



**UNIVERSITÀ
DEGLI STUDI
DI BERGAMO**

SCUOLA DI INGEGNERIA

Corso di Laurea in Ingegneria Informatica

Classe n. L-8

Sviluppo di tool per l'analisi del traffico di rete in contesti avversari

Relatore: Chiar.mo Prof./Chiar.ma Prof.ssa
Paraboschi Stefano

Prova finale di Davide Guido Bernasconi

Matricola n. 1040844

**ANNO
ACCADEMICO**

2017 / 2018



Indice

1 Descrizione della tesi	5
1.1 Introduzione alle CTF	5
1.2 Descrizione della tesi	6
1.2.1 Scopo	6
1.2.2 Specifiche	7
1.3 Progettazione	7
1.3.1 Progetto	8
1.3.2 Descrizione generale del software	8
2 Setup dell'ambiente di lavoro	9
2.1 Ambiente Linux	10
2.2 Virtualbox e le VM	10
2.2.1 Configurazione VM	10
2.3 Editor	11
2.4 Github	11
2.5 Snort	11
3 Prima parte: lo sniffer	12
3.1 File di configurazione	13
3.2 Libpcap	13
3.2.1 Il sistema di packet capture	14
3.2.2 Filtri BPF	14
3.3 Prototipi	16
3.3.1 Prototipo 1: filtri a callback	16
3.3.2 Prototipo 2: multithreading	16
3.3.3 Prototipo 3: il filtro ad OR	17
3.3.4 perf	18
3.4 I file pcap	19

3.4.1 Descrizione del filtro.	20
3.5 Time split	20
3.5.1 Implementazione.	21
3.6 Invio del file.	22
3.6.1 Tool.	22
3.6.2 Ipotesi.	23
3.6.2.1 Attesa dell'invio dei pacchetti	23
3.6.2.2 Creazione continua di file (e successiva cancellazione)	24
3.6.2.3 Creazione dei file e concorrenza	24
3.6.2.4 Soluzione intermedia.	25
3.6.2.5 Gestione dell'errore.	25
3.6.3 Rsync.	25
3.7 initpcap.sh.	26
3.7.1 Il problema delle password.	26
3.7.2 Implementazione.	27
3.7.2.1 Test degli argomenti da riga di comando	27
3.7.2.2 Public key authentication di ssh	27
3.7.2.3 Controllo sull'invio della password.	27
3.7.2.4 pcap evolve.c.	28
3.7.2.5 Attesa e ricezione dei segnali	28
3.8 Testing	29
 4 Seconda Parte: Lato client e analisi dei dati.	29
4.1 Snort	30
4.1.1 Le rules in breve	30
4.1.2 local.rules e flags	32
4.3 Il file di configurazione di Snort	33
4.4 Analizzatore	34
4.4.1 Il codice	35
4.4.1.1 Classe PcapInfo	35
4.4.1.2 Classe Analyzer	35

4.4.2 I thread	36
4.5 GUI	37
4.5.1 Descrizione generale e uso	37
4.5.2 Funzionalità dell'interfaccia	38
4.5.3 Implementazione	38
4.5.3.1 Libreria	38
4.5.3.2 La classe GUI	38
4.6 Testing	40
 5 Limiti	 40
5.1 lista delle porte non dinamica	40
5.2 Apertura dei file pcap non dinamica	40
5.3 Attesa dei processi invocanti: problemi di performance	41

1 Descrizione della tesi

La tesi si prefigge come obiettivo la progettazione e lo sviluppo di un analizzatore di traffico di rete sviluppato ad-hoc per le CTF, le competizioni internazionali di sicurezza informatica.

Di conseguenza, ogni decisione presa all'interno del progetto di questo software va compresa nel contesto di queste competizioni.

1.1 Introduzione alle CTF

Le Capture The Flag sono competizioni di sicurezza informatica volte al fine di far apprendere concetti fondamentali della sicurezza informatica, ma vengono anche utilizzate per reclutare e scoprire talenti nella sicurezza. Queste gare si compongono di team di programmatori che si sfidano in svariate sfide, tutte con l'unico scopo di conquistare le flag, la cui cattura e submit permette al team di acquisire punti. Le CTF si suddividono in due categorie: Jeopardy e Attack-Defense. Le CTF Jeopardy sono gare in cui i membri dei vari team si cimentano in singole challenge di vario genere (exploit, crypto, web, miscellaneous, ...), ognuna delle quali viene attribuito un punteggio in proporzione alla difficoltà della sfida. Nell'ambito della tesi, le CTF da considerare è quella di Attack-Defence, in quanto la tesi si sviluppa apposta per questo tipo di gare.

In questo tipo di CTF i membri di un team montano sui propri calcolatori una macchina virtuale, la VulnBox, una Virtual Machine su cui girano servizi appositamente vulnerabili ad exploit. Lo scopo dei team consiste nel difendere, ovvero creare patch dei servizi risolvendo le vulnerabilità di questi sulle proprie Vulnbox, e allo stesso tempo attaccare i servizi delle macchine dei membri delle altre squadre al fine di catturare le flag nascoste nelle Vulnbox avversarie. Il tutto è permesso dal fatto che vengono consegnati i binari dei servizi affinché sia possibile analizzarli alla ricerca delle vulnerabilità, in modo tale che sia possibile sia difendersi che attaccare.

In queste competizioni il punteggio viene calcolato come la somma dei punteggi individuali per ogni servizio, e, a sua volta, questo punteggio si compone di tre componenti:

- Offense: punti derivanti dalle flag catturate dagli altri team e sottoposte al Gameserver nel periodo di tempo valido.

- Defense: Punti derivanti dalla difesa delle proprie flag da parte degli attacchi degli altri team.
- SLA(Service Level Agreement): Punti derivanti dalla disponibilità e corretto comportamento dei propri servizi.

Inoltre è fondamentale conoscere e prendere in considerazione il Gameserver, ovvero il server di gioco della CTF gestito dagli organizzatori, che si occupa di salvare periodicamente le flag nella Vulnbox usando funzionalità insite nei servizi forniti e, sempre grazie a queste, le recupera successivamente .

Ogni servizio è progettato per permettere il GameServer di salvare uno specifico token per ogni flag oppure per generarne uno che successivamente ritorna al Gameserver.

Il GameServer usa questi token per controllare periodicamente la presenza della flag nella Vulnbox per verificare il corretto utilizzo della Vulnbox. In ogni caso, recupera la flag memorizzata attraverso questi token e determina il proprio SLA . Ovviamente, ne consegue che un uso scorretto della Vulnbox e dei suoi servizi compromette il proprio team attraverso l'abbassamento del proprio punteggio.

1.2 Descrizione della tesi

Il progetto si divide in due programmi distinti: la prima parte verrà fatta eseguita sulla VulnBox e provvederà a filtrare il traffico internet (sia in ingresso che in uscita) secondo alcune porte specificate dall'utente (ovvero le porte sulle quali gireranno i servizi vulnerabili), catturarlo e a salvarlo in file pcap, i quali verranno inviati sulle macchine ordinarie per successiva analisi; su queste ultime, infatti, verrà eseguita la seconda parte del programma, che consiste nell'analisi dei file ricevuti grazie all'uso di un software per l'analisi del traffico al fine di categorizzare il traffico ricevuto secondo opportune specifiche.

1.2.1 Scopo

Lo scopo di questo progetto consiste nel trovare il più velocemente possibili tentativi di exploit da parte di team avversari in modo tale da cercare di creare patch per i servizi

vulnerabili e riuscire anche a studiare i tentativi di exploit subito al fine di apprenderne le dinamiche e riutilizzarli a proprio vantaggio.

1.2.2 Specifiche

Di seguito verranno elencate le specifiche che il progetto dovrà rispettare:

Features prioritarie

- Le porte da monitorare sono definite in un file di configurazione dal quale il programma leggerà la lista delle porte.
- Il traffico di rete verrà scritto su file pcap e verrà generato un file per ogni porta presente sulla lista.
- Avere un file con il dump di tutte le altre porte non presenti nel file, al fine di controllare lo scorretto utilizzo della macchina durante la competizione.
- Il programma sarà diviso in due parti: una parte server che girerà sulla Vulnbox, che avrà il compito di catturare il traffico, generare i pcap e inviarli ogni cinque minuti, e una parte client, che riceverà i pcap e li analizzerà tramite un tool di analisi a scelta del tesista.

Features secondarie:

- Fare il clustering dei pacchetti con un software a scelta in modo da distinguere:

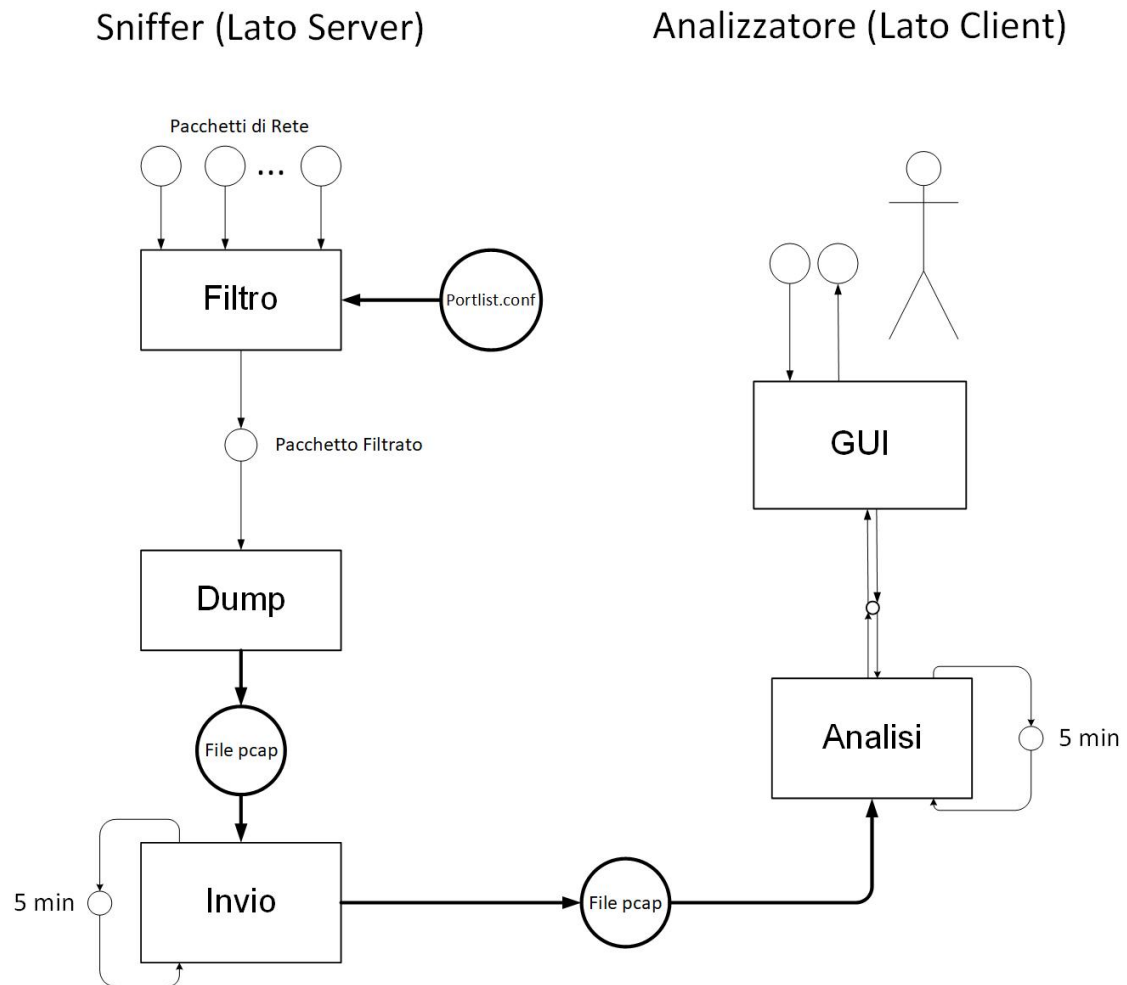
Tipologia	Traffico in ingresso	Traffico in uscita
Gameserver	Flag query	Flag
Exploits	Exploit	Flag
Spam	Qualsiasi	Non-flag

- Creare un interfaccia grafica che permette di selezionare la porta e la partizione temporale e mostri le informazioni sul risultato dell'analisi dei pcap.

1.3 Progettazione

Di seguito viene presentata una visione grafica in Microsoft Visio del progetto del software sviluppato e verrà riportata una descrizione generale dell'effettiva implementazione del software.

1.3.1 Progetto



1.3.2 Descrizione generale del software

La prima parte, come detto precedentemente, si preoccupa di ricevere il traffico entrante e uscente dalla VulnBox, di suddividerlo in partizioni temporali di 5 minuti e di scriverlo su file pcap. Non verrà ovviamente catturato tutto il traffico, ma soltanto quello passante per una determinata lista di porte. I file verranno così ripartiti in un file per ogni porta selezionata e per ogni slot temporale.

I pacchetti che transitano sulla rete verranno catturati e per ognuno di questi, al momento della cattura, verrà invocata una callback che si occuperà di filtrare il pacchetto: se questo pacchetto, nel suo TCP header, non presenta una delle porte

specificate nel file di configurazione, verrà semplicemente scartato, in caso contrario potrà essere invocata la funzione che farà il log del pacchetto sul file pcap corrispondente. Una volta che saranno passati 5 minuti, termina la partizione temporale corrente, vengono chiusi tutti i pcap e inviati alle macchine che si occuperanno dell'analisi successiva.

La seconda parte del programma consiste in un piccolo software che si occupa di eseguire l'analisi dei pcap utilizzando un tool specifico adatto al compito, ovvero Snort. Il componente software chiamato analizzatore manderà in esecuzione Snort ogni 5 minuti, e cercherà i pcap generati e inviati dal software illustrato precedentemente. Se non li trova attende questi ultimi, altrimenti li analizza in modo ordinario. Inoltre viene presentata una interfaccia grafica in cui l'utente interagirà selezionando porta e partizione temporale, oppure solo questa ultima, affinché venga fatta richiesta all'analizzatore di analizzare il file che risponde ai parametri della richiesta. Se il file è già stato analizzato, provvede a far vedere i risultati ottenuti precedentemente, altrimenti verrà invocato Snort per analizzare quel file e scrivere su file i risultati da far visualizzare all'interfaccia. È possibile inoltre, come funzionalità aggiuntiva dell'interfaccia grafica, visualizzare qualsiasi pcap si desideri. Il componente che si occupa dell'analisi e l'interfaccia grafica seguono il paradigma client-server, dove l'analizzatore può essere visto come il server e la GUI come il client.

2 Setup dell'ambiente di lavoro

Questo capitolo è dedicato alla descrizione dell'ambiente di sviluppo e dei programmi di ausilio per la tesi. La descrizione non entrerà nel dettaglio dei programmi/tools usati ma verrà comunque fornita una breve introduzione in modo da comprenderne il motivo principale per cui sono stati utilizzati. Ecco di seguito l'elenco di questi programmi:

- Virtualbox per l'utilizzo delle macchine virtuali;
- Microsoft Office per scrivere la parte scritta della tesi;
- IDE: Nessuno;
- Sistema Operativo su cui giravano le VM: Windows 10.
- Sistemi Operativi delle VM: Ubuntu a 32 e 64 bit;
- Gestione delle versioni: Github

- NIDS: Snort
- Altro: Microsoft Visio;

2.1 Ambiente Linux

La scelta dell'ambiente Linux è stata presa sulla base dell'idea secondo cui gli utenti che usufruiranno del programma sono programmatori o esperti di sicurezza informatica che hanno notevole familiarità con i sistemi GNU/Linux, oltre al fatto che in quanto solitamente le CTF richiedono l'utilizzo di un sistema operativo di questo tipo.

La scelta tra le varie distro Linux su cui sviluppare il software è ricaduta su Ubuntu.

2.2 Virtualbox e le VM

Lo sviluppo e il testing dei programmi sono stati eseguiti sulle macchine virtuali: sono stati installati su Windows delle macchine virtuali con VirtualBox di Oracle.

Per simulare la rete di due macchine sono state usate due macchine virtuali, di cui una simulava la VulnBox, dove veniva fatto girare lo sniffer di rete, mentre l'altra una macchina ordinaria che riceveva i file e che eseguiva l'analisi.

L'utilizzo delle macchine virtuali è stato scelto per due motivi:

1. Non necessita l'installazione di Ubuntu sulla macchina.
2. Simulazione di host diversi nella rete che eseguivano le due parti del programma.

2.2.1 Configurazione VM

Per quando riguarda la configurazione delle VM, l'unico aspetto degno di nota per la tesi riguarda la configurazione di rete: infatti, nelle opzioni la macchina è stata cambiata da NAT (impostazione di default) a Scheda con Bridge. Infatti con NAT veniva creata una rete virtuale tra VM e host in modo diretto, con una propria rete interna, il quale era possibile per la VM comunicare con l'esterno grazie alla conversione di tipo NAT da parte dell'host. Attraverso questa configurazione veniva simulato il caso in cui la VulnBox risiede sullo stesso host su cui viene fatta l'analisi. Con la Scheda di bridge, invece, la VM aveva un proprio IP basato sulla rete locale a cui è collegato l'host,

quindi figurava come un host separato appartenente alla LAN, allo stesso modo di come viene visto l'host ospite all'interno della rete. In questo modo viene simulato il caso in cui la VulnBox risiede su un host diverso dalla macchina per l'analisi.

2.3 Editor

Nello sviluppo del codice non sono stati utilizzati IDE, ma solo editor e compilatori in modo indipendente. Gli editor usati sono stati Gedit e Vim in modo alternato. I due tool presentano una fondamentale differenza: Gedit presenta un'interfaccia grafica che lo rende molto simile al Notepad di Windows, mentre Vim utilizza esclusivamente l'interfaccia a riga di comando.

2.4 Github

Per la gestione delle versioni e tutti i vari aggiornamenti del software durante lo sviluppo, è stato utilizzato Github.

Github è un sito web che fornisce un servizio di hosting per progetti software utilizzato principalmente utilizzato dagli sviluppatori che caricano il codice sorgente dei loro programmi in repository, di cui è disponibile il download. Inoltre, è possibile interagire con lo sviluppatore tramite un sistema di issue tracking, pull request e commenti che permette di migliorare il codice della repository risolvendo bug o aggiungendo funzionalità.

Accedendo al sito con nome utente Berna96, è possibile consultare la tesi finale e scaricare il codice dal branch master.

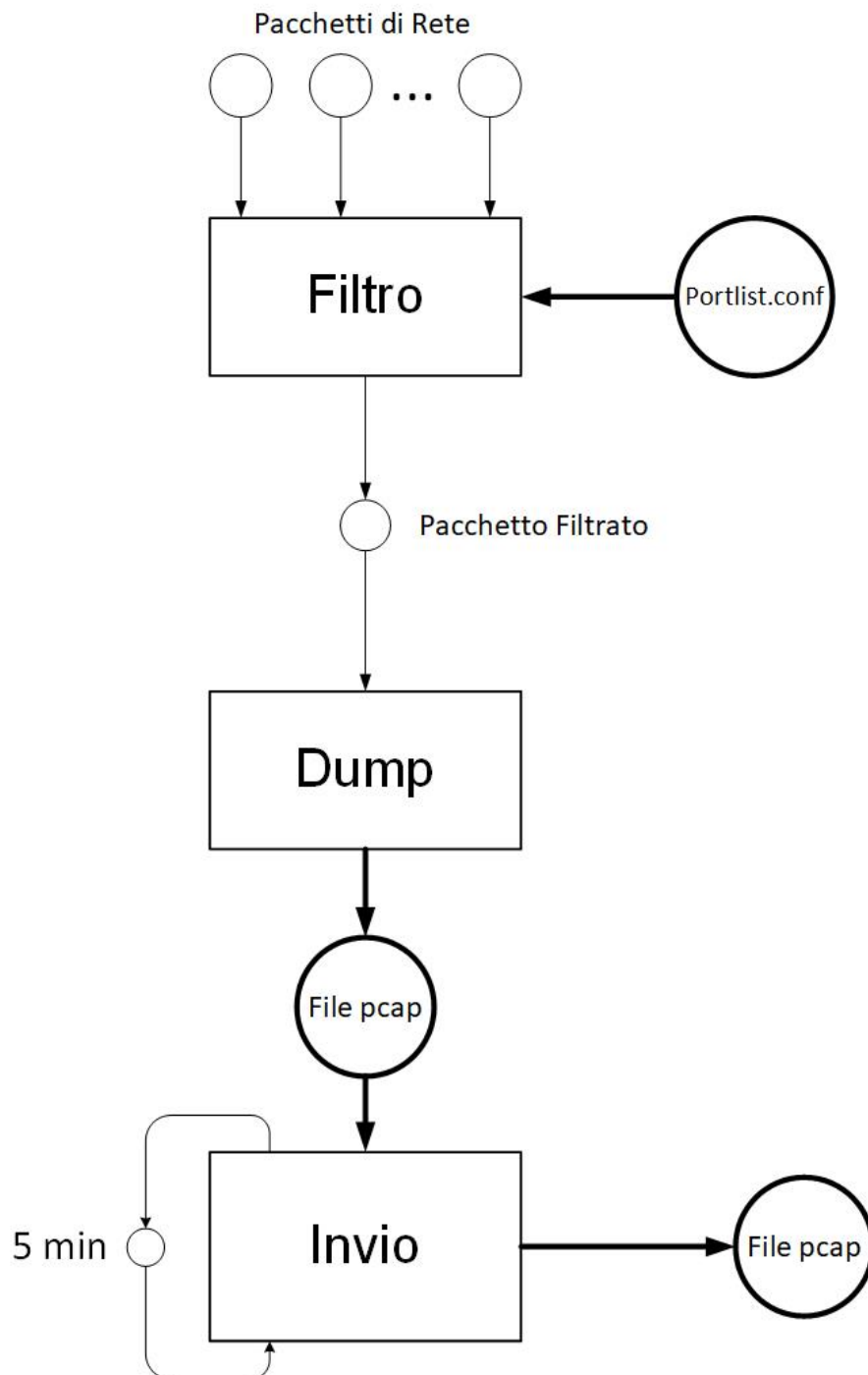
2.5 Snort

Il NIDS utilizzato per analizzare i pcap e trovare exploit e flag è stato Snort, tool scelto tra numerosi programmi di analisi presenti al seguente URL: <https://github.com/caesar0301/awesome-pcaptools>.

In questa sezione non verrà affrontata la spiegazione di Snort in quanto una spiegazione esaustiva verrà fornita al capitolo 4.

3 Prima parte: lo sniffer

La prima parte di questa tesi consiste nel sviluppare lo sniffer del traffico internet che verrà eseguito sulla Vulnbox e che provvederà a fornire i file pcap all'analizzatore. Di seguito viene presentato il dettaglio del progetto.



3.1 File di configurazione

Il file di configurazione presenta l'elenco delle porte da scansionare secondo la seguente sintassi:

```
port1;port2;port3;port4;port5;...;port(n-1);port(n)
```

Figura 1: sintassi della lista di porte da monitorare

In questo modo il programma leggerà il file e l'utente scriverà (con un normale editor) le porte che vuole andare a monitorare su questo file (seguendo la convenzione sopra altrimenti il programma va in segfault). L'utente dovrà configurare il file PRIMA di eseguire il programma in quanto il programma la configurazione non è dinamica, ovvero una volta definite le porte nel file e avviato il programma qualsiasi modifica al file non viene letta dal programma. Se si vogliono cambiare le porte da monitorare bisognerà obbligatoriamente interrompere il programma e riavviarlo come vedere le modifiche attuate.

3.2 Libpcap

Libpcap è una libreria open-soure portatile di C/C++ (Packet Capture library) che fornisce un interfaccia ad alto livello al packet capture systems, ovvero al sistema per la cattura dei pacchetti.

Tutti i pacchetti sulla rete sono accessibili attraverso questo sistema(API), perfino quelli destinati ad altri host. Inoltre supporta un sistema di salvataggio e di lettura dei pacchetti catturati grazie a un "savefile". Inizialmente sviluppato per sistemi Unix e Unix-like, esiste un porting della libreria per Windows chiamata Winpcap. Le funzioni della libreria utilizzate si trovano nella documentazione ufficiale e verranno spiegate brevemente durante la discussione del codice.

La documentazione ufficiale di libpcap si trova sul sito di Tcpcdump al seguente link: <http://www.tcpdump.org/manpages/pcap.3pcap.html>

3.2.1 Il sistema di packet capture

Il packet capture è l'attività di collezionare i pacchetti che viaggiano su una rete. Senza di esso, non sarebbe stato possibile implementare lo sniffer di questa tesi. Il packet

capture viene utilizzato in moltissime applicazioni nel campo della sicurezza informatica: alcuni esempi sono i tool per i test statistici delle reti, i NIDS, password sniffers, ARP poisoners e molti altri ancora.

Il packet capture system è ciò che rende possibile tutto questo. Quando un pacchetto viene ricevuto da un'interfaccia di rete, il driver della scheda di rete, di norma, lo passa allo stack di rete, che a sua volta consegnerà la parte dati del pacchetto senza header ai processi utente. Quando si usa questo, il driver di rete invia una copia di tutti i pacchetti transitanti, sia quelli trasmessi che quelli ricevuti, a una parte del kernel chiamato packet filter. Di default lasciano passare qualsiasi pacchetto, ma come verrà spiegato nel capitolo successivo, questi filtri possono offrire elevate capacità di filtraggio. Successivamente, a differenza del protocol stack, i pacchetti verranno poi consegnati al livello applicativo esattamente come sono stati catturati, ovvero con sia la parte dati che con gli header di tutti i protocolli di rete.

3.2.2 Filtri BPF

I filtri BPF sono filtri che vengono applicati al sistema di Packet Capture che lavora a livello kernel, ovvero offrono un servizio di filtraggio sui pacchetti prima di essere passati al livello utente, ovvero all'applicazione che usufruisce di questo servizio.

Come illustrato nel capitolo paragrafo precedente, nel sistema di cattura i filtri si interpongono tra il driver della scheda di rete e il livello applicazione.

Questo risulta particolarmente comodo al programmatore, in quanto riceverà direttamente i pacchetti già filtrati senza eseguire un'azione di filtraggio nel proprio programma, oltre al fatto che questi filtri risultano ottimizzati e performanti.

I filtri devono essere generati e compilati secondo una sintassi predefinita: la filter expression consiste di una o più primitive. Le primitive di solito consistono di un id (che può essere un nome o un numero) preceduto da uno o più qualificatori. Ci sono 3 diversi tipi di qualifier:

- *type*: i *type* qualifiers indicano a cosa si riferiscono l'id del nome o il numero. I types possibili sono *host*, *net*, *port* e *portrange*. Alcuni esempi, "host foo", "net 128.3", "port 20", "portrange 6000-6008".

- *dir*: i *dir* qualifiers specificano una particolare direzione di traffico dell' id. Possibili direzioni sono *src*, *dst*, *src or dst*, *src and dst*, *ra*, *ta*, *addr1*, *addr2*, *addr3*, e *addr4*. Alcuni esempi, "src foo", "dst net 128.3", "src or dst port ftp-data".
- *proto*: i *proto* qualifiers limitano il controllo a un particolare protocollo di rete. I protocolli supportati sono *ether*, *fddi*, *tr*, *wlan*, *ip*, *ip6*, *arp*, *rarp*, *decnet*, *tcp* e *udp*. Alcuni esempi, "ether src foo", "arp net 128.3", "tcp port 21", "udp portrange 7000-7009", "wlan addr2 0:2:3:4:5:6".

Per ulteriori dettagli è possibile consultare la documentazione ufficiale al seguente link: <http://www.tcpdump.org/manpages/pcap-filter.7.html>.

Il programma che applica questi filtri deve prevedere due passaggi: una fase di compilazione, dove viene passata la filter expression e viene creato il filter program secondo uno pseudo-linguaggio, e successivamente avviene l'applicazione del filtro al kernel.

Lo scopo di questi filtri, oltre a fornire al programmatore un supporto di filtraggio dei pacchetti necessari, servono anche a evitare di catturare tutti i pacchetti passanti per la rete in quanto potrebbe facilmente verificarsi numerose perdite di pacchetti. Il programma non prevedeva l'uso di filtri particolari, ma per evitare questo problema è stato necessario comunque mettere un filtro ed è stato scelto di ricevere solo pacchetti provenienti da connessioni TCP, in quanto durante le CTF le connessioni prevalentemente usate per fare exploit sono proprio TCP. La filter expression usate sono due e sono le seguenti: "tcp" e "ip src host \$MY_IP", dove \$MY_IP è sostituito dall'ipv4 dell'host.

3.3 Prototipi

In questo programma sono stati sviluppati alcuni prototipi di lavoro al fine di analizzare le performance e testare quale potesse essere la soluzione più efficiente da sviluppare. Le idee per lo sviluppo del programma sono state molteplici e soltanto tre di queste sono state implementate, di cui solo una è presente nell'implementazione finale.

Di seguito verranno presentate le tre soluzioni sopracitate nel dettaglio, valutandone i vantaggi e gli svantaggi.

3.3.1 Prototipo 1: filtri a callback

La prima soluzione consiste nell'utilizzo di filtri a callback. Questa soluzione si integra perfettamente con l'interfaccia fornita da libpcap: infatti, la funzione `pcap_loop` richiama una callback ogni qualvolta riceve un pacchetto (filtrato da un filtro BPF) e questa può essere sfruttata per implementare un filtro che distingue il traffico catturato. Il core del filtro consiste in una semplice scansione del pacchetto con la lista di porte da monitorare e, se trova il match con una porta, richiama la funzione `handle_packet`. Siccome la funzione “`apply_filter`” viene richiamata ogni qualvolta un pacchetto viene consegnato al livello utente per *qualsiasi* pacchetto, è stato aggiunto un filtro BPF per filtrare soltanto il traffico TCP (unico traffico che ci interessa), al fine di ridurre la probabilità di perdita dei pacchetti e, di conseguenza, aumentare le performance del programma, dovuto al minor numero di invocazioni.

Vantaggi:

1. Soluzione semplice da implementare e più compatta a livello di codice, in quanto il cuore del filtro risiede tutto in un'unica funzione.
2. Questa soluzione evita l'utilizzo di thread.

Svantaggi:

1. La funzione viene richiamata ogni qualvolta arriva un pacchetto filtrato (quindi un pacchetto TCP) indipendentemente dal tipo di porta; ne consegue che il numero di invocazioni è elevato e questo influisce negativamente sulle performance.

3.3.2 Prototipo 2: multithreading

Questa soluzione prevede la creazione di thread che hanno il compito di gestire il traffico per ogni porta. L'idea di base consiste nel generare un numero di thread uguale al numero di porte da monitorare in modo che ogni singolo thread sia associato a una sola e una sola porta, affinché il thread gestisca il traffico passante per la porta associata. Ogni thread è stato implementato nel seguente modo:

- è stato compilato e inizializzato un filtro BPF con la filter expression “`tcp port $PORT`”, dove `$PORT` è la porta che il thread andrà a monitorare;
- è stato creato un handle per ogni thread al fine di gestire il packet capture;

- invocare la funzione `pcap_loop` all'interno del codice del thread.

Vantaggi:

1. La soluzione offre tutti i vantaggi tipici della programmazione multitasking: ottimizzazione dello sfruttamento della CPU, ottimizzazione del tempo di esecuzione dovuto alla concorrenza dei thread oppure, eventualmente, alla parallelismo;
2. La soluzione segue un modello che per sua natura è parallelo.

Svantaggi

1. Ogni thread crea un handle per la gestione del packet capture, quindi se abbiamo N thread, significa che abbiamo N handle e questo pesa sulla memoria proporzionalmente a N; questo assicura che *tutti* i pacchetti transitanti nella rete passino in *ogni* filtro BPF di *ogni* thread (cosa che non sarebbe stata possibile usando un solo handle generato nel thread main) e quindi gli handle, in totale, cattureranno e i filtri processeranno **N x pacchetti totali**. Inoltre esiste la possibilità che qualche pacchetto possa essere perso da qualche thread.

3.3.3 Prototipo 3: il filtro ad OR

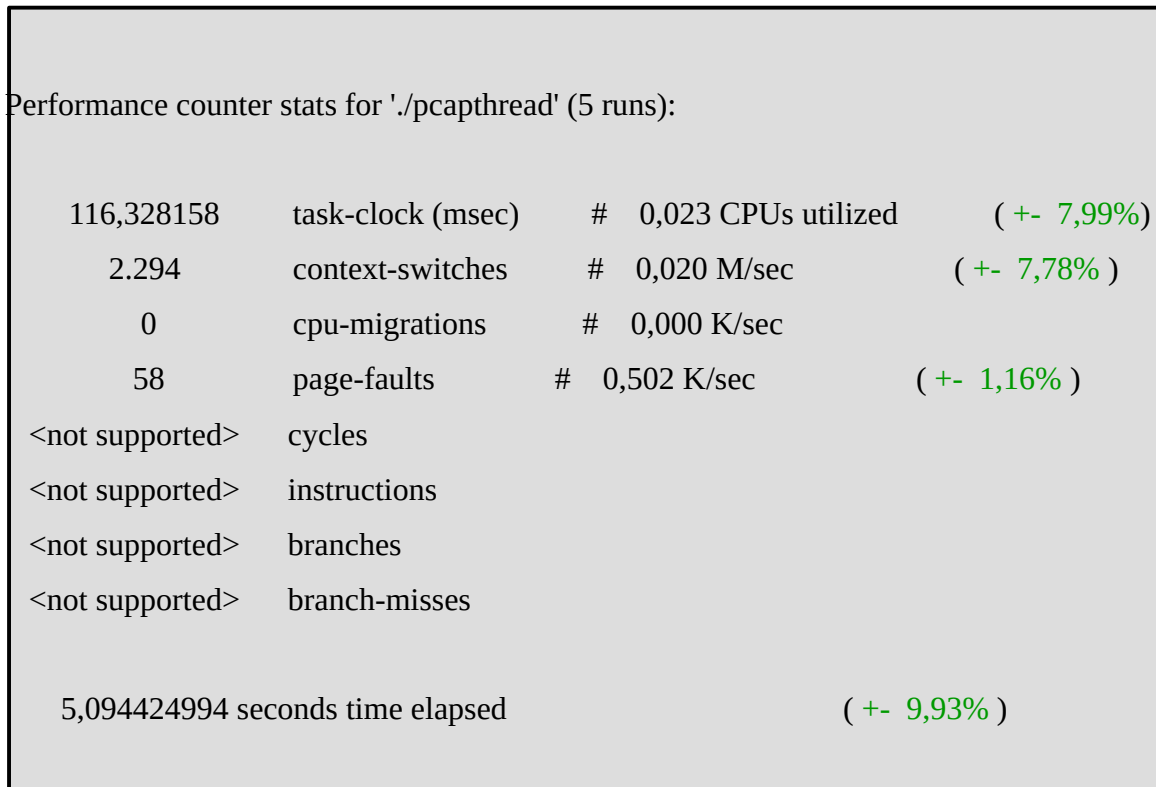
Questa soluzione inizialmente prevedeva un filtro con la sintassi “tcp port \$PORT1 or \$PORT2 or ... or \$PORTn” così che vengano consegnati solo i pacchetti che hanno nell'header TCP le porte selezionate, in ogni caso per salvare nel file corrispondente il pacchetto si doveva eseguire una scansione all'interno della callback e quindi la funzione di `handle_packet` è esattamente indentica al filtro `apply_filter` della prima soluzione. Purtroppo risulta sbagliata per quanto riguarda le richieste della tesi, in quanto anche i pacchetti che non hanno quelle porte devono essere salvati in un pcap apposito. La soluzione di conseguenza diventa identica alla prima soluzione, con l'unica differenza che consiste nell'aggiunta di una condizione, all'interno della callback `apply_filter`, che effettua il controllo con una condizione in disgiunzione delle porte. Si può vedere questa soluzione, infatti, come una variante della prima. Vantaggi e svantaggi sono identici a quelli della prima soluzione.

3.3.4 perf

Per scegliere quale dei tre approcci sviluppati era il migliore per quanto riguarda le performance, è usato un tool per l'analisi prestazionale dei processi, ovvero perf.

Il tool perf offre un ricco set di comandi al fine di collezionare e analizzare le performance e tracciare i dati. L'uso della riga di comando ricorda Git e implementa un numeroso set di comandi, come ad esempio stat, record, report, e molti altri ancora.

Per questa tesi è stato utilizzato il comando stat, in quanto permette di eseguire un programma e raccogliere statistiche sulle performance. Sono state eseguite cinque repliche, in modo da ottenere una media sulle statistiche. E decidere su quale criterio raccolte e su cui basare la scelta di quale soluzione fosse più performante. Di seguito viene mostrato un esempio dell'output di perf stat:



Performance counter stats for './pcapthread' (5 runs):

116,328158	task-clock (msec)	#	0,023 CPUs utilized	(+- 7,99%)
2.294	context-switches	#	0,020 M/sec	(+- 7,78%)
0	cpu-migrations	#	0,000 K/sec	
58	page-faults	#	0,502 K/sec	(+- 1,16%)
<not supported>	cycles			
<not supported>	instructions			
<not supported>	branches			
<not supported>	branch-misses			
5,094424994	seconds time elapsed			(+- 9,93%)

Figura 2: esempio di parametri delle performance del primo prototipo

Come si può osservare, le statistiche misurate sono le seguenti: overhead del context switch, numero di migrazione della CPU, numero di page faults, la frequenza di clock misurata per il task specifico e il tempo di esecuzione totale. La scelta è stata basata su

quest'ultimo parametro, in particolare, delle tre soluzioni è stata scelta quella con il valore del tempo di esecuzione più basso. Di conseguenza, la prima soluzione è stata quella selezionata.

3.4 I file pcap

Come già detto in precedenza, i pacchetti verranno salvati in file del formato pcap. In particolare, ad ogni file viene associato una e una sola porta presente nella lista delle porte da monitorare, e, se il pacchetto passa per quella determinata porta, allora verrà salvato nel corrispondente file. Se, invece, il pacchetto passa per una porta non presente nella lista, allora verrà inserito in un file comune dove vengono salvati tutti questi tipi di pacchetti.

I nomi dei file, di conseguenza, seguiranno la seguente regola: nel primo caso i file si chiameranno “port\$PORTA.pcap” dove \$PORTA è il numero della porta, e per il secondo caso “portother.pcap”. Questa operazione viene fatta per ogni partizione temporale, rappresentata anch'essa da un numero, creata come folder di nome “partition\$PART”, dove \$PART è il numero della partizione, dove i file descritti precedentemente verranno inseriti.

3.4.1 Descrizione del filtro

Un primo chiarimento deve essere fatto sul filtro dei pacchetti. Come detto nel capitolo precedente, la soluzione scelta è ricaduta sulla prima, e di conseguenza il codice del filtro risiede nella funzione `apply_filter()`, ma, ovviamente, essendo solo un prototipo, differisce dall'implementazione finale. Il motivo consiste nel fatto che non è sufficiente controllare se il pacchetto ha nel proprio campo `port` una delle porte presenti nella lista, di conseguenza il codice del prototipo (che, ovviamente, implementava questa soluzione) non andava bene.

L'idea alla base del corretto monitoraggio delle porte consiste nel fatto che, siccome le porte che bisogna monitorare fanno riferimento ai servizi presenti sulla Vulnbox, i pacchetti da tenere sotto controllo dovranno avere ip della macchina e porta da monitorare nella stessa direzione, o, in altre parole, devono contenere insieme ip e porta nel campo `source` oppure insieme ip e porta nel campo `destination` dei rispettivi

protocolli. Se il pacchetto verifica una delle due condizioni, allora verrà salvato nel file “port\$PORTA.pcap” con il rispettivo numero di porta presente nel pacchetto.

Questo, infatti, permette di suddividere i pacchetti importanti, che devono essere monitorati perché contenenti possibili exploit e flag, da quelli che non hanno necessità di essere controllati, ovvero tutti quei pacchetti che verranno salvati in “otherport.pcap”.

3.5 Time split

Per fare lo split del traffico è stato impostato un intervallo temporale adeguato in modo tale che sia sufficientemente ampio per avere intere sessioni TCP da analizzare per cercare di identificare integralmente tentativi di exploit, ma non troppo lungo in modo tale che, grazie ai pcap, sia possibile identificare tempestivamente un exploit e il servizio compromesso e creare velocemente una patch per risolvere le vulnerabilità. Il valore scelto è stato di cinque minuti.

3.5.1 Implementazione

Nel codice del programma, per risolvere questo problema viene usato un sistema di segnali: infatti viene usata specifica system call, ovvero `alarm()`, che permette al kernel di generare il segnale `SIGALRM` dopo il tempo specificato come argomento della funzione, che viene catturato dal processo utente e verrà gestito da un handler. Nel codice, l’handler in questione si chiama `break_time`, che, una volta invocato, incrementerà il contatore `partition`, una variabile che identifica il numero della partizione temporale, e richiamerà la funzione `send_an_flush_file()` che gestirà la chiusura dei file e l’invio di questi alle altre macchine, ma questo verrà spiegato nel dettaglio nei capitoli successivi.

Sono state sviluppate, e successivamente abbandonate in quanto la soluzione sopracitata risulta molto più semplice, dei metodi alternativi per calcolo del tempo:

1. Creazione un thread ad-hoc per il calcolo del tempo. L’idea consiste nel sospendere il thread per il tempo specificato (cinque minuti) e riprendere successivamente invocando nuovamente la funzione bloccante. In questo modo si evitano imprecisioni dovute alla schedulazione e il calcolo, di conseguenza

diventa molto preciso. Errori dovuti alla schedulazione vengono ignorati perché trascurabili.

2. Sostituire nel codice la funzione `pcap_loop` con un ciclo `while` dove viene invocata la funzione con `pcap_next`, che invece ritorna un singolo pacchetto. Di default questa funzione è bloccante e di conseguenza attende l'arrivo di un pacchetto, ma è possibile attraverso un argomento della funzione di disattivare l'attesa e ritornare `NULL` nel caso in cui, al momento dell'invocazione, l'interfaccia non cattura alcun pacchetto. Se la funzione non ritorna `NULL`, significa che ritorna il pacchetto catturato e di conseguenza viene invocata la funzione `apply_filter`. All'interno del ciclo, inoltre, si implementa un semplice controllo per il calcolo del tempo, ovvero attraverso la differenza tra il tempo corrente e il tempo dell'ultimo time split, e, se questa differenza supera o uguaglia i cinque minuti, allora verrà aggiornato il tempo di riferimento come ultimo time split e verranno inviati i pcap. Infine, per completezza, si potrebbe rendere atomico il codice all'interno della condizione (invio dei file), in modo da evitare di interrompere il processo (dovuto alla schedulazione) e quindi avere una migliore stima nel calcolo della differenza di tempo (questa aggiunta può essere comunque evitata perché l'errore risulterebbe trascurabile).

3.6 Invio del file

Questa parte del programma si occupa dell'invio dei file pcap dalla macchina su cui girerà il programma (ovvero la Vulnbox) alle macchine su cui verrà fatta l'analisi dei file. La funzione che gestisce l'invio viene invocata all'interno dell'handler del segnale `SIGALRM`, invocata ogni 5 minuti, e alla terminazione del programma, affinché venga inviato anche in chiusura del programma così da evitare la perdere degli eventuali pacchetti salvati nei file.

3.6.1 Tool

Esistono molti modi per inviare un file da una macchina all'altra. Uno di essi, ad esempio, prevede l'utilizzo dei socket: infatti è possibile aprire un socket sullo sniffer, a cui verranno passati i file pcap, e uno sull'analizzatore, che rimarrà in attesa dei file.

Senza entrare nei dettagli di come sarebbe potuto essere il programma se implementato, il vantaggio di questa soluzione sarebbe risultato nel fatto che tutta la parte di gestione dell'invio dei file sarebbe stata scritta tutta su un unico programma, mentre lo svantaggio sarebbe consistito, invece, nella necessità di creare un programma apposito sarebbe stato sul ricevente che apriva a sua volta il socket su una porta predefinita. In questo si è scelto di usare un approccio diverso, seguendo la filosofia di Unix secondo cui si utilizzano diversi tool che vengono messi insieme per creare un programma unici: si è deciso di usare un tool apposito per l'invio dei file, ovvero rsync.

3.6.2 Ipotesi

Ecco presentate alcune delle soluzioni possibili per l'invio dei file, tra cui quella effettivamente implementata nel programma, analizzandone i vantaggi e gli svantaggi.

3.6.2.1 Attesa dell'invio dei pacchetti

Non implementata. Una possibile soluzione consiste nell'attesa da parte del programma che i file vengano inviati completamente, per poi riprendere la scrittura dei pacchetti, riaprendo gli stessi file in scrittura.

La possibile implementazione di questa soluzione consisterebbe nel creare un task apposito, che potrebbe essere un processo, usando `fork()`, oppure un thread, per inviare i file e poi richiamare una funzione per l'attesa del task da parte del programma principale, usando `wait()` per la `fork()` oppure `pthread_join()` per i thread se, ad esempio, si usa POSIX, e catturare lo stato di terminazione del processo o thread in modo da valutare la gestione dei file e il conseguente comportamento del programma in caso di errore nell'invio.

Vantaggi:

1. Soluzione semplice;
2. Utilizzo di un solo file per porta per il dump dei pacchetti.

3. I file non sono soggetti a possibili race condition, in quanto la scrittura di questi avverrebbe solo *dopo* essere stati inviati completamente.

Svantaggi

1. L'attesa del completamento dell'invio dei file compromette la cattura di alcuni pacchetti. Infatti se il programma principale attende il task per la gestione dell'invio dei file, i pacchetti che arriverebbero durante l'invio non verrebbero processati, con la possibilità che vengano scartati e perduti. Non si avrebbe la certezza della perdita, ma se il rate di arrivo dei pacchetti nella coda dei pacchetti fosse elevato allora si riempirebbe il buffer molto velocemente e i pacchetti verrebbero persi, senza essere consegnati al processo utente. La perdita di pacchetti non è accettabile e pertanto non è una buona soluzione.

In conclusione, la soluzione risulta non efficiente.

3.6.2.2 Creazione continua di file (e successiva cancellazione)

Non implementata. Questa soluzione prevede di creare continuamente file nuovi scrivendo i pacchetti ad ogni partizione temporale. Per non appesantire troppo il disco, si adotta un procedimento di cancellazione dei file più vecchi, secondo diverse modalità, quali potrebbero essere, ad esempio, sfruttando finestre temporali oppure tenendo un numero massimo di file aperti e cancellare solo il più vecchio.

Vantaggi:

1. Molto affidabile: in caso di errore dell'invio del file, è possibile recuperare i file manualmente.

Svantaggi

1. Appesantimento dell'host dal punto di vista della memoria secondaria (disco).
2. Se prevede cancellazione, si possono perdere comunque i file, ma si possono recuperare in un secondo momento se si riesce ad accorgersi in tempo dell'errore.

Soluzione affidabile ma non efficiente.

3.6.2.3 Creazione dei file e concorrenza

Questa soluzione prevede di scrivere i file su una prima cartella (cartella 0) i pacchetti per la prima partizione temporale, e una volta raggiunti i cinque minuti, il programma chiude i file, genera il segnale di invio file e apre i file nella seconda cartella (cartella 1) e il programma continua la scrittura dei pacchetti su questi file; una volta scaduti i cinque minuti, il programma, diversamente da prima, chiude i file della seconda cartella (cartella 1), genera il segnale di invio file e apre i file nella prima cartella (cartella 0) e il programma continua la scrittura dei pacchetti su questi file. Di conseguenza, l'invio dei file (rsync) e la scrittura dei pacchetti che arrivano durante l'invio sono concorrenti.

Vantaggio:

1. Vantaggi dovuti alla concorrenza e al parallelismo dei processi (rsync e sniffer);
2. Soluzione che genera il minor numero di file permettendo la concorrenza dei processi;
3. modello molto snello e elegante

Svantaggio:

1. In caso di errore, non è possibile recuperare i file.

Soluzione adottata.

3.6.2.4 Soluzione intermedia

Per ovviare ai problemi illustrati nei 5.2.2 e 5.2.3 si può adottare una soluzione intermedia, ovvero creare più di due file, e adottare la scrittura come nella 5.2.3, ovvero scrivere nel primo file, chiuderlo, e durante l'invio scrivere sul secondo, poi chiuderlo, inviarlo e scrivere sul terzo, e così via.

Questa soluzione sfrutta il vantaggio dell'invio immediato e non appesantisce troppo l'hard disk della Vulnbox e allo stesso tempo se si verifica un errore è possibile, se ci si accorge in tempo, riuscire a recuperare manualmente i file e a inviarli. Questa soluzione risulta un trade-off tra le due soluzioni sopracitate, ma per fare in modo che sia tale, bisogna trovare, ovviamente, il numero giusto di file da tenere aperti.

Questa soluzione non è stata implementata.

3.6.2.5 Gestione dell'errore

In caso di errore nell'invio si crea con gzip un file compresso .gz dei file pcap. In questo è possibile salvare i file in modo tale che sia possibile recuperarli manualmente in un secondo momento (usando, ad esempio, ancora rsync manualmente per inviarli alla macchina per l'analisi). Questa soluzione è stata integrata con quella precedente.

3.6.3 Rsync

Rsync è un programma sviluppato per trasferire e sincronizzare in modo efficiente i file tra gli host remoti della rete, attraverso il controllo del timestamp e la dimensione dei file. Originariamente sviluppato esclusivamente per i sistemi Unix, questo tool risulta uno dei tanti tool standard per i sistemi Unix-like e non di rado lo si trova già installato nel sistema. Esistono, comunque, implementazioni anche per altri sistemi, come, ad esempio, Windows. L'algoritmo di rsync è un algoritmo di tipo delta encoding, che permette di minimizzare il trasferimento di dati e quindi l'utilizzo della rete. Zlib può essere utilizzato opzionalmente per compressione dei file da trasferire, e SSH oppure stunnel per la sicurezza dei dati.

Rsync è tipicamente utilizzato per la sincronizzazione di file e directory tra due sistemi diversi. Per esempio, se il comando `rsync local-file user@remote-host:remote-file` è eseguito, rsync usufruirà dei servizi offerti da SSH per connettersi come *user* a *remote-host*. Una volta connesso, invocherà rsync dell'host remoto e successivamente i due programmi determineranno quale parte del file necessita di essere trasferito attraverso la connessione aperta. Rsync può anche operare in modalità daemon, servendo e ricevendo file nel protocollo nativo di rsync.

In questa tesi, è stato usato per l'invio dei file secondo la modalità ssh e non demone, in quanto quella demone prevede la configurazione manuale sull'host target, mentre con la prima è sufficiente installare openssh-client, ma questa modalità, come verrà illustrato successivamente, crea una problema spiegato in seguito.

3.7 initpcap.sh

initpcap.sh è il software che deve essere avviato per far funzionare lo sniffer e svolge molte delle funzioni definite nelle specifiche, oltre ad invocare il programma pcapevolve, spiegato nel dettaglio in seguito. Questo codice è un BASH script e il

motivo principale è dovuto al fatto che il programma necessitava di utilizzare tool di Unix, come ad esempio rsync, e risultava comodo questo tipo di script.

3.7.1 Il problema delle password

Ovviamente l'uso di rsync tramite ssh è possibile se ci si connette ad un host autorizzato: conoscendo utente, nome dell'host (oppure ip) e la relativa password, non si incontrano problemi particolari. In ogni caso, l'inserimento della password è *necessaria* ogni qualvolta che si fa uso di rsync per inviare i file, e questo ha come effetto che ogni cinque minuti si debba continuamente inserire la password per permettere l'autorizzazione a rsync di copiare i file sull'host remoto, e questo risulta estremamente scomodo e tedioso per l'utente e può portare a errori se ci si dimentica di inserirla, con conseguente perdita dei file.

3.7.2 Implementazione

3.7.2.1 Test degli argomenti da riga di comando

Per verificare che gli argomenti della riga di comando siano corretti (ovvero che la folder di destinazione, utente e host esistano e siano attivi), si usa rsync inviando un file di prova (testfile.txt): se non si verificano problemi e il programma invia il file correttamente, significa che gli argomenti inseriti sono corretti e il programma può proseguire la sua esecuzione senza problemi.

3.7.2.2 Public key authentication di ssh

In quanto rsync usa ssh per inviare i file in modo sicuro, a sua volta ssh utilizza la public key authentication in modo da autenticare l'host in base alla crittografia asimmetrica e senza l'inserimento della password ogni volta che ssh si connette con l'host

tool di ssh: ssh-keygen e ssh-copy-id. Con ssh-keygen viene generata la coppia chiave privata-pubblica usando un passphrase (nel programma non si usa perché non necessario) per proteggere la chiave privata in caso di intrusione (attraverso una funzione di hashing). Con ssh-copy-id viene passata al target host la chiave pubblica

attraverso un canale crittografato e l'host salverà questa chiave. Per verificare però che l'host client possa fare login senza inserire sempre la password, bisogna che il client inserisca una sola volta la password e, una volta confermata, il server verificherà che l'host potrà connettersi con quel username e potrà connettersi ogni volta senza password in quanto con il sistema di crittografia asimmetrico, la chiave pubblica dell'host remoto corrisponderà a quella privata dell'host locale, confermando l'identità di quest'ultimo ad ogni connessione.

3.7.2.3 Controllo sull'invio della password

Una volta che la password è stata copiata sull'host remoto (e quindi ssh-pa), bisogna verificare che sia stata effettivamente salvata correttamente. Per fare ciò, si invia nuovamente un file di prova, ma invece che inviare textfile.txt, inviamo il file di configurazione delle porte portlist.conf, che ovvero il file che verrà usato nel programma di analisi per verificare le porte da monitorare. Per controllare se la password è stata salvata correttamente, basta vedere se chiede l'inserimento della password: in caso affermativo, si è verificato un errore e termina, altrimenti si prosegue con l'esecuzione.

3.7.2.4 pcap evolve.c

Il programma PcapEvolve è stato sviluppato in C e risulta specifico per i sistemi Unix in quanto, come si può osservare nel codice, vengono usate la libreria POSIX e unistd.h.

Logicamente il programma può essere suddiviso nelle seguenti parti:

1. lettura del file portlist.conf: il file viene letto secondo la sintassi definita nel capitolo 3.1 e le porte salvate nella struttura/lista ports.
2. apertura file, creazione e applicazione di filtri: vengono aperti tutti i file sui quali verranno scritti i pacchetti e vengono generati e applicati i due filter program dalle filter expression "tcp" e "ip src host \$IP", dove \$IP è l'indirizzo IPv4 dell'host;
3. pcap_loop, apply_filter: con la funzione pcap_loop viene implementato il loop di attesa dei pacchetti e apply_filter è la funzione che applica il filtro dei

pacchetti e viene invocata ad ogni pacchetto ricevuto dal kernel (ovviamente dopo essere stato filtrato dai filtri BPF);

4. `handle_split_time`, questa funzione viene invocata dall'eccezione `ALARM`, generata ogni 5 minuti tramite la system call `alarm()`;
5. `flush and send file`, segnali: in questa parte vengono inviati i file tramite la generazione del segnale `USR1` al processo padre di `PcapEvolve`, ovvero `initpcap.sh`.

3.7.2.5 Attesa e ricezione dei segnali

Il programma, dopo aver avviato `PcapEvolve` in modalità demone, si pone in attesa con un ciclo infinito per attendere la ricezione di qualche segnale. Se il segnale ricevuto è un interrupt signal, allora termina il programma, altrimenti riceve il segnale `USR1`, che permette di invocare la callback progettata per l'invio dei file.

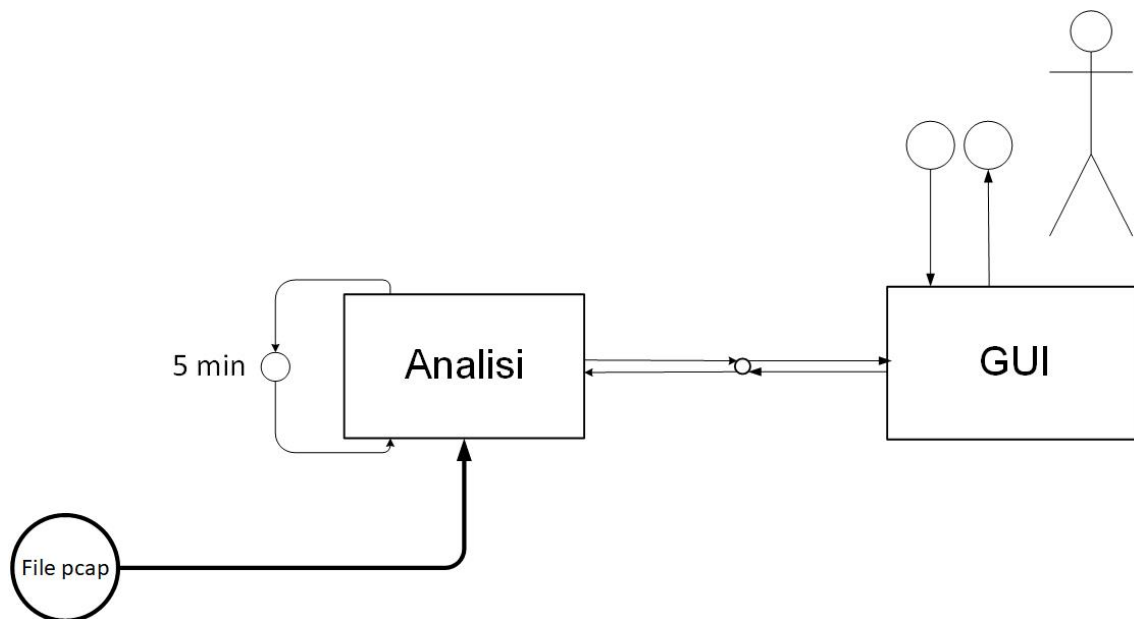
3.8 Testing

Per questa parte della tesi, il testing è stato effettuato simulando un HTTP server attraverso il comando `BASH python -m SimpleHTTPServer $PORT`, svincolandosi dalla necessità di dover installare un programma apposito, come ad esempio Apache. Per inviare i pacchetti a questo server, è stato sufficiente utilizzare un semplice browser, come ad esempio Google Chrome, per inviare al server delle richieste HTTP. Lo scopo di tutto questo consisteva nel verificare che lo sniffer riuscisse a catturare tutti i pacchetti e a smistarli nei file corretti secondo le porte specificate.

4 Seconda Parte: Lato client e analisi dei dati

Il programma di analisi si prefigge di gestire l'analisi dei file pcap attraverso l'utilizzo di Snort, in modo da inizializzarlo e configurarlo in modo corretto, e presentare un interfaccia grafica che visualizzi i file analizzati. I file da analizzare sono quelli inviati dalla Vulnbox da parte dello sniffer.

Ecco di seguito lo schema di questa parte del programma:



Il componente Analisi si suddivide in due parti: una parte “ricorsiva”, ovvero una parte che analizza in modo automatico i pcap della singola partizione ogni 5 minuti e si “sincronizza” col programma che gira sulla Vulnbox; e una parte “di attesa di richiesta”, ovvero che come un server si pone in attesa di una richiesta da parte della GUI sul file da analizzare e analizza su richiesta dell’utente (GUI). La Gui, dal canto suo, si basa sul controllo ad eventi, dove gli eventi sono generati dall’utente.

Il programma è stato sviluppato in Python.

4.1 Snort

Snort è un NIDS (Network Intrusion Detection System) open-source, un software per l’analisi del traffico di rete in tempo reale e la prevenzione da attacchi malware.

Snort fornisce una grande quantità di opzioni da riga di comando che l’utente può usufruire ed è molto flessibile. Snort può essere configurato per essere eseguito nelle sue tre modalità principali:

- modalità Sniffer, che semplicemente legge i pacchetti dalla rete e li stampa in un flusso continuo sulla console.
- modalità Packet Logger, che scrive su disco i pacchetti.

- modalità Network Intrusion Detection System (NIDS) mode, che svolge l'attività di analisi, prevenzione e investigazione sul traffico di rete.

Quest'ultima modalità è la più complessa e avente molte opzioni di configurazione.

Snort basa le sue ricerche su delle regole, le Snort rules, alcune delle quali vengono scaricate e configurate in fase di installazione, mentre altre possono essere scaricate dal sito ufficiale di Snort. Snort presenta numerosi file contenenti rules, e tra questi ci sono quelle usate per la tesi, ovvero i file `exploit-kit.rules`, `community-exploit.rules` e `exploit.rules`, che contengono le regole per individuare gli exploit nel traffico. Inoltre, è possibile configurare delle regole personalizzate, scrivendole nel file `local.rules`.

Ovviamente, questi file dovranno essere specificati nel file di configurazione `/etc/snort/snort.conf`.

4.1.1 Le rules in breve

Le rules sono le regole di Snort grazie alle quali il software potrà trovare nei pacchetti i pattern specificati nelle regole. Vengono scritti in file `.rules` nella cartella `rules/` in `/etc/snort/`. Da questi file Snort leggerà le regole all'avvio e leggerà esclusivamente quelle contenuti nei file specificati nel file di configurazione. Esistono delle regole già scritte dagli sviluppatori di Snort, scaricate automaticamente durante l'installazione di Snort, oppure si possono scaricare quelle sviluppate dalla community al seguente URL: <https://www.snort.org/faq/what-are-community-rules>.

Le Snort rules sono divise in due sezioni logiche, il rule header e i rule options.

Il rule header contiene l'azione della rules, il protocollo, l'indirizzo IP e port del mittente e del destinatario, e la maschera di rete. Le rule options contengono messaggi di alert e informazioni su quale parte del pacchetto dovrebbe essere ispezionata per determinare se l'azione dovrebbe essere intrapresa. La figura sottostante riporta un esempio di rules:

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

Figura 3: Semplice Regola di Snort

Qui di seguito verranno descritte brevemente le opzioni delle rule header dell'esempio sopra riportato:

- **alert:** rappresenta le rule actions, ovvero l'azione da intraprendere nel caso in cui si identifica un pacchetto che contiene gli attributi definiti nel content. In questo caso, alert mostra un messaggio di allerta e scrive il pacchetto su un file, ma esistono altre azioni diverse da quella precedente, quali log, che scrive il pacchetto su un pcap senza generare nessun alert, pass, che lo ignora, e molte altre ancora.
- **tcp:** indica il protocollo su cui Snort va ad indagare. Per il momento, esistono solo quattro protocolli che attualmente Snort analizza in cerca di attività sospette: TCP, UDP, ICMP, e IP.
- **Any any:** indica l'ip e la porta sorgente da analizzare.
- **192.168.1.0/24 111:** indica ip e porta destinatario.
- **→ :** indica l'orientamento, o la direzione, del traffico di rete ai cui farà riferimento la rule. L'indirizzo IP e il numero della porta sul lato sinistro dell'operatore di direzione fanno riferimento al traffico proveniente dal mittente. Invece, le informazioni sull'indirizzo IP e porta a destra dell'operatore riguardano il destinatario.

Le Rule options formano il cuore dell'intrusion detection engine di Snort, combinando facilità d'uso con flessibilità. Nell'esempio precedente:

- **msg:** Questa opzione indica al logging e alerting engine il messaggio da stampare insieme all'azione da intraprendere, quale il dump dei pacchetti oppure un alert.
- **Content:** La keyword content è una delle più importanti feature di Snort. Permette all'utente di settare le rules che cercano all'interno dei payload dei pacchetti uno specifico contenuto o pattern e causare un evento basato sull'azione definita nelle rule header.

La trattazione in dettaglio delle Snort rules esula della tesi. Questa spiegazione si prefigge solamente di illustrare il concetto generale ed il funzionamento di base.

Per ulteriori dettagli, fare riferimento al manuale di Snort.

4.1.2 local.rules e flags

local.rules è un file completamente personalizzabile, in cui viene data l'opportunità all'utente di scrivere delle proprie regole custom, le quali verranno lette da Snort e, se scritte correttamente, verranno importate e configurate all'interno del software.

Proprio in questo file sono state inserite le regole sulle flag, da ora in avanti chiamate flag rules: queste regole permetteranno a Snort di riconoscere nei pacchetti catturati le possibili flag inserite. Per scrivere queste regole bisogna innanzitutto conoscere la sintassi di una flag. Le flag in una competizione CTF possono seguire diversi pattern, e qui di seguito verranno illustrate tutte le possibili combinazioni. Le flag possono trovarsi come flag{...}, ctf{...}, nome_ctf{...}, dove le lettere possono essere in maiuscolo o minuscolo. Per generalizzare sono state considerate che le parentesi {} possano essere sostituite da [] o (), e che dopo flag/ctf/nome_ctf può essere seguito dalla data della CTF in questione, quindi flagDATA{...}. Per definire questa sintassi nelle flag rules è stata usata l'opzione pcre, che permette di scrivere il content come espressione regolare del Perl. Non verrà illustrata la sintassi del Perl, ma per ulteriori informazioni leggere le regex di Perl.

Di seguito verranno riportate le flag rules scritte nel file local.rules:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"CTF Flag Incoming"; pcre:"/(flag|ctf|$CTF)[\{[\[?.[\]\]]?/i"; sid: 1000000001; rev:1; priority:1;)

alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"CTF Flag Outgoing"; pcre:"/(flag|ctf|$CTF)[\{[\[?.[\]\]]?/i"; sid:1000000002; rev:2; priority:1;)
```

Figura 4: le flag rules

4.3 Il file di configurazione di Snort

Per configurare in modo personalizzato Snort, è necessario conoscere e modificare le regole e la sintassi del file di configurazione di default. Il file può essere scaricato dal sito ufficiale oppure viene scaricato durante l'installazione del programma come etc/snort/snort.conf. Questa tesi esula dalla trattazione in dettaglio del file di

configurazione, ma verrà comunque fornita una breve descrizione delle componenti principali:

- La keyword `include` permette di includere altri file di configurazione nel file `snort.conf` indicato nella riga di comando di Snort.
- Variabili: Esistono tre tipi di variabili definite in Snort: `var`, `portvar`, `ipvar`. Gli esempi riportati chiariscono bene il funzionamento senza spiegazioni aggiuntive:

```
var RULES_PATH rules/

portvar MY_PORTS [22,80,1024:1050]

ipvar MY_NET [192.168.1.0/24,10.1.1.0/24]

alert tcp any any -> $MY_NET $MY_PORTS (flags:S; msg:"SYN
packet";)

include $RULE_PATH/example.rule
```

Figura 5: Semplici esempi di variabili in snort.conf

- Preprocessori: Permettono di estendere le funzionalità di Snort permettendo agli utenti e programmatori di scartare plugins modulari abbastanza facilmente in Snort. Il codice dei preprocessori viene eseguito prima di richiamare il detection engine, ma dopo che il pacchetto sia stato decodificato.
- Output modules: Permettono a Snort di essere molto flessibile nella formattazione e presentazione dell'output. Gli output modules sono eseguiti quando vengono richiamati i alert o logging subsystems di Snort, dopo i preprocessors e detection engine. I moduli sono caricati runtime specificando la keyword dell'output nel file di configurazione.
- Statistiche: Snort fornisce statistiche sulle performance delle rules e dei preprocessori. Ognuna di queste richiede una semplice opzione di configurazione in `snort.conf` e Snort stamperà statistiche al termine dell'esecuzione.

Per ulteriori dettagli, consultare il manuale di Snort.

4.4 Analizzatore

Il primo componente, l'analizzatore, non è altro che un programma che fa da wrapper a Snort e per mette di invocarlo mettendo i automatico le giuste configurazioni per avviare Snort. Come già accennato, lo script si suddivide in due moduli, di cui uno principale, ovvero `analisi.py`, e uno di supporto, ovvero `analyzer.py`.

La prima funzione principale che l'analizzatore svolge consiste nell'analisi "ricorsiva", ovvero che esegue in automatico, ogni cinque minuti, l'analisi di tutti file appartenenti ad una partizione temporale, e prosegue con la successiva solo se esiste almeno un file in quella corrente. Il motivo di tale scelta consiste nel fatto che, se non esiste nessun file, allora il programma che risiede sulla Vulnbox non ha ancora inviato o generato i file di quella partizione temporale, e questo permette all'applicazione di sincronizzarsi con quella che risiede sulla Vulnbox. La seconda funzione principale, invece, gestisce le richieste dell'interfaccia grafica, che consistono di due tipi: analisi di tutti file di una partizione temporale specificata o di un singolo file con partizione e porta specificata.

Inoltre, come terza funzione principale, vengono eseguiti tutta una serie di controlli preliminari sugli argomenti da riga di comando e su Snort in modo che non l'uso scorretto dell'utente non vada a compromettere il funzionamento corretto del software.

4.4.1 Il codice

Ovviamente, come già detto precedentemente, è possibile consultare il codice su Github e non verrà riportato sulla tesi. Il codice si suddivide in due moduli python: `analyzer.py`, un modulo progettato ad hoc per il programma principale, e il programma principale `analisi.py` da lanciare e che usufruisce del modulo `analyzer`.

4.4.1.1 Classe PcapInfo

Questa classe non ha un utilizzo al di fuori del modulo, viene usata esclusivamente all'interno della classe `Analyzer`, e serve per costruire il path del file pcap, specificando solo la partizione oppure la partizione e porta e controllare l'esistenza del file, generando l'eccezione nel caso di non esistenza del file.

4.4.1.2 Classe Analyzer

La classe Analyzer si prefigge come scopo di provvedere ai dovuti controlli per il corretto funzionamento del programma ed effettuare l'analisi ricorsiva e su richiesta della GUI. Ecco di seguito i vari metodi:

- `testing_conf` : questo metodo permette di configurare in modo corretto Snort, oltre ad effettuare numerose serie di controlli per il corretto funzionamento dell'applicazione. Ogni controllo che non viene superato genera un'eccezione. Ecco di seguito i controlli effettuati:
 1. Controlla se la cartella specificata in `PCAP_PATH` è corretta, ovvero contiene i pcap che sono stati inviati;
 2. Controlla la presenza del file contenente l'elenco delle porte, ovvero nel path `PATH_PCAP/portlist.conf`.
 3. Modifica il file `local.rules` e inserisce il nome della ctf specificata nella riga di comando al posto di `$CTF` che si trova all'interno del file.
 4. Viene fatto eseguire Snort in modalità test in modo da verificare la corretta configurazione del software. Se non si verificano problemi, Snort termina correttamente e si prosegue, altrimenti il programma termina e l'utente dovrà aprire il file di configurazione manualmente e modificarlo di conseguenza.
- `rec_analysis`: metodo che si occupa della richiesta di analisi su richiesta: una volta passati porta e numero di partizioni, esegue la richiesta invocando `__analysis` secondo le modalità descritte in precedenza.
- `req_analysis`: metodo che esegue l'analisi ricorsiva dato la partizione e invocando il metodo `__analysis()` secondo le modalità descritte in precedenza.
- `__analysis`: specificato il pcap da analizzare, invoca Snort per l'analisi che, a sua volta, fa il dump dei pacchetti in file pcap che matchano le rules configurate. Siccome in Snort non è possibile personalizzare il nome dei file generati, allora viene modificato seguendo questa sintassi:
`PATH_PCAP/fpartition$PARTITION/fport$PORT.pcap`.

4.4.2 I thread

L'analizzatore si suddivide in due thread, uno che si occupa dell'analisi ricorsiva automatica, mentre l'altro si occupa di gestire le richieste dell'interfaccia grafica.

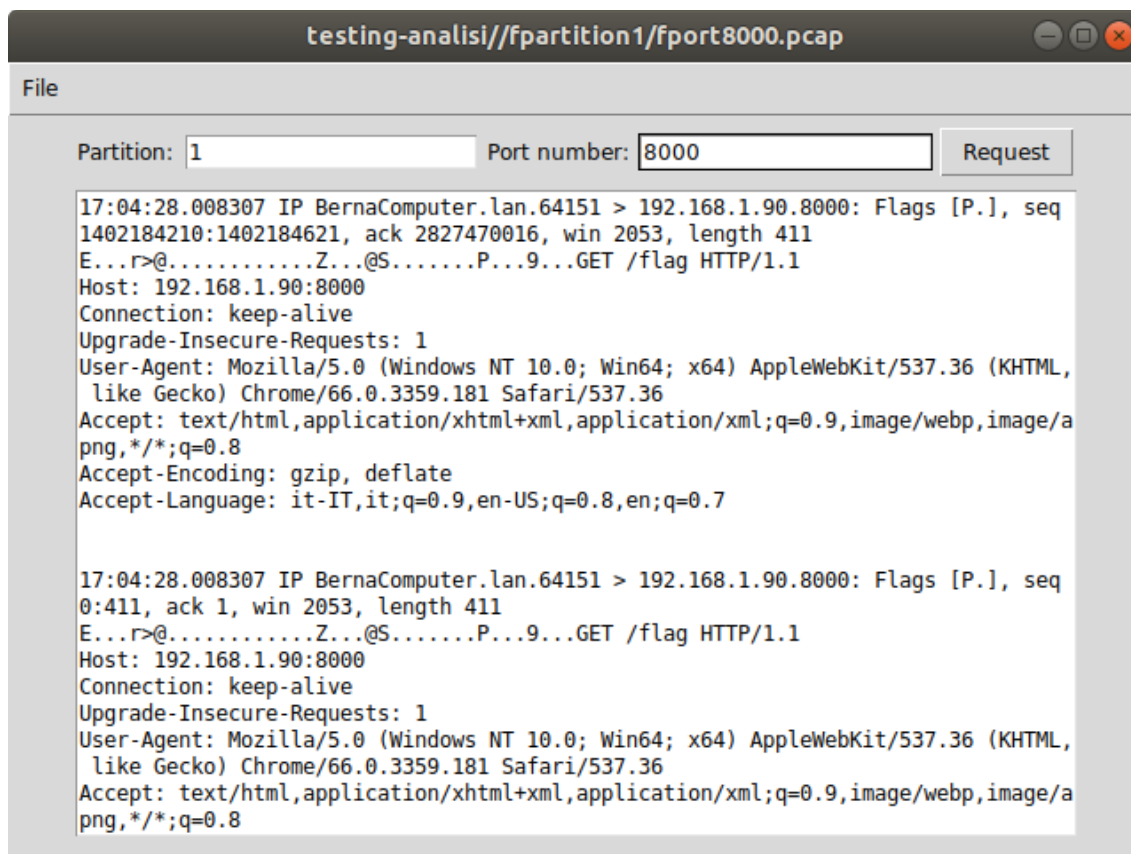
Nel programma intero, in realtà, il programma prevede tre thread, di cui due illustrati sopra e il terzo gestisce la GUI. Tra di essi esiste un meccanismo di sincronizzazione: infatti, anche se non esistono variabili comuni e quindi nessun problema di race condition, è necessario che il recursive thread (il nome del thread che gestisce l'analisi automatica ricorsiva) deve essere invocato prima dell'esecuzione degli altri due thread, perché proprio in questo viene eseguito il metodo `testing_config()` e deve essere eseguito prima di qualsiasi altra funzione del programma, in modo tale se si verifica qualsiasi problema il programma può terminare. La sincronizzazione è stata raggiunta utilizzando un oggetto chiamato Event, che dispone di una flag interna che venga settata a True quando si invoca `set()` e a False con il metodo `clear()`. Se la flag è False, il thread sospenderà la sua esecuzione, mentre la riprende quando diventa True. In questo modo, è possibile che il recursive thread, una volta terminato correttamente il metodo `testing_conf`, permetterà agli altri due thread di proseguire.

La libreria di Python che permette di implementare i multithreading si chiama *threading*.

4.5 GUI

L'interfaccia grafica permette all'utente di selezionare un pcap e visualizzare il contenuto di tutti i pacchetti scritti nei file pcap.

4.5.1 Descrizione generale e uso



L'interfaccia presenta un aspetto molto minimale e anche l'utilizzo risulta intuitivo: due caselle di testo dove inserire il numero di partizione temporale e la porta, e a destra il Button Request, grazie al quale viene fatta la richiesta all'analizzatore di analizzare i pcap in base ai dati inseriti.

Nella casella di testo sottostante verrà visualizzato tutto il contenuto del file. Siccome il file segue il formato pcap, parzialmente di esso in formato binario, di conseguenza non è possibile semplicemente leggerlo, ma bisognerà invocare un programma che sia in grado di leggere questo tipo di file. Per questo motivo viene invocato tcpdump, che, prendendo in input il file, scriverà sull'output il contenuto del file letto correttamente e questo verrà visualizzato all'interno della casella di testo in questione.

4.5.2 Funzionalità dell'interfaccia

L'interfaccia permette di svolgere due tipi di analisi: analisi del singolo file e analisi dell'intera cartella. Per analizzare l'intera cartella basta selezionare nel campo Partition

il numero della partizione temporale desiderata e lasciare vuoto il campo Port; per analizzare il singolo file, bisogna inserire sia il numero della partizione temporale, sia il numero di porta. Queste informazioni sono sufficienti per selezionare i/il pcap selezionato/i. Una volta inserite, si preme il bottone per fare la richiesta. Ovviamente viene generato un pop-up di errore nel caso venga premuto il pulsante senza inserire nessun dato nei campi oppure si inserisce un valore della porta che non fa parte della lista portlist.conf.

Per quanto riguarda la visualizzazione, l'unica opzione di visualizzazione consentita è quella di un singolo pcap selezionato e non tutti i file in quanto l'output risulta molto confuso e controproducente ai fini dell'analisi dei pacchetti.

4.5.3 Implementazione

Di seguito verrà illustrato brevemente le classi, i metodi e le librerie usate per il codice.

4.5.3.1 Libreria

La libreria usata per sviluppare l'interfaccia grafica è Tkinter, il modulo standard di Python per qualsiasi GUI che si voglia sviluppare.

4.5.3.2 La classe GUI

La classe GUI è composta dal costruttore e da metodi privati. Il costruttore contiene all'interno gli oggetti e le funzioni importate da Tkinter per creare gli oggetti grafici visibili a schermo. Ad esempio, Button è l'oggetto che rappresenta un pulsante grafico, Entry è la casella di testo, tk.mainloop() è il metodo che ascolta gli eventi, e così via.

I metodi invece della classe sono metodi privati: infatti, come si può osservare dal codice, questi metodi sono callback invocate nel momento in cui si verificano determinati eventi, ed è proprio per questo motivo che sono privati, perché essendo proprio delle callback, non è necessario che siano raggiungibili all'infuori della classe.

Brevemente verranno descritti questi metodi qui sotto:

- `do_quit`: questo metodo apre un pop-up di richiesta di uscire dal programma e in caso affermativo il processo termina. Si attiva quando si preme il pulsante `file→quit`.
- `do_open`: si attiva quando si preme `file→open...` e permette di aprire un file pcap (solo pcap) e di visualizzarlo attraverso `tcpdump`.
- `request`: questo è il metodo che viene invocato quando si preme il pulsante Request. In caso di errore, il metodo viene sempre terminato e compare un pop-up di errore. Le funzionalità implementate in questo metodo si possono riassumere quanto segue:
 1. Controlla che il campo Partition non sia vuoto.
 2. Controlla che nei campi Partition e Port venga sempre inserito un numero e non un carattere diverso.
 3. Controlla che il valore inserito nel campo Port compaia nella lista delle porte del file `portlist.conf`.
 4. Controlla, in base alla porta e partizione selezionata, se il file esiste: in caso affermativo, non è necessaria l'analisi da parte di Snort perché già eseguita, in caso contrario l'analisi deve essere effettuata e si passa al punto successivo.
 5. Invia la partizione e la porta selezionata al request thread e, se non si verificano errori, prosegue l'esecuzione visualizzando il file analizzato, altrimenti la callback termina.

4.6 Testing

Per quanto riguarda la fase di testing di questa parte del programma, la parte più significativa riguarda il test di Snort: infatti, era necessario trovare la configurazione corretta di Snort attraverso il file di configurazione e verificare che venissero effettivamente applicate le regole di Snort, quelle già preconfigurate sugli exploit e quelle customizzate sulle flag. Per fare ciò, sono stati inviati da parte dello sniffer alcuni file pcap contenenti alcuni pacchetti, di cui alcuni senza nulla di particolare, mentre altri

contenenti flag ed exploit. Se Snort termina in modo corretto, il programma mostra a schermo il risultato della sua individuazione.

5 Limiti

In questo capitolo verranno affrontati quelli che sono i limiti che l'applicazione presenta, dovuti principalmente alle scelte nella progettazione e implementazione del software.

5.1 Lista delle porte non dinamica

La lista delle porte da monitorare viene scritta in un file di configurazione, di conseguenza la lista verrà letta soltanto all'inizio del programma. In questo modo, dopo che il programma è stato avviato, in particolare durante lo sniffing e il logging dei pacchetti, non è più possibile cambiare le porte, o perlomeno non è possibile che il programma aggiorni la nuova lista creata dall'utente. Questo è dovuto al fatto che il programma legge una sola volta, all'inizio dell'esecuzione, il file di configurazione. Per cambiare la lista, è necessario interrompere il processo e invocarlo nuovamente con un file di configurazione modificato di conseguenza.

5.2 Apertura dei file pcap non dinamica

Questo limite deriva principalmente dall'idea secondo cui l'utente definisce nel file `portlist.conf` soltanto le porte che deve monitorare: infatti, se così fosse, sulle quelle porte passerà sempre qualche pacchetto, e quindi i file corrispondenti non saranno mai vuoti. Ma esiste comunque la possibilità, anche se remota, che per errore l'utente inserisca nella lista una porta non da monitorare e quindi implica che per essa non passa nessun pacchetto: in questo caso, il file corrispondente è vuoto ed è inutile. Nel programma, questo avvenimento è possibile, in quanto all'inizio del programma vengono aperti per la scrittura tutti i file per porta. Per ovviare a questo problema, è necessario che il file venga aperto solo nel caso per quella porta transiti un pacchetto, altrimenti non viene creato. In questo modo non verranno più creati file vuoti inutili.

5.3 Attesa dei processi invocanti: problemi di performance

Un limite rappresentato dal file di analisi consiste nel fatto che quando Snort viene invocato, l'analizzatore (il processo padre) attende la terminazione di Snort(processo figlio). Siccome Snort deve essere invocato per ogni singolo file, questo implica che nel momento in cui devono essere analizzati più file, i processi figli (Snort) risulteranno processati in serie e non concorrenti. Questo, ovviamente, comporta problemi di performance in quanto non vengono sfruttati i vantaggi della programmazione concorrente.