e della distribuzione dei file statici. La parte frontend della dashboard utilizza due librerie principali: **Chart.js**, impiegata per la creazione di grafici dinamici e aggiornabili, e **DataTables**, che permette di costruire tabelle interattive, ordinabili e filtrabili. Grazie a questa combinazione di strumenti, è stato possibile realizzare un'interfaccia intuitiva e al tempo stesso ricca di funzionalità, pensata per offrire un colpo d'occhio immediato sull'andamento del traffico di rete.

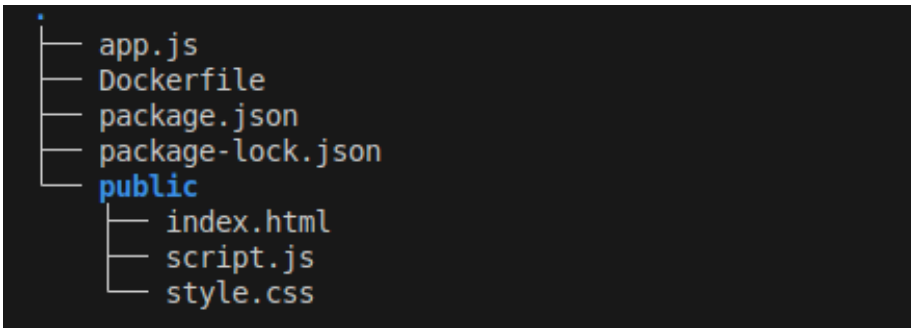

Figura 2: Struttura del server Node.js

2.2.1 Architettura della Dashboard

Lo scopo principale della dashboard è mostrare le informazioni relative al traffico rilevato dallo sniffer Python, offrendo una rappresentazione chiara e facilmente interpretabile anche da parte di un utente non esperto di reti. Per motivi di leggibilità e semplicità del design frontend, si è deciso di concentrarsi sui pacchetti IP, che rappresentano la parte più significativa del traffico analizzato. L'analisi di pacchetti ARP e ICMP viene comunque eseguita in background dal backend, così da garantire il rilevamento di potenziali minacce anche se queste non vengono mostrate direttamente nella dashboard. L'idea, in futuri sviluppi, è quella di ampliare progressivamente le informazioni rese disponibili al frontend, così da fornire un livello di dettaglio ancora maggiore.

L'architettura della dashboard è suddivisa in tre sezioni principali:

1. **Intestazione:** oltre al titolo dell'applicazione, questa sezione mostra anche eventuali *alert di sicurezza*. Gli avvisi compaiono dinamicamente quando il backend rileva comportamenti sospetti e scompaiono dopo un intervallo di tempo, permettendo di richiamare subito l'attenzione dell'utente senza appesantire l'interfaccia.

2. **Grafici interattivi con Chart.js:** i dati ricevuti vengono rappresentati in forma grafica per facilitare la comprensione dei trend. In particolare:

- **Line chart per traffico IP in/out:** mostra l'andamento temporale del traffico (in Mbytes) in ingresso e in uscita, permettendo di individuare picchi improvvisi o anomalie.
- **Bar chart per traffico per IP:** evidenzia i cinque indirizzi IP più attivi sulla rete. Ogni barra rappresenta la quantità di traffico associata a un singolo IP.
- **Pie chart per traffico per protocollo di trasporto:** offre una visione immediata della distribuzione percentuale del traffico tra i diversi protocolli di trasporto (ad esempio TCP e UDP).

3. **Tabella live con DataTables:** elenca in tempo reale i pacchetti IP catturati, fornendo una visione dettagliata di ciascun pacchetto. Le colonne includono:
   - Timestamp della cattura.
   - Protocollo di rete.
   - IP sorgente e destinazione.
   - Dimensione del pacchetto.
   - Protocollo di trasporto utilizzato.
   - MAC address sorgente e destinazione.
   - Porte sorgente e destinazione (sport, dport).
   - Flags TCP rilevati.
   - Eventuali informazioni DNS, come query e tipo di query.

   La tabella è interattiva e si aggiorna continuamente, consentendo anche l'ordinamento e la ricerca dei pacchetti.

Figura 3: Snapshot della dashboard

2.2.2 Comunicazione WebSocket lato Node.js

Per ricevere i dati dallo sniffer Python, il server Node.js utilizza **Socket.IO Server**, che gestisce le connessioni WebSocket in modo affidabile. Questo consente una comunicazione bidirezionale costante tra backend e frontend, riducendo al minimo la latenza e rendendo la dashboard veramente reattiva.

Gli eventi principali gestiti dal server includono:

- `"packet_log_data"`: aggiorna in tempo reale la tabella dei pacchetti.
- `"ip_log_data"`: aggiorna il grafico dei top IP.
- `"protocol_traffic_data"`: aggiorna il bar chart con la distribuzione dei protocolli.
- `"io_traffic_data"`: aggiorna il line chart relativo al traffico in entrata e uscita.
- `"security_alert_listener"`: aggiunge un nuovo alert alla sezione dedicata e notifica immediatamente l'utente.

Ogni evento ricevuto dal backend Python viene immediatamente propagato al frontend, garantendo che la dashboard sia costantemente aggiornata.

2.2.3 Aggiornamento in tempo reale

La combinazione di **Socket.IO**, **Chart.js** e **DataTables** consente di realizzare un sistema di visualizzazione live, senza la necessità di ricaricare la pagina.
I dati inviati dallo sniffer vengono ricevuti, elaborati e mostrati istantaneamente, permettendo all'utente di osservare l'evoluzione del traffico e di identificare tempestivamente eventuali anomalie o attacchi simulati.
Questa caratteristica rende la dashboard uno strumento particolarmente utile per attività di monitoraggio continuo e di risposta rapida alle minacce.

2.3 Simulatore di Attacchi

Il simulatore di attacchi rappresenta la componente utilizzata per testare e validare il funzionamento del sistema di monitoraggio. Il suo scopo è generare traffico malevolo sulla rete locale, in modo da verificare che le euristiche e le soglie di rilevamento implementate nello sniffer Python siano effettivamente in grado di riconoscere i vari scenari di attacco.

Le tipologie di attacco simulate comprendono:

- **ARP Spoofing:** per ingannare la tabella ARP e intercettare il traffico.
- **SYN Flood:** per saturare le connessioni TCP aprendo numerose richieste incomplete.
- **ICMP Flood:** per inviare grandi quantità di pacchetti ICMP e ridurre la disponibilità della rete.

- **TCP Reset Attack:** per interrompere connessioni TCP attive inviando pacchetti di reset.
- **UDP Amplification:** per generare traffico amplificato sfruttando server vulnerabili.
- **DNS Tunneling:** per simulare la trasmissione nascosta di dati attraverso query DNS.
- **DDoS:** per simulare un attacco distribuito volto a esaurire le risorse del sistema.

Ogni attacco è stato progettato per superare le soglie definite nello sniffer, così da generare alert reali e verificabili nella dashboard.

Per facilitare l'utilizzo durante le dimostrazioni, è stato sviluppato uno script bash denominato `run-demoattack.sh`, che fornisce un menu testuale da riga di comando. Tramite questo menu, l'utente può selezionare rapidamente l'attacco che desidera simulare, lo script si occupa poi di eseguire automaticamente lo script Python corrispondente, riducendo la complessità dell'operazione.

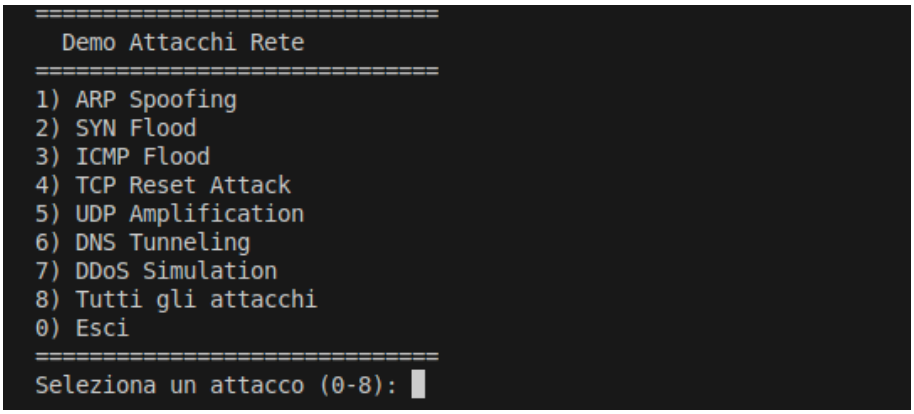

Figura 4: Interfaccia da riga di comando del simulatore di attacchi

2.4 Ambiente di Sviluppo

2.4.1 Virtual Environment e Containerizzazione

Per garantire portabilità e isolamento degli ambienti, il sistema è stato strutturato in modo da separare chiaramente le dipendenze delle diverse componenti. Il modulo sniffer e il simulatore di attacchi in Python vengono eseguiti all'interno di un **virtual environment (venv)**, così da evitare conflitti con altre installazioni presenti sulla macchina ospite e mantenere facilmente riproducibile l'ambiente di lavoro.

Parallelamente, il server Node.js è stato containerizzato utilizzando **Docker**, scelta che consente di garantire un comportamento uniforme su qualunque macchina e di semplificare le operazioni di avvio e configurazione. Per la gestione e l'orchestrazione dei container viene utilizzato **Docker Compose**, che permette di definire e avviare in modo automatico tutti i servizi necessari al funzionamento del sistema.

2.4.2 Permessi per Scapy

Un aspetto importante da considerare riguarda i permessi necessari per l'esecuzione di Scapy. Essendo la cattura di pacchetti un'operazione che richiede privilegi elevati sul sistema, lo sniffer necessita di essere avviato con permessi di root. Gli script forniti includono quindi la possibilità di eseguire il modulo sniffer con `sudo`, in modo da garantire un corretto funzionamento senza richiedere configurazioni manuali all'utente.

2.5 Deployment

Vista la complessità del sistema, composto da più moduli e ambienti di esecuzione, si è scelto di automatizzare la fase di avvio e distribuzione attraverso uno script di supporto. Lo script bash `run-networkmonitor.sh` è stato sviluppato proprio con l'obiettivo di ridurre al minimo le operazioni manuali richieste all'utente, rendendo l'intero processo semplice e affidabile.

Le funzioni principali svolte dallo script, eseguite nell'ordine indicato, sono le seguenti:

1. Pulizia di eventuali container Docker residui che potrebbero contenere build obsolete o difettose.
2. Avvio del server Node.js all'interno di un container Docker.
3. Eliminazione di eventuali virtual environment Python non più validi.
4. Creazione di un nuovo virtual environment Python tramite `pip`.
5. Installazione del modulo sniffer in modalità `editable`, così da semplificare lo sviluppo.
6. Avvio dello sniffer sulla rete locale con i privilegi necessari.

In questo modo, con un unico comando, l'intero sistema di monitoraggio viene avviato e configurato automaticamente, garantendo efficienza e riducendo al minimo i possibili errori derivanti da operazioni manuali.

3 Sfide e Difficoltà Incontrate

3.1 Progettazione delle funzioni di analisi

I pacchetti rilevati possono fornire una grande quantità di informazioni sul traffico di rete. In fase di progettazione del sistema è stato quindi necessario individuare con attenzione quali informazioni fossero realmente utili da mostrare all'utente.
Mostrare troppe informazioni rischiava di generare confusione e ridurre l'efficacia della dashboard, mentre essere troppo selettivi avrebbe potuto comportare una perdita di dati potenzialmente rilevanti.
Come illustrato nella Sezione 2, si è scelto di suddividere il carico informativo tra backend e frontend:

- **Frontend:** limitato ai soli pacchetti IP, con informazioni sugli indirizzi IP più attivi, i protocolli di trasporto e la quantità di traffico I/O. L'obiettivo era mantenere l'interfaccia semplice, intuitiva e veloce da interpretare.
- **Backend:** include anche pacchetti ARP e ICMP, consentendo un'analisi più completa del traffico utile a individuare possibili anomalie e minacce non immediatamente visibili a livello applicativo.

Questa separazione ha permesso di mantenere la dashboard user-friendly senza sacrificare l'accuratezza dell'analisi lato server. Tuttavia, la definizione di quali protocolli includere, quali metriche mostrare e con quale granularità ha richiesto diversi tentativi e un bilanciamento continuo tra chiarezza e completezza.

3.2 Aggiornamento in tempo reale

L'aggiornamento in tempo reale della dashboard è una delle caratteristiche chiave del sistema, ma ha rappresentato una sfida significativa in fase di sviluppo.
L'approccio iniziale prevedeva l'uso di **API REST** esposte dal server Node.js; tuttavia, questo metodo non consentiva un vero aggiornamento live, introducendo ritardi nella visualizzazione e rischi di perdita di pacchetti.

La soluzione scelta è stata l'adozione di **WebSocket**, attraverso la libreria **Socket.IO**, per la comunicazione tra client Python e server Node.js. Questa decisione ha portato benefici in termini di reattività, ma anche difficoltà implementative, in particolare:

- **Sincronizzazione tra server e client:** poiché sniffer e server vengono avviati in parallelo, è stato necessario implementare meccanismi di retry lato client Python, così da garantire che il tentativo di connessione avvenga solo dopo l'avvio del server. È stato adottato un sistema di ritentativi fino a dieci volte prima di generare un errore.
- **Compatibilità tra ambienti:** le librerie di WebSocket, seppur diffuse, possono comportare differenze di comportamento tra sistemi operativi o versioni diverse di Node.js/Python, richiedendo una fase di debug accurata.

3.3 Esecuzione dello sniffer

In fase iniziale si era valutato di eseguire lo sniffer all'interno di un container Docker per migliorarne la portabilità. Tuttavia, Docker utilizza una rete interna dedicata ai container, che avrebbe impedito il rilevamento diretto del traffico sulla scheda di rete locale, a meno di configurazioni particolari (bridge o network host) che avrebbero introdotto criticità di sicurezza.

Si è quindi deciso di configurare lo sniffer come un **package Python installabile**, con un'app CLI dedicata. Questa scelta ha risolto il problema della visibilità del traffico reale, ma ha portato a nuove difficoltà:

- **Gestione dei permessi di root:** Scapy richiede privilegi elevati per catturare pacchetti. L'esecuzione come root però comporta la perdita della variabile d'ambiente che punta alle librerie Python installate dall'utente. Mentre per un singolo script è possibile risolvere con il flag `-E`, l'esecuzione di un intero package ha richiesto l'adozione di un **virtual environment (venv)**, così da isolare le dipendenze e gestire correttamente i permessi.
- **Portabilità:** alcuni moduli di Scapy possono comportarsi diversamente a seconda della piattaforma (Linux, macOS, Windows), richiedendo test su più ambienti per garantire un funzionamento stabile. Per vincoli legati alle risorse hardware disponibili, la validazione del sistema è stata condotta esclusivamente in ambiente **Linux/Ubuntu**. Non è stato pertanto possibile effettuare test su sistemi operativi alternativi, in quanto la macchina utilizzata non disponeva della capacità computazionale necessaria all'esecuzione di macchine virtuali con prestazioni adeguate. Questo lascia dei dubbi sull'efficienza del sistema al di fuori dell'ambiente nel quale è stato sviluppato e testato.

3.4 Altre difficoltà

Oltre agli aspetti già discussi, il progetto ha posto ulteriori sfide:

- **Integrazione tra componenti eterogenei:** la necessità di far comunicare in maniera fluida componenti scritti in linguaggi diversi (Python per lo sniffer, Node.js per il server, JavaScript per il frontend) ha comportato problemi di compatibilità e debug complesso.
- **Gestione degli errori e resilienza:** il sistema doveva garantire continuità di funzionamento anche in caso di errori transitori (ad esempio, perdita temporanea della connessione WebSocket). Ciò ha richiesto l'implementazione di meccanismi di riconnessione e fallback.
- **Usabilità della dashboard:** tradurre dati di basso livello (pacchetti, flag, porte) in rappresentazioni intuitive per l'utente finale non è stato immediato. La scelta dei grafici, delle metriche e della terminologia ha richiesto iterazioni successive.

3.5 Possibili Miglioramenti Futuri

- Testare il sistema su ambienti diversi da Linux/Ubuntu.

- Espandere le funzioni di analisi per fornire più informazioni sul traffico locale.
- Implementare algoritmi di machine learning per rilevamento anomalie.
- Memorizzazione dello storico dei pacchetti in un database.
- Aggiungere attacchi più complessi (MITM, session hijacking, ARP poisoning avanzato).
- Migliorare la dashboard con filtri avanzati e notifiche in tempo reale, ad esempio tramite bot Telegram.

4 Conclusioni

Il progetto ha integrato con successo cattura e analisi del traffico di rete, rilevamento di attacchi e visualizzazione dei dati in tempo reale. Gli obiettivi di monitoraggio, analisi e test tramite attacchi simulati sono stati raggiunti, fornendo una base solida per possibili sviluppi futuri nel campo della sicurezza di rete e della visualizzazione interattiva dei dati.

5 Riferimenti

Scapy documentation: https://scapy.readthedocs.io
Socket.IO documentation: https://socket.io
Docker documentation: https://docs.docker.com
Python official documentation: https://docs.python.org/3/