Indice

1) Introduzione 3

2) Rete: funzionamento e analisi del traffico 5

2.1) La suite TCP/IP 5

2.2) Cenni sulla sicurezza 7

2.3) L'importanza dell'analisi del traffico 10

3) Il progetto: CSD – DMAD 13

3.1) Obiettivi del progetto 13

4) Strumenti utilizzati e librerie 15

4.1) JDK/JRE 8.0-162 - Eclipse Oxygen 15

4.2) Jnetpcap – Winpcap/libpcap 16

4.3) Ini4j 16

4.4) Apache.commons.net 17

4.5) Java.util.regex.Pattern 18

4.6) Eclipse WindowBuilder 19

5) CSD – Capture Split and Dump 19

5.1) CSD: Progetto 19

5.2) CSD: la Classe Main() 22

5.3) CSD: la Classe CaptureCycle() 30

5.3.1) CaptureCycle() : il metodo startCycle() 31

5.3.2) CaptureCycle() : il metodo Split() 32

5.4) CSD: la Classe Dumper() 42

5.4.1) Dumper() : il metodo checkFTP() 42

5.4.2) Dumper() : il metodo SaveRemoteFile() 43

5.5) CSD: la Classe ThreadDumper() 45

5.6) CSD: la classe charAnimationPrinter 46

5.7) CSD: il File CONFIG.ini 47

6) DMAD – Download Merge Analyze and Dump 49

6.1) DMAD: il progetto 49

6.2) DMAD: la GUI (Graphic User Interface) 51

6.2.1) GUI: la funzione LoadIni() 52

6.2.2) GUI: la funzione SaveIni() 53

6.2.3) GUI: la funzione Check() 55

6.2.4) GUI: la funzione DownloadFiles() 56

6.2.5) GUI: la funzione MergeNofilter() 59

6.2.6) GUI: la funzione FilterAndMerge() 61

6.2.7) GUI: visualizzazione ed eliminazione dei file locali 66

6.2.8) GUI: il selettore FTPON/OFF 68

6.3) DMAD: la classe CaptureCycle() 69

6.4) DMAD: la classe Dumper() 69

6.5) DMAD: la classe CapMerger() 72

7) Introduzione all'uso 73

7.1) Introduzione all'uso - CSD 73

7.2) Introduzione all'uso - DMAD 79

8) Conclusioni e sviluppi futuri 85

1) Introduzione

Questo elaborato nasce con l'obiettivo di affrontare e risolvere alcune problematiche pratiche relative alla verifica della sicurezza di un calcolatore connesso alla rete (d'ora in poi vi si farà riferimento con il nome di "VulnBox", indicando una macchina che potrebbe ospitare servizi potenzialmente affetti da vulnerabilità), avente la possibilità di inviare e ricevere i diversi flussi di dati, secondo i protocolli previsti dallo standard TCP/IP. In particolare, questo progetto deriva da diverse esigenze relative alla cattura ed analisi del traffico di rete della VulnBox, per cui si è pensato di realizzare due applicazioni – CSD per la cattura su VulnBox e invio a server remoto; DMAD per l'analisi dei file di cattura (estensione ".cap") – usando il linguaggio Java.

La scelta è ricaduta su questo linguaggio principalmente per via della sua portabilità, e, salvo alcune eccezioni, per la sua indipendenza dal sistema operativo su cui si appoggia.

Con l'utilizzo delle librerie descritte in dettaglio nel capitolo 4, CSD si propone come software multipiattaforma, e propone le seguenti funzionalità:

1- Permette di gestire la configurazione della scansione da file .ini.

2- Cattura il traffico relativo alla VulnBox per N minuti, e lo salva in un file .cap.

3- Separa il traffico registrato per ogni porta specificata.

3b- A seconda della configurazione, applica un filtro al traffico registrato e salva i file che superano tale filtro, mantenendo una copia del dump non filtrato.

4- Invia i dump su server remoto.

Quindi l'applicazione client "DMAD" si occupa di offrire un servizio di merging e download dei file ottenuti, filtraggio opzionale dei pacchetti per porta, per periodo, o con un ulteriore filtro. Il software DMAD, inoltre, permette anche una visualizzazione di base del contenuto dei pacchetti rilevati nell'intervallo di interesse.

Nel capitolo 2 (cattura e analisi del traffico di rete) verrà fornita una panoramica di base sui concetti fondamentali inerenti la trasmissione dei pacchetti, al fine di introdurre le metodologie e i principali Protocolli di rete utilizzati nella creazione degli algoritmi necessari. Verrà inoltre introdotto il formato degli header contenenti le informazioni necessarie al trasferimento dei pacchetti in rete, con esempi relativi ai principali protocolli di rete previsti dai 4 livelli dello stack TCP/IP .

Verranno inoltre introdotte le principali problematiche relative alla sicurezza nella trasmissione e ricezione di dati in rete, con alcuni esempi riguardanti le principali tecniche di attacco e difesa dei sistemi.

Nel capitolo 3, dopo aver spiegato quale è il ruolo delle applicazioni in analisi all'interno dello scenario precedentemente introdotto, saranno spiegati più nel dettaglio gli obiettivi di questo progetto e le funzionalità proposte.

Il capitolo 4 invece si occuperà di introdurre le librerie e gli strumenti utilizzati nella realizzazione delle applicazioni. Qui è possibile trovare una breve descrizione dell'IDE usato, le librerie per la gestione del file .ini e la gestione del server FTP. Quindi, seguirà una breve introduzione all'utilizzo e installazione di JnetPcap, una libreria per gestire device, pacchetti, e protocolli di rete, con software scritto in Java (oltre a molte altre funzionalità che esulano dal campo di applicazione di queso progetto).

Nel capitolo 5 verranno affrontate nel dettaglio le caratteristiche strutturali e l'implementazione delle classi rappresentanti gli oggetti del sistema di cattura, filtraggio, e salvataggio del traffico presenti in CSD.

Nel capitolo 6, verrà analizzata nel dettaglio DMAD, l'applicazione client di analisi dei dati raccolti da CSD.

Nel capitolo 7 verranno spiegati nel dettaglio l'installazione e l'utilizzo del software, con alcuni consigli e raccomandazioni per ottimizzare la user experience e l'impostazione dei parametri di CSD e DMAD, a seconda dell'uso previsto.

Infine, il capitolo 8 si occuperà di raccogliere le conclusioni sui risultati ottenuti durante la realizzazione di questo progetto, aggiungendo note e spunti su alcune modifiche/aggiunte, nell'eventualità di futuri aggiornamenti del software.

2) Rete: funzionamento e analisi del traffico

2.1) La suite TCP/IP

In questa breve introduzione sulla suite di protocolli TCP/IP, verranno presentate le principali modalità di trasmissione dei dati nelle reti a pacchetto, oltre a una breve descrizione dei protocolli di interesse in questo progetto.

Il TCP/IP è una famiglia di protocolli (ognuno dei quali si occupa di gestire le informazioni sulla trasmissione del pacchetto a un preciso livello di astrazione, determinato dal tipo di protocollo), e prevede 4 livelli (a differenza dei 7 previsti da ISO/OSI), ognuno dei quali risolve uno specifico insieme di problematiche nella trasmissione/ricezione dei dati su rete a pacchetto (come ad esempio assicurarsi che un pacchetto sia ricevuto, inviarlo al destinatario corretto, far si che solo l'applicazione interessata riceva il pacchetto, controllare l'integrità del pacchetto, ecc.).

Il protocollo di un certo livello, inoltre, ha la funzione di fornire servizi di livello superiore, innalzando di volta in volta il livello di astrazione. Per fare ciò, ogni protocollo ha un preciso set di campi a disposizione (definito nel suo RFC - Request for Comments), che vengono inseriti nell'header previsto.

Quando un pacchetto viene inviato, i dati passano sequenzialmente attraverso i protocolli coinvolti, prima di essere inviati. Ad ogni livello, vengono aggiunti nell'header i parametri relativi alle funzionalità gestite dal protocollo: il payload del livello precedente (N), con l'aggiunta dell'header, viene incapsulato come nuova payload nel pacchetto di livello N-1. L'operazione viene eseguita per ogni livello fino al livello di accesso alla rete (o livello 1), che si occupa di trasmettere sul mezzo il payload originale (la sequenza di dati da inviare) inserito negli header aggiunti in precedenza.

Quando il pacchetto viene ricevuto, l'operazione prima descritta viene ripetuta in senso opposto, partendo dal livello più basso, e rimuovendo ad ogni livello successivo i relativi header di controllo. Dopo l'elaborazione degli header, avvenuta fino al livello interessato in quella comunicazione, i dati ricevuti sono disponibili per essere elaborati e gestiti a seconda del caso (ad esempio, da un applicazione che li riceve dal livello Transport).

TCP/IP prevede una pila di 4 livelli:

4 - APPLICAZIONE: ingloba applicazioni standard della rete

3 - TRASPORTO: assicura e controlla l'invio dei dati

2 - RETE: si assicura dell'invio del pacchetto al destinatario

1 - ACCESSO DI RETE: gestisce lo specifico tipo di connessione e codifica

Ecco alcuni protocolli TCP/IP utilizzati nel progetto CSD e il formato dei loro header:

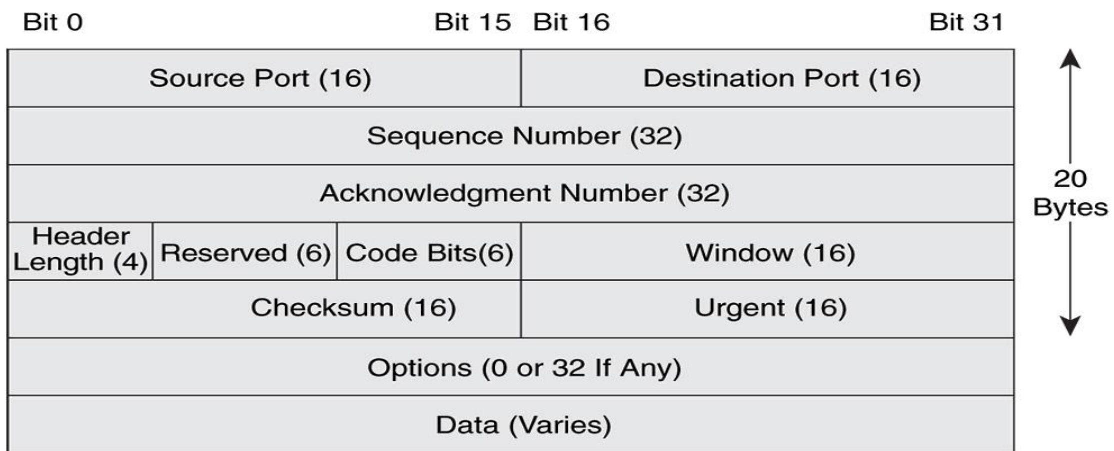

Figura 2.1 - Transmission Control Protocol (Livello Trasporto)

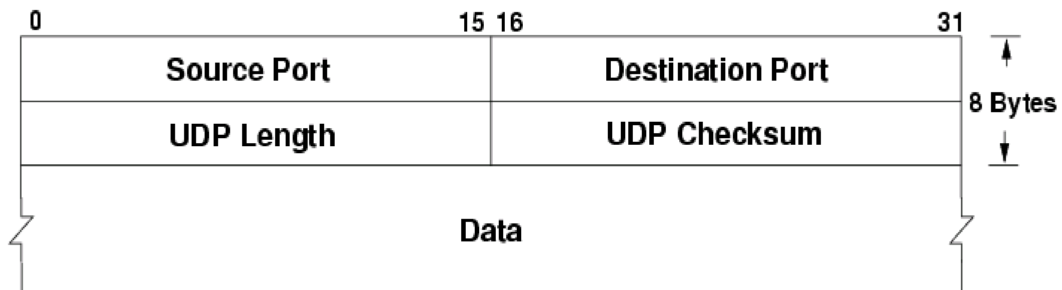

Figura 2.2 - User Datagram Protocol (Livello Trasporto)

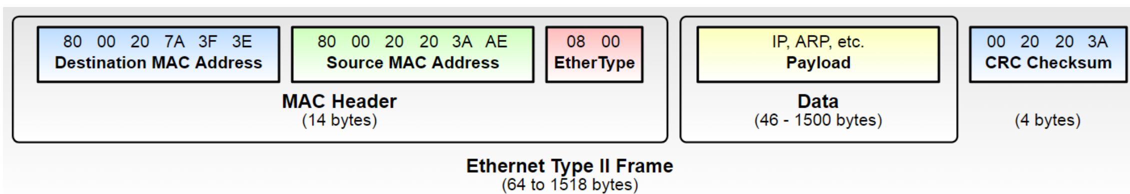

Figura 2.3 - Ethernet Protocol (Livello Accesso di Rete)

2.2) Cenni sulla sicurezza

Un calcolatore connesso in rete, come visto nel precedente capitolo, ha la possibilità di comunicare dati e messaggi con gli altri pc in rete, utilizzando i protocolli previsti da TCP/IP (o in alternativa ISO/OSI) per il trasferimento di dati a pacchetto. Ciò, oltre a permettere una moltitudine di servizi basati su informazioni condivise (o comunque locate al di fuori dello spazio di memoria contenuta nel calcolatore), introduce una serie di problematiche di sicurezza e blocco degli accessi indesiderati, siccome tali dati possono essere intercettati, modificati e inviati, andando a creare una potenziale minaccia per gli utenti. Inoltre, i protocolli esistenti soffrono di vulnerabilità peculiari dipendenti dal loro funzionamento, che possono essere sfruttate in diversi modi per ottenere un comportamento teoricamente non permesso dal protocollo/sistema, con lo scopo di aggirare i controlli di sicurezza che si basano su tali protocolli.

Per questi motivi è molto importante monitorare ed analizzare il traffico di rete di una macchina connessa in rete, specialmente se su di essa risiedono dati sensibili, oppure se essa è in grado di svolgere operazioni potenzialmente irreversibili o dannose, e in molti altri casi. Ci sono moltissimi modi in cui la privacy dei dati e delle connessioni di un calcolatore può essere compromessa, tuttavia, la maggior parte di essi consiste nell'intercettare dati, ottenere privilegi che non si dovrebbero possedere in un sistema software, o rendere irraggiungibile un servizio in rete. In un mondo fortemente influenzato dalla tecnologia e dalla diffusione di calcolatori attraverso un numero sempre maggiore di dispositivi, tali problematiche possono avere conseguenze disastrose: si pensi per esempio alla frequenza sempre maggiore con cui si utilizzano servizi relativi a settori fondamentali per la nostra società, quali sanità, fiscalità, commercio, e servizi bancari: eventuali vulnerabilità e/o mancati accorgimenti nella sicurezza del sistema informatico, potrebbero avere conseguenze molto pesanti nella vita reale e nei processi che tali software gestiscono.

Ad oggi, il software è diventato a tutti gli effetti un componente essenziale di quasi la totalità dei servizi principali utilizzati dalla popolazione, portando sicuramente una miriade di vantaggi, ma introducendo anche un elemento ulteriore di vulnerabilità del sistema, che può ora essere attaccato anche dalla sua interfaccia software, e non solo dalla sua interfaccia hardware. Per questo motivo è molto importante il monitoraggio delle porte attive e dei pacchetti ricevuti ed inviati dalla VulnBox, al fine di avere un

file di cattura contenente la registrazione del traffico avvenuto durante lo sniffing (con questo nome si indica l'operazione di cattura dei pacchetti di rete in transito), che può essere utilizzata per diversi tipi di analisi.

Per analizzare il traffico registrato, vengono processati i campi degli header di protocollo in cui il payload è stato incapsulato. Tali campi contengono tutte le informazioni di controllo e trasferimento utilizzate dai protocolli, e, se opportunamente analizzate, possono fornire elementi preziosi su come la rete è stata utilizzata durante il periodo di analisi, ed eventualmente riconoscere la tipologia di traffico (e lo scopo legittimo/malevolo).

A tal proposito verranno ora forniti alcune tecniche di difesa e attacco tra le più diffuse:

- TCP syn flooding : Questo è un attacco di tipo Denial Of Service, e consiste nell'inviare a una macchina target una serie (molto lunga) di pacchetti TCP con il campo SYN settato a true (detto anche three way handshake, che rappresenta il primo passo di instaurazione di una nuova connessione tcp) ,con l'obiettivo di saturare il numero di connessioni instaurabili, e rendendo il sistema irraggiungibile da remoto. Attacchi simili a questo sono ARP flooding e Smurfing.

- Man in the Middle (Famiglia di attacchi) : questa categoria di attacchi può essere eseguita in diversi modalità e sfruttando diversi protocolli, tutti con la caratteristica comune di intercettare/alterare/gestire/ritrasmettere il traffico relativo a una comunicazione tra due diversi host . Ad esempio, con un ARP poisoning è possibile comunicare il proprio MAC associandolo a un IP diverso dal proprio, e permettendo così di fingersi un host diverso da quello reale, ad esempio per ricevere risposte destinate a pc altrui, o per intercettare dati di accesso comunicati in rete nella fase iniziale, sfruttando le debolezze del protocollo scelto (*ARP poisoning* colpisce il protocollo di Address Resolution, responsabile dell'associazione tra IP e MAC in una rete locale). Altri esempi di questa tipologia di attacco sono il *DNS spoofing* e l'*Eavesdropping*

- SQL injection (esempio noto di attacco basato sull'input parsing): Questa vulnerabilità affligge le vecchie versioni di SQL (o nel caso in cui il programmatore abbia introdotto delle vulnerabilità, ad esempio non utilizzando i

prepared statements), e consiste nell'esecuzione errata di input inserito dall'utente nelle queries per l'ottenimento di dati (anche sensibili) da database SQL. Se questi input non vengono correttamente filtrati, potrebbero essere eseguiti come condizione booleana sull'input (ad esempio KEY=password || 1=1 elude il controllo sulla corrispondenza tra la password inserita e quella associata all'utente, presente nel database) e permettere di bypassare diversi tipi di controllo di possesso dei diritti di acquisizione dei dati. Nell'esempio fornito nelle parentesi, la condizione || 1=1 risulta sempre vera, e il fatto che sia dopo un or logico, fa si che qualunque valore venga assunto dalla condizione precedente KEY=Password la query venga comunque eseguita correttamente.

- Malware e Spyware: con questi due termini si intende un programma malevolo in senso generico, o un software atto al furto di dati non autorizzati su un sistema non posseduto o su cui non si ha l'autorizzazione all'accesso. Tale software (ad esempio nel caso di un keylogger o di un trojan), raccoglie informazioni durante l'utilizzo della macchina vittima e le comunica più o meno sporadicamente con una (o più di una) macchina attaccante locata all'esterno (o all'interno) della rete a cui l'host è connesso.

- Sniffing (attivo o passivo): questa metodologia consiste nell'intercettazione (e salvataggio in file) dei pacchetti in transito sul device di rete scelto per la cattura. Può essere utilizzata sia in attacco (scheda in modalità monitor/promiscua posta in una rete locale altrui), che in difesa (scheda utilizzata per scopi legittimi in rete locale propria), ed è un elemento fondamentale per raccogliere le informazioni necessarie sulla rete. Questa modalità, nella sua variante passiva, verrà approfondita nei successivi capitoli, essendo l'argomento principale del progetto CSD (relativamente all'utilizzo di questa tecnica a fini difensivi, controllando il traffico in transito nel pc da monitorare).

- Port scanning: si tratta di una modalità di analisi in cui vengono mandate richieste, utilizzando i protocolli messi a disposizione da TCP/IP (come ad esempio ICMP, incapsulato in IP a livello di rete, che permette le richieste di tipo ping per testare l'apertura/chiusura di una specifica porta). In questa tecnica

vengono sfruttate le modalità di check previste da TCP (syn e fin), ICMP (ping) e altri protocolli, con la caratteristica comune di sfruttare la risposta prevista a tali richieste: se il server risponde a una richiesta su una specifica porta, può significare che la porta è chiusa o che la porta è disponibile (a seconda della risposta ricevuta); Nel caso in cui non perviene risposta dal server significa che tale porta è bloccata da un firewall. Con questa modalità è possibile scoprire quali porte sono aperte e quali chiuse/filtrate da firewall, in modo da avere un controllo sulla possibilità di comunicare con la macchina target, operazione molto utile ed utilizzata sia nella difesa che nell'attacco dei sistemi software.

2.3) L'importanza dell'analisi del traffico

Come accennato nel capitolo precedente, l'analisi del traffico di rete si rivela uno strumento particolarmente efficace,si per il monitoraggio di un sistema a scopo difensivo,sia per scopo di attacco.
Siccome nell'ambito della presente tesi l'interesse è rivolto verso la fase di difesa, e alla relativa analisi delle informazioni presenti sul sistema sotto controllo, utilizzando diverse tecniche, le regole di comunicazione imposte dai protocolli TCP/IP, e la struttura degli header previsti da tali protocolli.
Tralasciando lo sniffing passivo utilizzato nelle tipologie più comuni di attacco, che può essere di difficile rilevamento, in quasi tutte le principali tecniche introdotte nel precedente capitolo, le informazioni catturate (o i pacchetti richiedenti risposte in maniera da aggirare la sicurezza dei protocolli) per raggiungere il server o la macchina attaccante, necessitano di transitare attraverso determinate porte del server (VulnBox), sia per le comunicazioni in uscita che per quelle in entrata.
Installando un programma di cattura del traffico sulla VulnBox, e analizzando il contenuto della payload dei pacchetti catturati (oltre ai campi degli header di tali pacchetti), è possibile riconoscere attività indesiderate svolte sulla VulnBox stessa, verificando l'eventuale corrispondenza della payload con diverse espressioni regolari tipiche dei pacchetti relativi a un attacco.
Al fine di difesa della privacy nelle comunicazioni tra la VulnBox e una macchina esterna, risulta molto efficace l'utilizzo della cifratura dei dati comunicati in rete, utilizzando i protocolli di comunicazione sicura quali FTPS, https, VPN, e alcuni

accorgimenti per ottimizzare l'efficacia di tali protocolli. In questo modo, la VulnBox risulta essere molto meno vulnerabile alla cattura del traffico, essendo esso criptato prima di essere inviato.

Allo scopo di monitorare i pacchetti in transito su un sistema (e usando le modalità monitor/promiscua, persino dell'intera rete, in molti casi), sono disponibili diversi strumenti che permettono di registrare il traffico, ed effettuare un'analisi, anche approfondita, del traffico di rete registrato. Tra i principali software di sniffing di rete, Wireshark e Firesheep sono solo alcuni dei più utilizzati.

Il primo dei due, Wireshark, permette infatti di catturare pacchetti in diverse modalità (non promiscua, promiscua, e in alcuni casi anche monitor), e rappresentarli in un'interfaccia grafica molto intuitiva e accattivante, permettendo un rapido accesso alle informazioni chiave, oltre a un efficace sistema di filtraggio dei pacchetti di interesse.

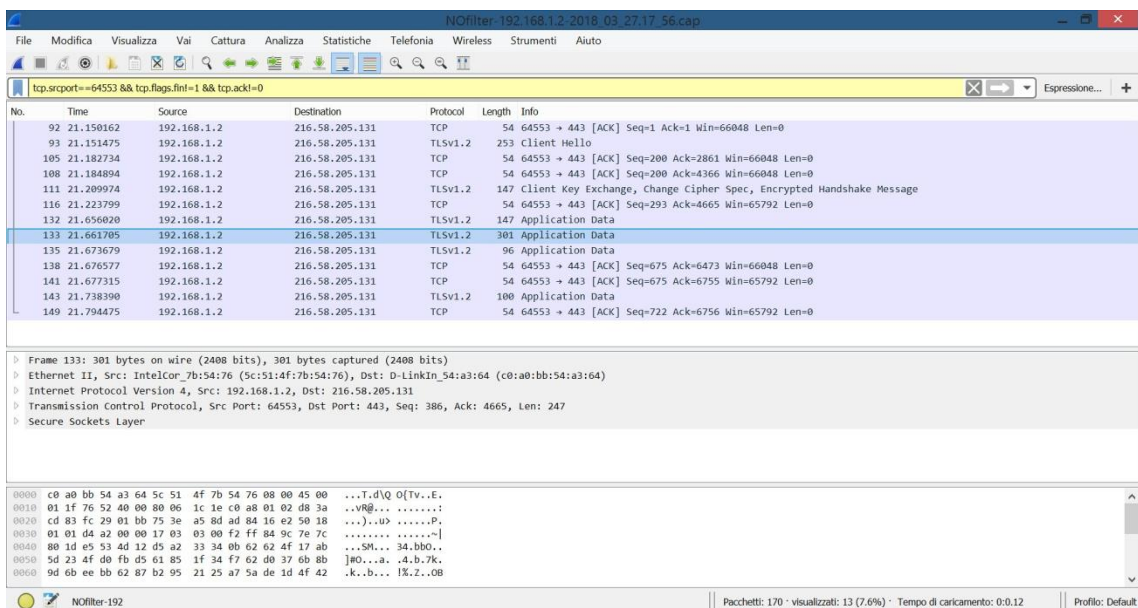

Figura 2.3.1 - Wireshark - esempio di apertura file .cap contenente dati del protocollo TLS

3) Il progetto: CSD – DMAD

3.1) Obiettivi del progetto

Ora che è stato introdotto lo scenario in cui si colloca il progetto in esame, è giunto il momento di approfondire le soluzioni proposte dai software CSD e DMAD.
Come accennato nel capitolo precedente, esistono diversi software in grado di effettuare sia la cattura che l'analisi dei pacchetti in transito su una (o tutte) le interfacce di rete, tuttavia, al fine di ottimizzare la personalizzazione della cattura e l'analisi, sono state individuate alcune utili funzionalità da affiancare a tali software, tra cui:

1. Catturare il traffico di rete, e filtrare i pacchetti in transito generando un file ".cap" grezzo non filtrato, e un file ".cap" per ogni porta specificata. Questa operazione viene eseguita ciclicamente ogni N minuti, fino al termine.
2. Al termine di ogni ciclo i dump vengono copiati in locale e/o su server FTP (scelta personalizzabile dall'utente). Quindi, viene ripresa la cattura.
3. Al fine di minimizzare la perdita di pacchetti tra un ciclo e l'altro, è prevista una modalità di salvataggio facente uso di thread.
4. I dati del server FTP, oltre a diverse impostazioni sulla cattura, sono specificati dall'utente in un file di configurazione con estensione ".ini", e successivamente letti dal software all'avvio.
5. E' prevista la possibilità di specificare diverse opzioni sul filtraggio dei pacchetti sulle porte specificate, come ad esempio l'esclusione dei flussi TCP generati dalla VulnBox, e il controllo delle espressioni regolari contenute nella TCP payload.
6. I file salvati sul server FTP vengono quindi letti e scaricati su una seconda macchina (funzione eseguibile anche direttamente su VulnBox a cattura ultimata), su cui viene svolta l'analisi attraverso una interfaccia grafica (GUI).
7. In fase di analisi , è possibile specificare un intervallo di tempo e una porta, oltre ai diversi parametri del filtro, e unire in un unico file di cattura i pacchetti che rispettano i parametri specificati catturati nel periodo di analisi. In alternativa all'uso del server FTP, è possibile caricare manualmente i dump generati durante la cattura ed effettuarne l'analisi.

8. E' stata pensata inoltre una funzione per esportare e unire in un unico dump tutti i pacchetti (non filtrati) catturati sulla VulnBox nel periodo specificato. In questo modo è possibile avere un unico export di grandi dimensioni contenente tutti i pacchetti catturati, utile per analisi approfondite con i software di analisi esistenti.

Le funzionalità descritte nei punti 1-5 sono presenti nel software CSD, avente interfaccia a riga di comando e file di configurazione, da avviare sulla VulnBox.

I punti 6-8 sono invece affrontati da DMAD, avente interfaccia grafica e un file di configurazione per il salvataggio/caricamento delle impostazioni.

I due software oggetto di questa testi, si offrono come strumento di integrazione alle analisi eseguibili con i numerosi software esistenti.
Tutti i file prodotti utilizzano il formato ".cap", un formato riconosciuto da molti software comunemente utilizzati (tra cui Wireshark), e questo rende CSD e DMAD due utili strumenti per ottimizzare le analisi in tali software, raccogliendo automaticamente i dump su server(o in locale), e fornendo diversi possibili livelli di filtraggio dei pacchetti catturati a seconda delle necessità, rendendo molto più efficace il controllo dei dati di interesse con Wireshark e/o altri programmi di analisi esistenti compatibili con il formato ".cap". Inoltre, l'unione dei dump di interesse in un unico dump in maniera automatica velocizza notevolmente le operazioni di analisi, rendendo disponibili in un unico dump tutti i pacchetti catturati, senza il bisogno di analizzare tutto il periodo catturato, o i pacchetti relativi a porte non interessanti a fini di analisi.
Per questi motivi, nella progettazione dei due software si è cercato di tenere come obiettivo principale la necessità di ideare e creare classi e metodi che potessero dare all'utente finale una varietà quanto più vasta possibile dei possibili casi di utilizzo delle applicazioni CSD e DMAD, con la possibilità di impostare diversi livelli il filtraggio dei pacchetti e con diverse modalità a seconda delle necessità dell'utente.
Tutto ciò vale sia per la cattura e invio dei dati (CSD), che per la loro analisi (DMAD).

4) Strumenti utilizzati e librerie

Per la realizzazione del codice relativo ai due progetti CSD e DMAD, è stato scelta la tecnologia Java, con l'utilizzo dell'IDE Eclipse, Windowbuilder (un plugin per la realizzazione di interfacce grafiche), e alcune librerie di terze parti per le diverse funzioni necessarie (tra cui ini4j, una libreria per l'utilizzo di file di configurazione in formato INI, e JnetPcap, un wrapper JNI per l'utilizzo in Java delle librerie libPcap, librerie esclusive del linguaggio C).

4.1) JDK/JRE 8.0-162 - Eclipse Oxygen

Come anticipato, Java è il linguaggio scelto per la realizzazione delle due applicazioni in esame. Tale scelta è dovuta alla interoperabilità offerta da tale tecnologia nella realizzazione di applicazioni eseguibili sui sistemi operativi per cui è disponibile una Java Virtual Machine (JVM) in grado di leggere i *Bytecode* (estensione .class) contenenti il codice eseguibile dalla JVM.

Inoltre, in questo progetto è stato scelto di utilizzare prevalentemente un sistema Windows sia per la realizzazione che per i test, al fine di focalizzare l'utilizzo su macchine per cui la disponibilità di software "ad hoc" per l'analisi risulta più limitata.

Tuttavia, come premesso, le applicazioni CSD e DMAD sono progettate per essere eseguite anche su diversi sistemi con Kernel Linux (consultare la pagina web *http://jnetpcap.com/node/270* per informazioni dettagliate sul supporto della cattura dei pacchetti utilizzando la JnetPcap), tenendo conto che il loro funzionamento è testato principalmente per sistemi Windows, e potrebbero perciò emergere eventuali comportamenti inattesi o malfunzionamenti sui sistemi diversi da quello di Microsoft.

Per la compilazione del codice realizzato è stato utilizzato il Java Development Kit 8 (JDK).

Per la stesura del codice è stato scelto Oxygen, la versione 4.7 di Eclipse, un IDE (Integrated Development Environment) comprensivo di una vasta varietà di funzioni e tool per lo sviluppo, in diversi linguaggi, di applicazioni multipiattaforma, applicazioni web, e molto altro.

4.2) Jnetpcap – Winpcap/libpcap

JnetPcap è una libreria Java (sviluppata da Sly Technologies Inc.) che permette di catturare e gestire i pacchetti di rete, sia in transito su una scheda di rete che in lettura da file di cattura avente formato valido (come ad esempio ".cap", ".pcap").

Come anticipato, Jnetpcap utilizza la tecnologia JNI per richiamare, attraverso un *wrapper* "JnetPcap.dll", le librerie libpcap/WinPcap (spiegazione a seguire), definite in linguaggio C. Esse si occupano di gestire nativamente l'accesso ai device di cattura e permettono la gestione dei pacchetti, oltre a permettere l'accesso ai file di cattura.

Il funzionamento di questa libreria si basa su librerie native diverse a seconda che si tratti di un sistema Windows o Linux:

- Libpcap è una libreria per l'accesso in ambiente Linux alle risorse necessarie per catturare, filtrare, trasmettere, e analizzare i pacchetti relativi al traffico di rete.
- WinPcap è una libreria open source sviluppata per rendere disponibile su sistemi Windows l'accesso ai dati grezzi necessari per la realizzazioni delle funzioni di cattura, filtro, trasmissione e analisi dei pacchetti, siccome nel sistema di casa Microsoft non sono previste le primitive necessarie per l'implementazione di tali funzioni.

Nello specifico delle applicazioni CSD e DMAD, JnetPcap è utilizzata sia per catturare periodicamente l'intero traffico di rete separandolo in un diverso flusso per ogni porta da monitorare, che per le funzioni di controllo e filtraggio previste nell'analisi.

A tal proposito, per gestire l'utilizzo e installazione di CSD e DMAD in maniera semiautomatica (solo nel caso di sistemi Windows), sono stati realizzati due file batch contenenti le istruzioni per l'aggiunta delle librerie al *classpath* di esecuzione, e per la copia del wrapper "JnetPcap.dll" all'interno della directory di sistema "C:\\Windows\System32" .

4.3) Ini4j

Ini4j permette di leggere e scrivere dati di tipo *String* in file di configurazione in formato INI.

Questo tipo di file prevede una strutturazione dei dati in sezioni (definite tra parentesi quadre) e variabili String associate a quella sezione, inizializzate a un valore con una semplice uguaglianza. Sarà poi il programma a preoccuparsi di gestire e/o convertire tali

dati nei formati adatti all'utilizzo (in DMAD ad esempio sono gestiti tramite conversione da String dati rappresentanti interi e date, oltre alle stringhe di caratteri).
Il principale vantaggio del formato INI è senza dubbio la sua facile interpretabilità, essendo costituito da caratteri dell'alfabeto facilmente comprensibili da un essere umano. Per questo motivo è stata adottata questa soluzione per la gestione dell'input in CSD, riducendo al minimo la ripetitività/complessità tipiche dei software eseguiti da terminale.
Ini4j permette diverse operazioni sui file, tra cui *multi-values* e *tree-model,* tuttavia nei progetti in esame sono state utilizzate esclusivamente i metodi *Get, Put,* e *Store*. Questi tre metodi permettono rispettivamente di leggere, preparare i dati alla scrittura, e scrivere su file .ini, e possono essere invocati su oggetti della classe *Wini,* speficicando come parametri la sezione e il nome del valore (oltre ovviamente al dato da salvare, nel caso di *Put*).

4.4) Apache.commons.net-3.6

Le librerie commons.net sono delle librerie per sdk java (nelle sue versioni dalla 1.6 in poi), create e distribuite da Apache Software Foundation.

In questa API sono definite classi e metodi per la gestione dei principiali protocolli applicativi presenti sul web, e gestire diversi servizi associati ad essi.

Tra i più importanti, vanno citati FTP/FTPS, HTTP, IMAP, POP3, WHOIS, e TELNET. Con quest'ultimo, ad esempio, è possibile gestire sessione di input per gestire macchine remote connesse in rete.

Nell'ambito di questa tesi, viene utilizzata la classe FTPSClient, contenente costruttori e metodi utili per gestire le connessioni FTPS, utilizzando uno specifico protocollo di sicurezza tra quelli disponibili (nel nostro caso, verrà utilizzato *Secure Sockets Layer*).
Inoltre, tale classe è utilizzata a scopi di controllo delle directory remote durante le fasi di download “pre-analisi” (software DMAD, classe Dumper).
E' stato scelto di utilizzare il protocollo TLS (Transport Layer Security), dopo una precedente fase di test con modalità FTP non sicura, e una successiva versione intermedia che utilizzava il protocollo SSL (Secure Sockets Layer). La scelta finale è ricaduta su TLS, essendo il protocollo più recente ed aggiornato, previsto da FTPS.

Tuttavia, non sono da escludersi scoperte future di nuove vulnerabilità, con la conseguente necessità di modifica nel tipo di protocollo utilizzato.

4.5) Java.util.regex.Pattern

Questa libreria fa parte delle librerie standard del JDK, e racchiude una collezione di classi e metodi molto utili per la ricerca di espressioni regolari in una variabile di tipo String.

Nello specifico dei progetti CSD e DMAD, è stato utilizzato il metodo statico Pattern.matches(String control, String payload) per cercare espressioni regolari nelle payload TCP (dopo averle convertite da array di Byte in String). Al fine di raggiungere questo obiettivo, il metodo matches() prende come parametro la stringa contenente l'espressione regolare, e la stringa da analizzare: nel caso viene trovata una corrispondenza viene restituito il valore booleano TRUE. L'espressione regolare deve essere definita secondo una precisa sintassi, e in essa vengono indicati i parametri che stabiliscono la struttura da riconoscere all'interno della stringa in analisi. Ecco alcuni esempi in grado di mostrare le potenzialità della sintassi regex (Regular Expression):

1. [a-zA-Z_0-9]{6}\d\d

>> Stringa formata da sei parole alfanumeriche, seguite da due numeri.

2. [a-zA-Z0-9._%-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}

>> Stringa formata da un indirizzo e-mail.

3. .*[BLOCK].*

>> E' contenuta una stringa di tipo BLOCK (come ad esempio i casi 1 e 2).

4. .*http\://[a-zA-Z0-9\-\.]+\.[a-zA-Z]{2,3}(/\S*)?.*

>> E' contenuta una stringa rappresentante un URL http.

5. .*scelta-[^abc].*

>> E' contenuta la stringa "scelta-" seguita da un carattere diverso da "a", "b", o "c".

Si consiglia di approfondire i casi previsti dalla classe Pattern, al fine di massimizzare i risultati potenzialmente ottenibili dal filtro pacchetti contenuto in CSD e DMAD. Per informazioni, consultare la documentazione di Pattern sul sito di Oracle:

- https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

4.6) Eclipse WindowBuilder

Si tratta di un plugin per Eclipse molto intuitivo, che permette di creare UI (User Interface) utilizzando le famose librerie java SWT e SWING. Si installa direttamente da Eclipse, e permette in poco tempo di realizzare interfacce utente trascinando i componenti sulla finestra principale del software, personalizzandone le varie opzioni.

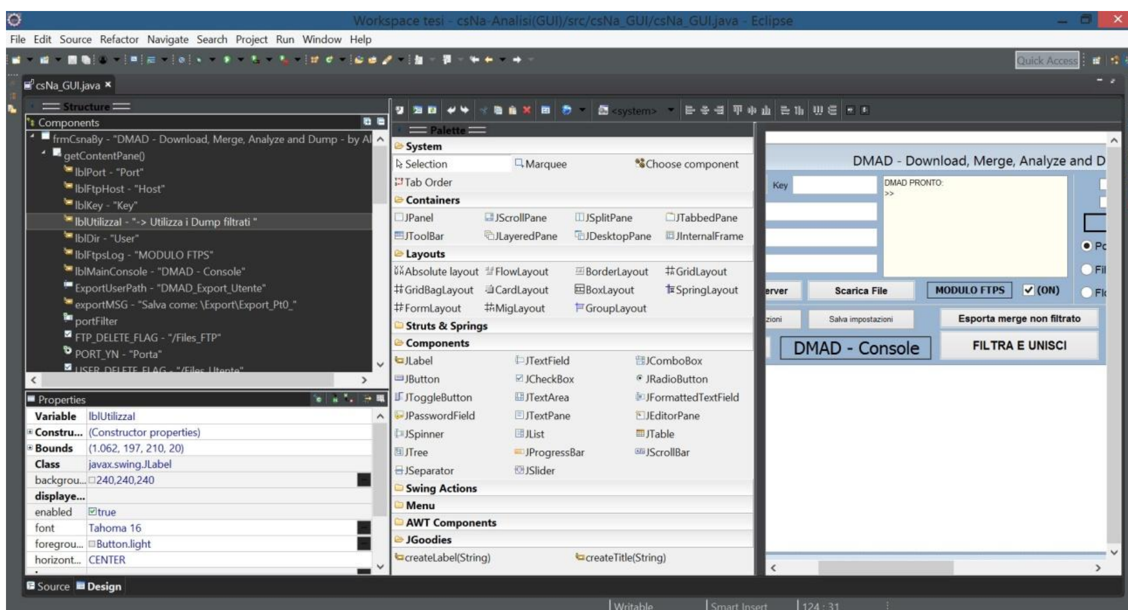

Figura 4.6.1 - l'IDE Eclipse Oxygen. Particolare del tool WindowBuilder

5) CSD – Capture Split N' Dump

In questo capitolo verranno presentate nel dettaglio le classi facenti parti del software da eseguire sulla VulnBox per la cattura del traffico, contenenti le funzioni che implementano la cattura del traffico e la sua suddivisione per porta, oltre al codice relativo alla lettura del file CONFIG.ini e al caricamento su server FTP.

5.1) CSD: progetto

Per quanto riguarda l'attribuzione delle funzionalità alle relative classi, con la conseguente scelta di quali compiti delegare a oggetti e metodi esterni alla classe main(), è stato scelto di limitare volutamente il numero di oggetti definiti, siccome non sono state individuate particolari necessità in fatto di scalabilità del software. Per questo motivo, le funzioni di creazione di file locali temporanei (e di backup), oltre alla lettura dell'input da file .ini e altre funzionalità secondarie, sono state definite direttamente all'interno del main.

Alla luce di ciò, e viste le osservazioni fatte nei capitoli 2) e 3), la scelta è ricaduta su un numero di 5 classi in totale:

- Main : gestisce input essenziale da terminale, legge file .ini e scrive files locali.
- Dumper : gestisce il controllo dei dati del server, e il salvataggio remoto su server FTP.
- ThreadDumper : gestisce la modalità di salvataggio thread, facendo uso di Dumper.
- CaptureCycle : contiene i metodi per catturare il traffico e filtrarlo data una porta.
- CharAnimationPrinter : classe per la stampa animata del testo su terminale.

Per tenere traccia dei dump non filtrati, è stato scelto di partire catturando tutto il traffico, e salvarne una copia locale al termine del ciclo i-esimo di scansione, al fine di evitare perdita di dati durante il successivo ciclo (nella thread-version, la cattura riprende pochi secondi dopo il termine del ciclo, e il caricamento potrebbe non essere completo entro tale termine). Quindi, sarà tale copia locale ad essere caricata sul server: in tal modo si evita la concorrenza sull'accesso ai file durante il caricamento, o sovrascritture non volute dei pacchetti registrati. A questo punto i file creati e copiati nella cartella locale "dump/Local_capture", dopo essere stati caricati sul server vengono processati dal metodo split, che si occupa di generare un file di cattura contenente i pacchetti che rispettano i parametri del filtro scelto (tra cui porta e le varie opzioni secondarie).
A questo punto, il file "splittato" per la porta data, è pronto per essere copiato nella cartella "dump/Local_capture" e caricato su server.
Quindi, queste operazioni sullo "split" (filtro,backup locale, caricamento server) vengono ripetute per ognuna delle porte specificate dall'utente, sempre a partire dallo stesso file non filtrato catturato all'inizio del ciclo.
Una volta esaurite le porte in analisi, il ciclo termina e si passa al ciclo successivo (si ricorda a tal proposito che nella modalità thread la cattura inizia concorrentemente subito dopo l'avvenuta creazione dei backup locali), e le operazioni descritte si ripetono fino al termine della durata di scansione specificata.

A seguire viene presentata la struttura delle classi e la loro interconnessione:

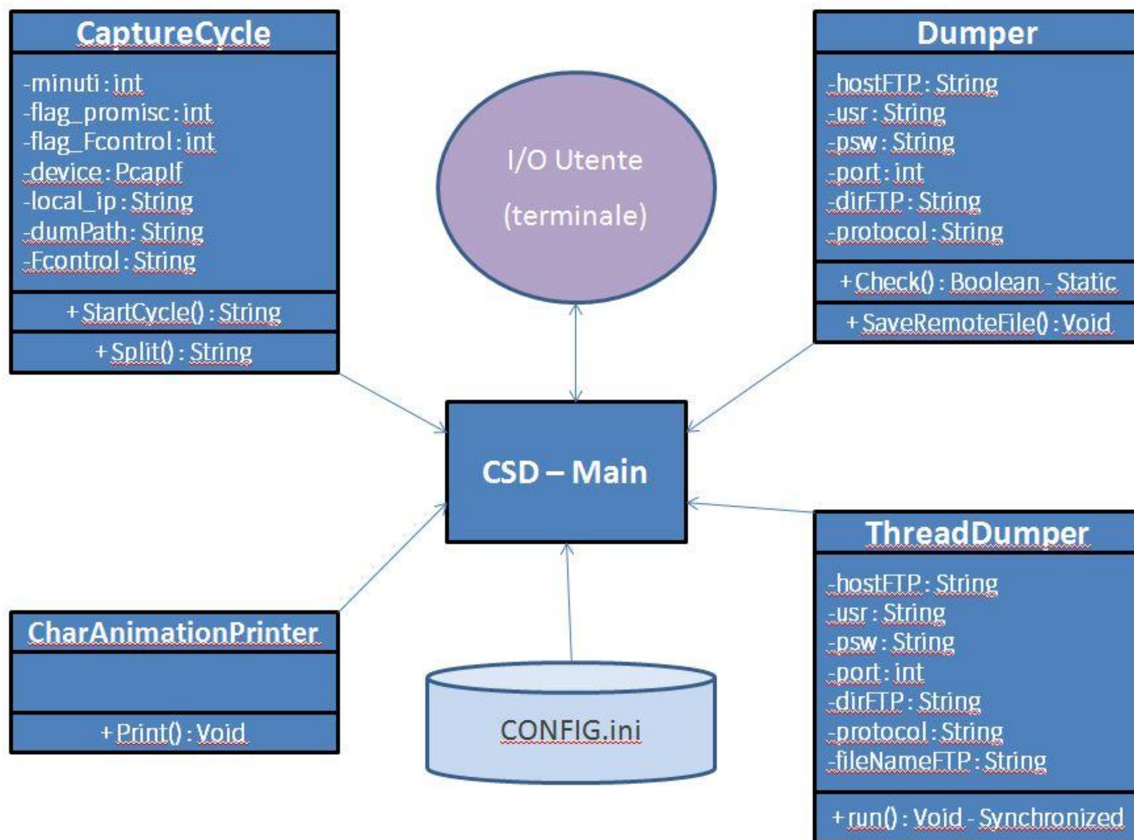

Figura 5.1 - Schema delle classi - CSD

A partire dal successivo capitolo, verranno mostrate nel dettaglio le classi e metodi introdotti, con riferimento al codice sorgente, partendo dalla classe Main(), che racchiude la logica generale del software (escludendo le funzioni specifiche svolte dagli oggetti appositamente creati, che verranno trattate nei loro rispettivi capitoli di appartenenza).

5.2) CSD: la Classe Main()

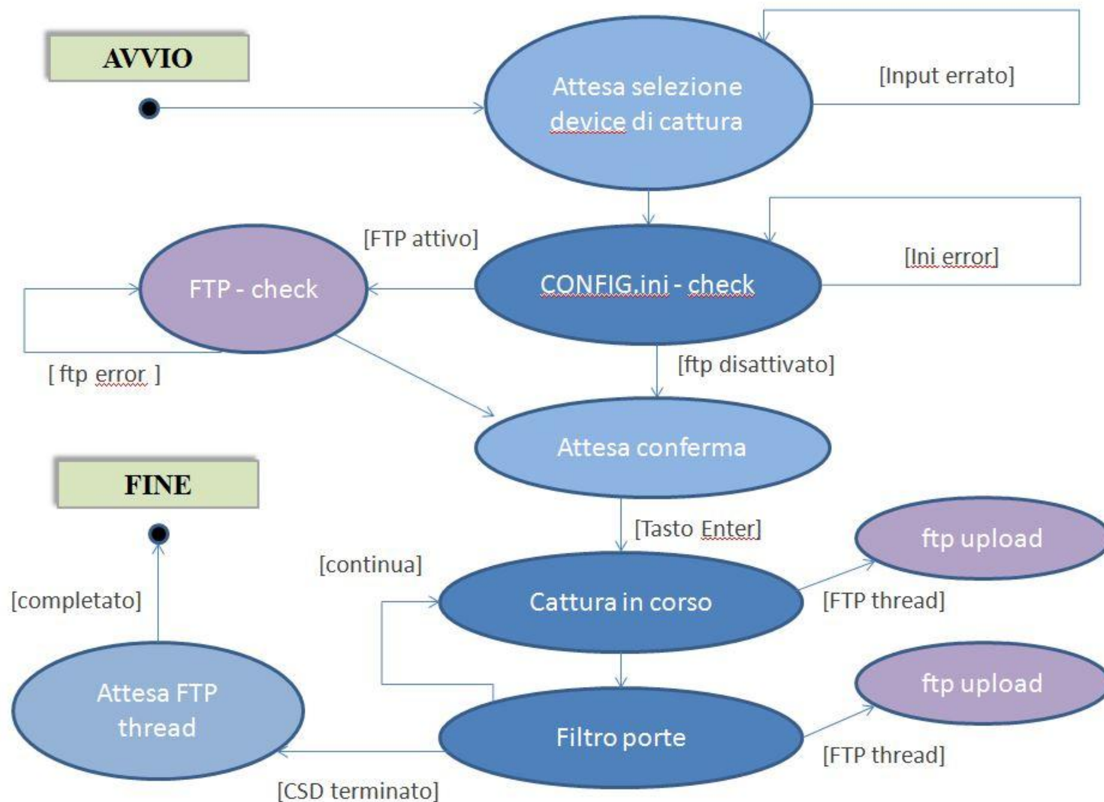

Figura 5.2 - CSD - Diagramma degli stati - Main

Nell'immagine proposta, viene mostrato il diagramma degli stati della classe Main, il nucleo di CSD, contenente: gestione dell'I/O, gestione file CONFIG.ini, le chiamate alle classi CaptureCycle, Dumper, ThreadDumper e CharAnimationPrinter, ed infine la terminazione del software.

A seguire una descrizione del codice relativo ai punti descritti nella figura.

```
37      List<PcapIf> alldevs = new ArrayList<PcapIf>();
38      StringBuilder errbuf = new StringBuilder();
39      //ottengo una lista dei device trovati sulla VM
40      int r = Pcap.findAllDevs(alldevs, errbuf);
41      //se viene riscontrato un qualche problema il programma si arresta
42      //e ritorna il messaggio di errore salvato nel buffer degli errori
43      if (r == Pcap.NOT_OK || alldevs.isEmpty()) {
44          System.err.printf("Errore \n %s", errbuf.toString());
45         return;
46         }
47      System.out.println("Dispositivi di rete trovati:\n");
48      int i = 0;
49      for (PcapIf device : alldevs) {
50          String description =  (device.getDescription() != null) ?
```

```
                  device.getDescription()
51                   : "[Nessuna descrizione disponibile]";
52             System.out.printf("#%d: %s [%s]\n", i++, device.getName(),
                  description);
53             }
54            int devAscolto=-1;
55            Scanner input = new Scanner(System.in);
56            while (devAscolto<0||devAscolto>(alldevs.size()-1)) {
57             try {
58                 System.out.println("\n-> Scegliere un dispositivo compreso
                      tra 0 e "+(alldevs.size()-1)+") \n");
59                 devAscolto = Integer.parseInt(input.nextLine());
60                 if (devAscolto<0||devAscolto>alldevs.size()-1) {
61                     System.out.println("\nDispositivo non in lista");
62                 }
63             }catch(Exception e) {System.out.println("\nDigitare un Numero");}
64            }
65
66        PcapIf device = alldevs.get(devAscolto); // seleziono il device
```

Codice 5.2.1 - CSD: la classe Main()

le prime righe di codice creano una lista di device per il salvataggio dei device trovati, e una buffer in cui salvare gli eventuali messaggi di errore restituiti da Jnetpcap.

Successivamente (righe 54-66) viene richiesto tramite console di specificare l'interfaccia di rete sulla quale catturare il traffico.

I possibili input errati sono gestiti tramite eccezione (riga 63: parseInt() lancia eccezione e viene riconosciuto un inserimento non numerico) o tramite struttura if(){} (se numero non compreso tra 0 e il numero massimo digitabile), e in entrambi i casi viene nuovamente richiesto di specificare il device di cattura. Questa funzionalità è implementata tramite l'utilizzo di un ciclo while.

```
                           //LETTURA CONFIG.ini
71        Ini inifile= new Ini();
72        boolean continua=false;
73        while(!continua) {
74            try {
75                Ini conFile = new Ini(new FileReader("CONFIG.ini"));
76                inifile=conFile;
77                System.out.println("(CONFIG.ini caricato correttamente)\n");
78                continua=true;
79            }catch(Exception e)
                  {System.out.println("errore nella lettura di CONFIG.ini"
80                +e.getMessage()+" premere invio per riprovare.");
81            }
82        }
83
84        String parentFTPdir=inifile.get("FTP_LOGIN", "PathFTP");
85        String host=inifile.get("FTP_LOGIN", "HostFTP");
86        String usr=inifile.get("FTP_LOGIN", "UsrFTP");
87        String hasKEY=inifile.get("FTP_LOGIN", "KeyFTP");
88        String iniPORTS=inifile.get("SPLIT_OPTION", "Port");
```

```
89      int splitDumpOnly=Integer.parseInt(inifile.get("FTP_OPTION",
            "DumpOnlySplit"));
90      int PortFTP=Integer.parseInt(inifile.get("FTP_LOGIN", "PortFTP"));
91      int DumpYN=Integer.parseInt(inifile.get("FTP_OPTION", "DumpFTP"));
92      String threadFLAG=inifile.get("FTP_OPTION", "Threaded_version_Beta");
93      int FLAG_th=0;//flag per attivare disattivare il salvataggio thread
94      if(threadFLAG.equals("1")) {
95          FLAG_th=1;
96          out_msg=" \n*** Modalit�alvataggio thread FTP attiva ***";
97          ConsoleAnimated.Print(out_msg, 20);
98      }else {
99          out_msg="*** Modalit�alvataggio FTP sequenziale attiva ***;
100         ConsoleAnimated.Print(out_msg, 20);
101     }
102
103     try{
104         Thread.sleep(300);
105     }catch(Exception e) {}
```

Codice 5.2.2 - CSD: la classe Main()

Nelle righe 71-82 viene effettuato il caricamento del file CONFIG.ini (non essendo usato nessun path assoluto viene assunto che il file si trovi nella cartella radice di CSD). Nelle righe seguenti vengono create ed inizializzate una ad una le variabili contenenti i parametri impostabili dall'utente tramite il file CONFIG.

ConsoleAnimated.Print() verrà approfondita al termine del capitolo 5), e si occupa di stampare i caratteri attendendo un tempo proporzionale a quello specificato dall'utente in millisecondi.

```
108     //ottengo le porte indicate nel CONFIG.ini
109     StringTokenizer tokenizer = new StringTokenizer(iniPORTS, ",");
110     int PortNumber=tokenizer.countTokens();
111     int [] ports=new int[PortNumber];
112     System.out.println("\n"+"-> Rilevate "+PortNumber+" porte da
            scansionare: \n");
113     for(int j=0;j<PortNumber;j++) {
114         ports[j]=Integer.parseInt(tokenizer.nextToken());
115         System.out.println("porta "+(j+1)+": "+ports[j]);
```

Codice 5.2.3 - CSD: la classe Main()

Nel frammento di codice 5.2.3 viene effettuata la lettura delle porte, utilizzate per la separazione del dump di cattura. Le porte, separate da una virgola, vengono lette dal file ini tramite la stringa iniPORTS, e salvate in un array di interi con dimensione uguale al numero di porte specificate (token).

```
131     //se il salvataggio FTP Settato da CONFIG
132     if(DumpYN==1){
133
134         System.out.println("\n-> Lettura path cartella di destinazione su
                "+host+" (specificata in CONFIG.ini):"+parentFTPdir+" <=");
```

```
136         boolean ask=false;
//serve per chiedere chiave FTP in caso di errore nella psw in CONFIG.ini

137         do
138         {
139            FTPKey=hasKEY;
141            if(ask&&!hasKEY.equals("")) {
142               System.out.println("Per proseguire,individuare il problema
143                     + "e successivamente premere invio. \n\n-> ");
144               out_msg="Caricamento ...";
145               input.nextLine();//se premi invio ricarico dati FTP
146               ConsoleAnimated.Print(out_msg, 100);
148               try {
149                     Ini conFile = new Ini(new FileReader("CONFIG.ini"));
150                     inifile=conFile;
151                     System.out.println("CONFIG caricato\n");
152                 }catch(Exception e) {System.out.println("Problemi: "
153                     +e.getMessage());
154                 }
155               FTPKey=inifile.get("FTP_LOGIN", "KeyFTP");//chiave
156               host=inifile.get("FTP_LOGIN", "HostFTP");//host
157               usr=inifile.get("FTP_LOGIN", "UsrFTP");//user
158            PortFTP=Integer.parseInt(inifile.get("FTP_LOGIN", "PortFTP"));
159            }//se ask settato o chiave nulla
161            if (hasKEY.equals("")){
162                System.out.println("\n-> Inserire la password FTP");
163                FTPKey=input.nextLine();
164            }
166            if(Dumper.CheckFTP(host, usr, FTPKey,PortFTP)==false) {
167                ask=true;
168                System.out.println("Password inserita: "+FTPKey+"\n\n->
                      Errore nella conessione al server FTP: "
169      +host+ "\n\n(Se la password è corretta e la connessione di rete
                 funzionante, Controlla CONFIG.ini)\n"
170      + "Assicurati di non avere un firewall attivo che impedisca la
            comunicazione con il server attraverso la porta "+PortFTP);
172            }
173            }while(Dumper.CheckFTP(host, usr, FTPKey,PortFTP)==false);
174            System.out.println("\n-> Dati corretti, connesso+host);
175      }//end IF(DumpFTP==1)
```

Codice 5.2.4 - CSD: la classe Main()

Nelle righe 131-175 vengono gestiti i casi in cui il caricamento FTP dei dump viene rilevato attivo, ovvero con valore dumpYN=1.

In questa porzione di codice vengono gestiti anche i casi di errore quali :

- Utente, host, porta o password non corrette: viene richiesto di riavviare il check una volta che i valori del file CONFIG sono stati corretti;
- Password non specificata in CONFIG: viene richiesta da terminale;
- Firewall attivo che blocca la comunicazione, modalità passiva non supportata, o mancanza di connessione alla rete.

Tali funzionalità sono realizzate per mezzo della Classe dumper e del metodo statico Check() in essa definito, come verrà descritto nel capitolo 5.3.

```
178     long iniDurata_cattura;
179     try {
180 iniDurata_cattura=Long.parseLong(inifile.get("INFO","CaptureUntil"));
181         System.out.println("-> [CaptureUntil="+iniDurata_cattura+"]"+
182           " >> CSD si chiuderà automaticamente dopo "+iniDurata_cattura
183                     +" minuti dall'inizio dello scan.\n");
184     }catch(Exception e) {
185         iniDurata_cattura=60;//se input non corretto default 60 minuti
186         System.out.println("CaptureUntil non specificato correttamente in
              CONFIG.ini. Speficicare un numero intero."
187         +"\n\n-> Utilizzo CaptureUntil= "+iniDurata_cattura+ " minuti di
            DEFAULT\n");
188     }
```

Codice 5.2.5 - CSD: la classe Main()

In questa sezione, vengono letti dal file CONFIG i minuti di cattura per cui CSD dovrà catturare il traffico, interrompendo e salvando ogni N minuti (valore presente nella variabile Cycle definita in CONFIG.ini). Nel caso di input non numerico, la struttura *try-catch* permette di assegnare un valore di default scelto dal programmatore (60 minuti).

```
189     System.out.println("PREMERE INVIO PER INIZIARE LA SCANSIONE");
190     input.nextLine(); //attesa input utente per iniziare lo scan
191     input.close();
192     //creo un'istanza di FTPDumper per il salvataggio file su FTP
193     // definendo host, username e password.
195     //il codice dell'accesso FTP E' attivato solo all'invocazione dei
196     //metodi check() e saveRemoteFile() )
197     Dumper FTPDumper=new Dumper(host, usr, FTPKey,PortFTP);
198     //inizializzo la blacklist che manterr� pacchetti di riferimento
199     //che rappresentano ogni flusso tcp bloccato (richiesta in uscita o
200     //risposta in entrata)
201     Vector <PcapPacket> packetBlacklist = new Vector<PcapPacket>();
202     //ora posso avviare un ciclo di cattura.
206     String tempDumpPath="Dump/temp";
207CaptureCycle ciclo = new CaptureCycle(device,errbuf,inifile,tempDumpPath);
212     int contaCicli=0;
213     //inizializza Elapsed, usato per terminare il programma
214     long ELAPSED=System.nanoTime();
215     long Limit=ELAPSED+(iniDurata_cattura*60*1000000000);
        //calcola valore limite di system.currenttime()
```

Codice 5.2.6 - CSD: la classe Main()

Una volta confermato l'avvio della scansione da parte dell'utente, prima dell'avvio del primo ciclo vengono inizializzate le variabili utilizzate dal filtro pacchetti, oltre al Dumper su cui verranno chiamati i metodi per salvare i file su server FTP in modalità

sicura. Viene inoltre inizializzato il valore della cartella /temp in cui verranno salvai i dati temporanei del ciclo di scansione dei pacchetti di rete.

Nelle righe 214-215, viene inizializzato il valore di tempo corrente: una volta convertito in minuti, sarà possibile capire quando terminare CSD confrontandolo con il valore in specificato nel file CONFIG.

```
216 while(ELAPSED<Limit) { //inizio cattura - fino a tempo esaurito
218     System.out.println("In ascolto sul device"+device.getName());
219     String sourcefile=tempDumpPath+"/preDump.cap";
220     String date = new SimpleDateFormat("yyyy_MM_dd").format(new Date());
221     Calendar cal = Calendar.getInstance();
222        SimpleDateFormat sdf = new SimpleDateFormat("HH_mm");
223        String time=sdf.format(cal.getTime());
224        String backupfile="Dump/Local_capture/NOfilter-"+ip+"-
              +date+"."+time+".cap";
225     try{ //inizia ciclo i-esimo di cattura e ottieni string del file out
231         sourcefile=ciclo.startCycle();
235         File File_in = new File(sourcefile);
236         File File_out= new File(backupfile);
237         FileInputStream fsin = new FileInputStream(File_in);
238         FileOutputStream fsout = new FileOutputStream(File_out);
239         FileChannel sourceChannel = fsin.getChannel();
240         FileChannel destChannel = fsout.getChannel();
241         destChannel.transferFrom(sourceChannel, 0, sourceChannel.size());
242         sourceChannel.close();
243         destChannel.close();
244         fsin.close();
245         fsout.close();
246         System.out.println("\n Il file "+backupfile+" ha"
     +File_out.length()+" Bytes di dati.Salvato in \"Dump/Local_capture\"");
247     }catch(Exception e) {System.out.println(e.getMessage());};
```

Codice 5.2.7 - CSD: la classe Main()

A questo punto inizia il while che durerà fino al termine dell'esecuzione di CSD. Vengono inizializzati i valori di tempo e data utilizzati per la creazione del nome unico del dump dato da *data_ore_minuti*, e il file temporaneo relativo al dump non filtrato.

Quindi avviene la scansione dei pacchetti con la chiamata (bloccante) del metodo StartCycle(), che si occupa di catturare pacchetti per N minuti e salvarli in un dump all'interno della directory "/temp". A questo punto viene effettuata una copia del dump locale non filtrato e salvata all'interno della cartella "Dump/Local_capture".

```
250     //FASE DI INVIO DUMP NON FILTRATO (Se attivo dumpFTP e NOfilter FTP)
262     String folderFTPath = parentFTPdir+"/"+date;
264     String FTPfilename=("NOfilter-"+ip+"-"+date+"."+time+".cap");
265     //nome nuovo file su FTP: FTPfilename
268     if (splitDumpOnly==0&&DumpYN==1) {
270         if(FLAG_th==1) {//se flag thread save attivo
272             synchronized (backupfile) {
274           ThreadDumper dumper=new ThreadDumper(host, usr,
```

```
            FTPKey,PortFTP, backupfile, FTPfilename, folderFTPath);
276         new Thread(dumper).start();
278         }
279       }else { // Se thread save flag TCP disattivato
281    FTPDumper.SaveRemoteFile(backupfile,FTPfilename,folderFTPath,inifile);
282    }}
```

Codice 5.2.8 - CSD: la classe Main()

Le righe presenti nel blocco di codice 5.2.8 effettuano il salvataggio su server FTP associato alla variabile dumper precedentemente inizializzata, dopo aver controllato che il salvataggio su server non sia stato disabilitato secondo le impostazioni inserite in CONFIG.ini (nello specifico SplitDumpOnly e DumpYN indicano rispettivamente il salvataggio di solo file splittati su server e il salvataggio generico su server).

Tale salvataggio può avvenire in modalità thread, che evita l'attesa del completamento dell'upload prima di proseguire con l'esecuzione del main, oppure in modalità sequenziale. Tale scelta viene specificata dall'utente in CONFIG.ini, nella sezione "FTP OPTIONS".

```
                   //ora inizia la fase di split
290     // ripeto il ciclo per ogni porta da scansionare
291     for(int z=0;z<ports.length;z++) {
296         String dumpSplit=ciclo.Split(tempDumpPath+"/preDump.cap",
              ports[z],packetBlacklist);
297         String ftPortoutname="Port-"+ports[z]+"-"+ip+"-
              "+date+"."+time+".cap";
298         String FTPSplitPath="Dump/Local_capture/"+ftPortoutname;
299         File splitFile = new File(dumpSplit);
300         File backup= new File(FTPSplitPath);
301         try {
302             FileChannel sourceChannel=null;
303             FileChannel destChannel=null;
304             FileInputStream fsin = new FileInputStream(splitFile);
305             FileOutputStream fsout = new FileOutputStream(backup);
306             sourceChannel = fsin.getChannel();
307             destChannel = fsout.getChannel();
308             destChannel.transferFrom(sourceChannel, 0,
                  sourceChannel.size());
309             sourceChannel.close();
310             destChannel.close();
311             fsin.close();
312             fsout.close();
313         }catch(Exception e) {}
316         if(DumpYN==1) {//se salvataggio FTP attivato salvo su server FTP
318             if(FLAG_th==1) { // thread save attivo
319                 synchronized (FTPSplitPath) {
321                     ThreadDumper dumperS=new ThreadDumper(host, usr,
             FTPKey,PortFTP, FTPSplitPath, ftPortoutname, folderFTPath);
322                     //prova salvataggio con thread
323                     new Thread(dumperS).start();//avvia il thread
324                     //file dump non filtrato salvato su server FTP
```

```
325             }}else { // thread save disattivato
327FTPDumper.SaveRemoteFile(FTPSplitPath,ftPortoutname,folderFTPath,inifile);
328        }}
```

Codice 5.2.9 - CSD: la classe Main()

Questa sezione (5.2.9) comprende una sequenza di funzioni analoga a quelle descritte nelle righe 225-282, con l'unica differenza che invece di partire da una scansione di rete, parte un ciclo *for* in cui ad ogni iterazione viene utilizzato il dump NOfilter, precedentemente salvato in "/Local_Capture", come sorgente di pacchetti da filtrare per la porta corrente, applicando le regole del filtro specificate nel file CONFIG.ini all'interno della sezione "SPLIT OPTION". Anche qui, nel caso fosse specificato, verrà utilizzata la modalità thread piuttosto che la modalità sequenziale, con diversi impatti sulla affidabilità del software e sulla sua efficienza potenziale in termini di pacchetti di rete non catturati tra un ciclo di cattura su device e il successivo.

```
351     ELAPSED=System.nanoTime();//aggiorna la variabile di controllo
353     contaCicli++;
354     if(ELAPSED<Limit) {
356         System.out.print("Ciclo"+contaCicli+" ; trascorsi:
               "+(ELAPSED/1000000000) +" - Limite(iniziale+nanosecondi
                richiesti): "+(Limit/1000000000)+"\n-> Mancano "+
    ((Limit/1000000000)-(ELAPSED/1000000000)) +" Secondi al termine.\n\n");
358  }  }//uscita dal while di cattura
360 while(Thread.activeCount()>2) {// aspetta completamento FTP
361     try {
362         Thread.sleep(2000);  //aspetta 2 secondi
363         System.out.println("Attesa Thread residui in elaborazione:"
                  +(Thread.activeCount()-2));
365         Thread.sleep(1000); // aspetta 1 secondo
366     } catch (InterruptedException e) {}}
368 System.out.println((Thread.activeCount()-2)+" Thread in lavorazione...");
375     System.out.println(":) \n\n-> CSD terminato! ");
376     Thread.sleep(300000); // aspetta 5 minuti e chiudi
377 } catch (InterruptedException e) {System.out.println("Exiting");}
378  }//End main
379}//End main Class
```

Codice 5.2.10 - CSD: la classe Main()

A questo punto viene aggiornata la variabile *elapsed*, ottenendo il valore del tempo corrente in nanosecondi, e stampando a video il messaggio di termine scansione, nel caso il valore di elapsed abbia superato il valore di *limit* calcolato precedentemente, causando l'uscita dal while. Quindi viene attesa la terminazione dei thread di upload (nel caso di file pesanti e/o connessione lenta, in combinazione alla modalità thread attiva) controllando ogni 3 secondi il numero di thread residui e verificando che scenda sotto la soglia minima per la terminazione.

5.3) CSD: la Classe CaptureCycle()

In questa sezione verranno approfondite le funzionalità implementate nella classe CaptureCycle, specialmente i metodi Split() e StartCycle().

```
23                          // CAMPI DELLA CLASSE
25  String local_ip;
26  public String Fcontrol;
//stringa contenente l'espressione regolare di filtro inserita nel file .ini
27  public int FLAG_Fcontrol;//valore intero: filtro flusso ON/OFF
28  public String local_MAC; //contiene l'indirizzo MAC del device di cattura
29  public String dumPath;   //path relativo a cui viene salvato il .cap
30  public PcapIf dEV;       //contiene il device per la cattura
31  public int minuti;       //contiene i minuti di scansione richiesti
32  public int maxpLenght;   //contiene la massima dimensione del pacchetti
33  public int timeOut;      //contiene il timeout di svuotamento del buffer
34  public int npack;        //contatore pacchetti del ciclo in corso
35  public int flag;         //contiene il flag per la modalit�romiscua
36  public String deviceName;//contiene il nome del device specificato
37  public StringBuilder errBuffer; //contiene il buffer degli errori
```

Codice 5.3.1 - CSD: la classe CaptureCycle()

Il codice 5.3.1 contiene una breve descrizione dei campi utilizzati dalla classe.

```
39                         // COSTRUTTORE DELLA CLASSE
public CaptureCycle(PcapIf dev, StringBuilder buff,Ini inifile, String path){
43      try {
44          this.local_MAC = Arrays.toString(dev.getHardwareAddress());
45      }catch(Exception e) {}
47      try {
48          this.local_ip="/"+InetAddress.getLocalHost().getHostAddress();
49      }catch(Exception e) {}
52      Fcontrol=inifile.get("SPLIT_OPTION", "FlowControl");
53      if(Fcontrol.equals("1")) {
54          this.FLAG_Fcontrol=1;
55      }else if(Fcontrol.equals("0")){
56          this.FLAG_Fcontrol=0;
57      }else {this.FLAG_Fcontrol=2;}
60      this.deviceName = dev.getName();
61      this.errBuffer = buff;
62      this.dumPath=path;
64      int prom=Integer.parseInt(inifile.get("INFO", "Promisc"));
65      if(prom!=Pcap.MODE_PROMISCUOUS)//se settato a 0->non promiscua
66          this.flag = Pcap.MODE_NON_PROMISCUOUS;
67      if(prom==Pcap.MODE_PROMISCUOUS)//se settato a 1->modalità promiscua
68          this.flag = Pcap.MODE_PROMISCUOUS;
70      try {
71          this.minuti=Integer.parseInt(inifile.get("INFO", "Cycle"));
72      }catch(Exception e) {
73          this.minuti=10;   //se valore non settato setto default 10 minuti
74          }
75      // gli altri valori vengono settati ai predefiniti
76      this.timeOut = 1000; //1000 millisecondi=1 secondo massimo timeout
77      this.maxpLenght = 64 * 1024;//pacchetto intero (64 KByte)}
```

Codice 5.3.2 - CSD: la classe CaptureCycle()

E' stato pensato un solo costruttore per la classe CaptureCycle, per cui sono richiesti:

- Un device di cattura *dev* di tipo PcapIf (definito nella classe JnetPcap);
- Un buffer *buff* di tipo StringBuilder, per i messaggi di errore;
- Una variabile *inifile* di tipo Ini(definito nella libreria Ini4J);
- Una stringa *path,* contenente il percorso della cartella in cui salvare;

Quindi, vengono lette le opzioni del filtro dal file *inifile* fornito, e vengono inizializzati i valori dei campi dell'oggetto istanziato rappresentanti gli indirizzi IP e MAC, utilizzati in seguito per riconoscere i pacchetti in arrivo o in uscita su una determinata porta associata alla VulnBox, nel caso di chiamata al metodo Split(). Questi due campi sono utilizzati anche per il riconoscimento dei pacchetti relativi a traffico generato dalla VulnBox.

I valori di timeout di dispatch dei pacchetti dal buffet della *JVM,* e della massima dimensione consentita nel salvataggio dei pacchetti catturati, sono invece impostati di default per catturare i pacchetti interamente, con un timeout di dispatch massimo pari a un secondo.

5.3.1) CaptureCycle : il metodo startCycle()

```
93  public String startCycle() throws IOException {
94      this.npack=0;
95      Pcap pcap = Pcap.openLive(this.deviceName, this.maxpLenght,
            this.flag, this.timeOut, this.errBuffer);
96      if (pcap == null) {
97          System.err.printf("Errore nell'apertura del device di cattura:"
98                     + this.errBuffer.toString()+"\n \n");
99                 return (this.dumPath+"/preDump.cap");
100     }
102     //variabile che indica quando chiudere la cattura
103     boolean timeToStop=false;
104     String ofile = this.dumPath+"/preDump.cap";
105     PcapDumper dumper = pcap.dumpOpen(ofile); //apertura dump
107     //variabile in cui viene salvato il tempo di inizio
108     long Init= System.nanoTime();
110     // inizio un loop che cattura 1 pacchetto per volta
111     // e continuo fino a che sono passati this.minuti dall'assegnamento
113     int cont=0;
114     while(!timeToStop) {
115         pcap.loop(1, dumper);// ad ogni iterazione del while scrivo
116            // un pacchetto nel dump, e così via fino alla fine del ciclo
117         cont++;
118         this.npack=cont;
119         if((System.nanoTime()-Init)/1000000000>this.minuti*60) {
```

```
120             System.out.println("CICLO FINITO (trascorsi "+
121               (System.nanoTime()-Init)/1000000000+" secondi dall'inizio");
122             timeToStop=true;
124          }
125          if(!timeToStop) {
126              System.out.println("Pacchetto in transito rilevato: (durata
                         ciclo richiesta:" + this.minuti*60 +
127                      " - secondi trascorsi: "+(System.nanoTime()-
                         Init)/1000000000+") -> Continua a catturare!");
128              System.out.println("Pacchetto numero "+cont);
129
130          }
132      }//fine del ciclo di ricezione e dump pacchetti
134      dumper.close(); // Chiudo il dump
136      pcap.close(); //chiudo il pcap generato inizialmente per la cattura
137         return ofile; //Restituisce il path del dump creato
138   }
```

Codice 5.3.3 - CSD: la classe CaptureCycle() - il metodo StartCycle()

Il metodo inizia con la dichiarazione della variabile nPack inizializzata a 0, utilizzata come contatore dei pacchetti rilevati, ed incrementata ad ogni pacchetto ricevuto. Quindi viene aperta un'istanza del monitor di cattura invocando il metodo statico openlive() contenuto nella classe Pcap (libreria JnetPcap).

A questo punto vengono inizializzate le variabili relative a segnale di fine ciclo, il path in cui salvare il dump, e un dumper associato al Pcap creato, utilizzato per il salvataggio dei pacchetti rilevati.

E' qui che inizia il while rappresentante il ciclo di cattura: ad ogni iterazione viene attesa la ricezione di un pacchetto, che viene scritto sul dump appena viene ricevuto dal buffer della JVM (allo scadere del timeout di dispatch); quindi viene calcolato il tempo trascorso dall'inizio del ciclo e il while viene ripetuto fino a che tale tempo non eccede la durata del ciclo specificata dall'utente e passata al metodo tramite il campo *this.minuti* (vedi costruttore - lettura valori CONFIG.ini). A questo punto il flusso di esecuzione esce dal *while,* i dumper e pcap creati vengono chiusi e il path del dump creato viene restituito al chiamante tramite *return*.

5.3.2) CaptureCycle() : il metodo Split()

```
158     public String Split(String sourceDumpPath, int portFilter,Vector
                <PcapPacket> packetList){
159      String splitted =this.dumPath+"/Port-"+portFilter+"-localDump.cap";
160     final Pcap splitter = Pcap.openOffline(sourceDumpPath,this.errBuffer);
161      if (splitter == null) {
162          System.err.println(this.errBuffer);
             // eventuale errore salvato nel buffer
```

```
163         return "Error"+splitted;
164     }//ora inizializzo il dumper per salvare i pacchetti in un nuovo .cap
165     PcapDumper splitdumper=splitter.dumpOpen(splitted);
166     //inizio del loop di scan per portfilter su questo pc
167     //(definisco in handler e in nextpacket algoritmo)
168     splitter.loop(Pcap.DISPATCH_BUFFER_FULL, new
          JPacketHandler<StringBuilder>() {
169         //inizializzo il tipo dei pacchetti di riferimento che user�r
170         //riconoscere se salvare il pacchetto in nextpacket()
171         final Tcp tcp = new Tcp();
172         final Ip4 ip4=new Ip4();
173         final Udp udp = new Udp();
174         final Ethernet eth=new Ethernet();
175
176         public void nextPacket(JPacket packet, StringBuilder errbuf) {
178             final PcapPacket pa = new PcapPacket(packet);
```

Codice 5.3.4 - CSD: la classe CaptureCycle() - il metodo Split()

Prima di tutto si può osservare come il metodo Split() sia dichiarato public e di tipo String. Inoltre, come si può notare alla riga 158, sono richiesti 3 parametri per il metodo:

- *portFilter* - un intero contenente la porta su cui splittare il dump sorgente;
- *sourceDumPath* - una stringa contenente il path in cui trovare il sorgente;
- *packetList* - un Vector di PcapPacket usato per i memorizzare la blacklist;

Quindi, viene inizializzata la stringa *splitted* contenente il path del dump temporaneo di output in cui salvare i pacchetti che superano il filtro: tale stringa verrà ritornata al chiamante al termine del metodo, a meno che venga incontrata una qualche eccezione che ritorni un valore diverso da *splitted.*

Viene poi creato un dumper associato a tale path, su cui invocare il metodo dump(), salvando il pacchetto sul file temporaneo di output quando necessario. A questo punto viene invocato il metodo loop(), che è il vero responsabile dell'inizio della scansione del file di cattura NOfilter.cap .

Per gestire le operazioni da eseguire alla ricezione di un pacchetto, vengono sovrascritti tramite *override* il JpacketHandler() (interfaccia di gestione dei pacchetti), e il metodo nextPacket() in esso contenuto.

Subito dopo la dichiarazione dell'handler sono instanziate delle variabili di tipo tcp,udp,ip,ethernet: queste verranno utilizzate per salvare le informazioni dell'header del

pacchetto corrente relative a un preciso protocollo, utilizzandone i campi per effettuare i diversi controlli necessari al funzionamento di CSD.

Alla riga 178 viene creata una variabile PcapPacket inizializzata tramite cast a partire dal pacchetto Jpacket corrente fornitoto come parametro dal metodo nextPacket(): tale variabile sarà utilizzata per salvare il pacchetto tramite il metodo splitter.dump(PcapPacket p, packet.getHeader()).

Il successivo blocco di codice (5.3.5), mostra la prima parte del controllo effettuato all'interno del metodo nextPacket() contenuto nel JpacketHandler() introdotto precedentemente.

```
180if(packet.hasHeader(Tcp.ID)&&packet.hasHeader(ip4)&&packet.hasHeader(eth))
            //Pacchetto con header tcp e header ipv4
181                {
182                    packet.getHeader(tcp);
183            //caso in cui il pacchetto E' in transito dalla porta richiesta
185                    if(tcp.source()==portFilter)
186                    {
187                        packet.getHeader(ip4);
188                        packet.getHeader(eth);
189                        String packetsrcIP=new String();
190  try { packetsrcIP=InetAddress.getByAddress(ip4.source()).toString();
192                        }catch(Exception e) {}
193         //se inoltre indirizzo ip sorgente corrisponde a questo pc- > uscita
195if(local_MAC.equals(Arrays.toString(eth.source()))||local_ip.equals
        (packetsrcIP)){
197         //ricavo il quartetto che descrive il pacchetto corrente
198                       byte[] SRCip=ip4.source();
199                       byte[] DSTip=ip4.destination();
200                       int SRCp=tcp.source();
201                       int DSTp=tcp.destination();
203                       if (FLAG_Fcontrol==0)
204                       {  //se controllo di flusso disattivato
205                          System.out.println("\nFlow control disattivato -
pacchetto TCP in uscita dalla porta "+portFilter+" \nSource IP: "+ local_ip);
206               System.out.println("frame #"+ packet.getFrameNumber());
207               System.out.println("tcp.dst_port="+ tcp.destination());
208               System.out.println("tcp.src_port="+ tcp.source());
209               System.out.println("tcp.ack="+ tcp.ack());
210               splitdumper.dump(pa.getCaptureHeader(), packet);
               // salva il pacchetto nel dump
212               }else if (FLAG_Fcontrol==1) {  //se controllo di flusso attivo
215                  System.out.println("\nFlow control attivo - Pacchetto TCP
              in uscita dalla porta "+portFilter+" \nSource IP: "+ local_ip);
216              System.out.println("frame #"+ packet.getFrameNumber());
217              System.out.println("tcp.dst_port="+ tcp.destination());
218              System.out.println("tcp.src_port="+ tcp.source());
219              System.out.println("tcp.ack="+ tcp.ack());
220     //ora controllo se per ogni p. in blacklist il quartetto corrisponde
221     //con quello del pacchetto corrente.Se corrisponde setta a true.
```

```
222             boolean inBlacklist=false;
223             PcapPacket blacklistPnt=null;
224             for(PcapPacket iter:packetList) {
225                if(iter.getHeader(tcp).source()==SRCp &&
          Arrays.equals(iter.getHeader(ip4).source(), SRCip))
226                    if(iter.getHeader(tcp).destination()==DSTp &&
              Arrays.equals(iter.getHeader(ip4).destination(), DSTip)) {
227                         inBlacklist=true;
228                         blacklistPnt=iter;
229                         }
            // controllo se dest corrisponde a source sia per IP che porta
            // per aggiungere anche le comunicazioni in senso opposto
233            if(iter.getHeader(tcp).source()==DSTp &&
          Arrays.equals(iter.getHeader(ip4).source(), DSTip))
234                if(iter.getHeader(tcp).destination()==SRCp &&
             Arrays.equals(iter.getHeader(ip4).destination(), SRCip)) {
235                     inBlacklist=true;
236                     blacklistPnt=iter;
237                }
238             }
    //fine scansione blacklist (se il pacchetto associato a un flusso in
    //blacklist setto inBlacklist=true). Inizia la fase di riconoscimento
    //dei flussi,blocco traffico gen. in uscita dalla vulbox (relativo
    //traffico di risposta=stesso flusso tcp=stesso ID) memorizzo i flussi
    //da scartare quando riconosco un SYN(da questo ip),oppure SYN-ACK. Li
    //rimuovo quando leggo un FIN
244         if(tcp.flags_SYN()&&!tcp.flags_ACK()&&!inBlacklist) {
246                 packetList.add(pa);
247                 System.out.println("rilevato traffico generato da "+
           local_ip+"\n-> Pacchetto scartato e aggiunto in blacklist");
248  //se rilevo che questo pc sta chiedendo di instaurare un flusso
249  // tcp, invece di inviare il pacchetto lo salvo nel vector dei pacchetti
250  // che rappresentano i quartetti da scartare(di riferimento)
251         }
252    //se pacchetto corrisponde a flusso in blacklist scartalo
253         if(inBlacklist) {
254             System.out.println("Pacchetto in uscita blacklist scartato");
255         }
256 //se invece sta mandando altro traffico && non E' presente in blacklist
257 //allora salva il pacchetto
258         if(!inBlacklist && !tcp.flags_SYN()) {
259             splitdumper.dump(pa.getCaptureHeader(), packet);
            // salva il pacchetto nel dump
260             System.out.println("Pacchetto conservato");
261         }
      //se FIN() settato a true->flusso tcp chiuso libera valore blacklist
264         if(tcp.flags_FIN()&& inBlacklist) {
266              packetList.remove(blacklistPnt);
267              System.out.println("Connessione in blacklist rimossa");
268         }
270      } // fine if controllo di flusso attivo
272      else if(FLAG_Fcontrol==2) {
      //se modalit�nalisi espressione regolare
274          String payload=new String(tcp.getPayload());
275          if (Pattern.matches(Fcontrol, payload)){
277              System.out.println("Corrispondenza trovata nella payload");
278              splitdumper.dump(pa.getCaptureHeader(), packet);
```

```
                 // salva pacchetto nel dump
280          }else {
                System.out.println("Corrispondenza non trovata nel pack");
281          }
282       } //fine if controllo espressione regolare
283     } // fine if IPsrc=Local_IP
284   } //fine if TCP src=PortFilter
```

Codice 5.3.5 - CSD: la classe CaptureCycle() - il metodo Split()

Nelle righe 180-195 si possono notare 3 strutture *if* annidate aventi rispettivamente lo scopo di riconoscere i pacchetti con header tcp,ip ed ethernet (usato come controllo aggiuntivo), inviati attraverso la porta in analisi (tcp.src==portFilter), e che siano in uscita dalla VulnBox (sourceIP.equals(local_IP)). Successivamente viene ricavato il quartetto di valori di sorgente (ip.src,tcp.src), e destinatario (ip.dst,tcp.dst) a partire dai valori contenuti all'interno dell'header tcp.

Quindi, alla riga 203 inizia la fase di riconoscimento del tipo di filtro impostato dall'utente: a seconda del valore contenuto in *FLAG_FControl* il flusso di esecuzione entra nella relativa struttura if attivata dalla condizione FLAG_FControl="numero", dove "numero" corrisponde al valore specificato nel file INI e letto dal costruttore della classe CaptureCycle().

Verranno scansionati in sequenza i casi:

- FLAG_FControl==0 -> significa che il filtro avanzato è disattivato. Siccome ho già selezionato il traffico con porta corrispondente, in uscita dalla VulnBox, in questo caso è sufficiente salvare il pacchetto nel dump stampando alcuni valori importanti dell'header su console (righe 203-210).

- FLAG_FControl==1 -> significa che è attivo il filtro con controllo di flusso. Questa modalità esclude tutti i pacchetti che sono relativi a flussi tcp richiesti dall'utente: per fare ciò, è stata sfruttato il riconoscimento dei messaggi di instaurazione di un flusso di comunicazione previsto dal protocollo TCP (in particolare dal cosiddetto *three way handshake*).

  Siccome il codice 5.3.5 tratta la parte relativa a pacchetti uscenti dalla VulnBox attraverso la porta in analisi, in questa sezione viene rilevata una richiesta di instaurazione quando SYN=1 & ACK=0, mentre un pacchetto avente FIN=1 denota un pacchetto di richiesta di chiusura di un flusso .

Per tenere traccia dei flussi da scartare (ovvero quelli per cui stata precedentemente inviata una richiesta con SYN=1&ACK=0), e dei flussi che sono stati chiusi (e quindi non vanno più scartati, a meno di nuove richieste in uscita su quel quartetto), in questi due casi il pacchetto, nel caso in cui non sia già presente, viene aggiunto (o, analogamente, eliminato) al Vector *packetList* di pacchetti passato come parametro al metodo.

Per realizzare questa funzione, le righe 222-238 includono un controllo, effettuato prima del riconoscimento dei flag ACK,SYN,FIN, che permette di capire se il pacchetto in analisi è già associato a flussi in blacklist (e in tal caso contrassegnandolo come da scartare settando a true il flag boolean *inBlacklist*), oppure se il pacchetto corrente non è associato a flussi in blacklist.

Si ricorda che per flusso in blacklist si intende un flusso con quartetto associato relativo a una comunicazione richiesta dalla VULNBOX, e quindi da scartare.

- FLAG_FControl==2 -> significa che è attivo il controllo di espressioni regolari contenute nella payload tcp. Questo controllo è contenuto nelle righe 272-282, che partono con la creazione di una stringa contenente la conversione in caratteri leggibili dell'array di byte rappresentante la payload contenuta nell'header, e proseguono con il suo confronto con la stringa inserita nella variabile FControl specificata nel file .ini e letta dal costruttore CaptureCycle().

  Come spiegato nel capitolo 4.5, questa funzione è realizzata attraverso il metodo Pattern.matches(Fcontrol,payload), fornito dalla libreria Java.util.regex.pattern.

```
286   // caso in cui il traffico è in entrata dalla porta in analisi
      // (devo anche controllare che il destinatario sia la VulnBox)
287   // controllo porta destinazione = a portfilter
288                if(tcp.destination()==portFilter)
289                {
290                    packet.getHeader(ip4);
291                    String packetdstIP=new String();
292                    try {
293       packetdstIP=InetAddress.getByAddress(ip4.destination()).toString();
294                    }catch(Exception e) {}
      //se indirizzo ip destinatario corrisponde a questo pc-> in ingresso
297            if(local_MAC.equals(Arrays.toString(eth.destination()))
                       ||local_ip.equals(packetdstIP))
298            {//inizia codice analogo a pacchetto in uscita
```

Codice 5.3.6 - CSD: la classe CaptureCycle() - il metodo Split()

Il segmento di codice 5.3.6 rappresenta il caso alternativo al codice 5.3.5 visto prima, cioè il caso in cui il pacchetto (avente header Tcp, Ip4, e Ethernet) sia in ingresso alla VULNBOX attraverso la porta tcp in analisi. Il codice è perciò analogo alle prime righe del codice 5.3.5, conl'unica differenza che porta, Local_ip e Local_mac vengono confrontati con i rispettivi valori nel campo destinatario del pacchetto invece che sorgente (accettando appunto solo i pacchetti destinati all'IP della VULBOX attraverso la porta tcp richiesta).

Come si può vedere dal codice sorgente (qui omesso in buona parte per il fatto che si tratta si una ripetizione), anche la parte seguente presenta una struttura molto simile a quella vista nel blocco 5.3.5.

```
298                         {
299                             byte[] SRCip=ip4.source();
300                             byte[] DSTip=ip4.destination();
301                             int SRCp=tcp.source();
302                             int DSTp=tcp.destination();
303
```

Codice 5.3.7 - CSD: la classe CaptureCycle() - il metodo Split()

Le righe iniziali di questo blocco, infatti iniziano allo stesso modo, con la creazione e inizializzazione delle variabili che rappresentano il quartetto del flusso tcp.

Quindi, ancora una volta inizia il controllo del valore contenuto in *FLAG_Fcontrol,* a seconda del quale viene eseguita la struttura *if* rappresentante la casistica corrispondente:

- FLAG_FControl==0 -> Questa porzione di codice viene omessa in quanto si comporta esattamente come quella descritta nel caso di pacchetto uscente dalla VULNBOX.
- FLAG_FControl==1 -> Significa che è attivo il filtro con controllo di flusso. In questo caso, il codice è molto simile a quello già visto per i pacchetti in uscita, e verranno citati quindi solo gli elementi che differiscono da esso in maniera sostanziale (ad esempio, la parte iniziale di controllo sulla presenza nella blacklist del flusso associato al pacchetto è identica a quella già vista, per il fatto che tale controllo è bidirezionale, e non dipende dal fatto che un pacchetto sia in ingresso o in uscita - righe 325-341).

```
// inizia la fase di riconoscimento dei flussi, e blocco del traffico
// in ingresso associato a una richiesta effettuata dalla vulbox
// (lo capisco vedendo risposta SYN,ACK, oppure se presente nella blacklist)
345       if(tcp.flags_SYN()&&tcp.flags_ACK()&&!inBlacklist) {
346       //se syn,ack, e pacchetto non in blacklist -> lo aggiungo
347             packetList.add(pa);
348             System.out.println("Rilevato traffico di risposta a "+
             local_ip+"\n-> flusso scartato e aggiunto in blacklist");
349            //se rilevo che questo pc sta chiedendo di instaurare un flusso
350            //tcp, invece di inviare il pacchetto lo salvo nel vector dei
               //pacchetti che rappresentano i quartetti da scartare
352       }
353       //se pacchetto in blacklist -> lo scarto
354       if(inBlacklist) {
355          System.out.println("Rilevato pacchetto in blacklist ->scartato");
356       }
357       //se invece sta mandando altro traffico non in blacklist
358       //allora salva il pacchetto
359       if(!inBlacklist && !tcp.flags_SYN() && !tcp.flags_FIN()) {
360          splitdumper.dump(pa.getCaptureHeader(), packet);
             //salva il pacchetto nel dump
361          System.out.println("Pacchetto conservato");
362       }
363       //se FIN() settato a true libera il valore dalla blacklist
364       if(tcp.flags_FIN()&& tcp.flags_ACK()&& inBlacklist) {
366            packetList.remove(blacklistPnt);
367           System.out.println("traffico relativo a flusso in blacklist,
            rimosso dalla blacklist per FIN=true");
368       }
370    } // fine if controllo di flusso attivo
371    else if(FLAG_Fcontrol==2) {
       // se modalità analisi espressione regolare
372
373         String payload=new String(tcp.getPayload());
374         if (Pattern.matches(Fcontrol, payload)){
376             System.out.println("Corrispondenza trovata TCP Payload");
377             splitdumper.dump(pa.getCaptureHeader(), packet);
                // salva pacchetto nel dump
378         }else {
379             System.out.println("Corrispondenza non trovata");
380         }
381       }// fine if espressione regolare
382     }// fine if IPdst=Local_IP
383    }// fine if DstPort = portfilter
384 }// fine if pacchetto con header TCP e IP4
```

Codice 5.3.8 - CSD: la classe CaptureCycle() - il metodo Split()

Osservando le righe 345-366 si può notare come la struttura sia molto simile a quella già vista nella sezione di codice 5.3.5, con alcune piccole differenze, siccome nel caso di pacchetto in arrivo devo considerare come da scartare i pacchetti relativi a flussi instaurati dalla VULNBOX, cioè:

1. SYN=1 & ACK=1 -> traffico di risposta a un syn(richiesta instaurazione della VulnBox), e quindi da scartare e mettere in blacklist nel caso non sia già stata rilevata la corrispondente SYN richiesta e il flusso risulti già in blacklist.
2. FIN=1 & ACK=1 -> risposta a richiesta di chiusura del flusso tcp. Il pacchetto viene scartato e il flusso rimosso dalla blacklist.
3. Se si tratta di un pacchetto corrispondente a flusso in blacklist (FIN,SYN,ACK qualunque & inBlacklist==true), allora semplicemente il pacchetto viene scartato.
4. Se invece il pacchetto non è associato a flusso in blacklist (e nemmeno alle casistiche 1-2-3 che lo escludono) allora viene salvato nel dump.

- FLAG_FControl==2 -> Questa porzione di codice (righe 371-381) si comporta esattamente come quella descritta nel caso di pacchetto uscente dalla VULNBOX.

Quindi, la fase di controllo relativa a pacchetti con header tcp,ip, e ethernet si conclude, e inizia la fase relativa a pacchetti con header udp.

Questa sezione è molto simile alla precedente, ma molto più semplice in quanto si limita ai casi FLAG_Fcontrol=0 e FLAG_Fcontrol=2 (non è stato previsto un metodo per il riconoscimento dei flussi udp, funzionanti in maniera leggermente diversa).

A tal proposito, si lascia come spunto per progetti futuri l'idea di inserire un controllo di flusso anche nel caso in cui venga rilevato un pacchetto udp.

Il codice relativo al filtraggio di pacchetti udp è omesso in quanto praticamente identico a quello visto nei casi FLAG_Fcontrol=0 e FLAG_Fcontrol=2 visti con il filtro di pacchetti tcp, salvo l'utilizzo della classe udp al posto della classe tcp nel calcolo dei valori.

```
446         }//end nextpacket()
447
448     }, this.errBuffer);//end loop scansione file
449     splitdumper.close();//chiusura dump
450     splitter.close(); //chiusura pcap
451     return splitted;
452     //nome del file contenente lo split
453 }//end metodo
```

Codice 5.3.9 - CSD: la classe CaptureCycle() - il metodo Split()

Subito dopo il termine della sezione relativa a pacchetti con header udp, terminano anche le definizioni del metodo nextPacket (si ricorda che esso viene eseguito per ogni pacchetto ricevuto dal buffer di cattura creato da JnetPcap attraverso la JVM), e del JpacketHandler() che lo contiene.

Alla riga 448 termina il loop di scansione del file (avendo invocato *loop*() con il parametro *dispatch_FULL* si è indicato a JnetPcap di scansionare il dump sorgente fino al termine dei pacchetti in esso contenuti). Quindi, vengono chiusi anche il dumper e l'oggetto Pcap creato all'inizio, a cui è associata la scansione.

Splitted, contenente il path del file in cui sono stati salvati i pacchetti dal metodo Split(), viene restituito sotto forma di String al chiamante del metodo.

5.4) CSD: la Classe Dumper()

Questa classe prevede un costruttore che inizializza i dati per la connessione al server con FTPS(protocollo SSL), un metodo statico Check() che controlla la correttezza dei dati immessi, e un metodo SaveRemoteFile() che si occupa di salvare un file su server con protocollo FTPS.

```
1    package CSD;
3    import java.io.File;
4    import java.io.FileInputStream;
5    import java.io.IOException;
6    import java.io.InputStream;
7    import org.apache.commons.net.FTP.FTP;
8    import org.apache.commons.net.FTP.FTPSClient;
9    import org.apache.commons.net.FTP.FTPClient;
10   import org.ini4j.Ini;
11
13   public class Dumper {
15       // campi
17       public String hostFTP;
18       public String usr;
19       public String psw;
20       public int Port;
22       public FTPSClient FTPS;
23       public String protocol = "SSL";
25       //costruttore
27       public Dumper(String host,String user, String password,int PORT) {
28           this.hostFTP=host;
29           this.usr=user;
30           this.psw=password;
31           this.Port=PORT;
32       }
```

Codice 5.4.1 - CSD: la classe Dumper()

5.4.1) Dumper() : il metodo checkFTP()

```
34                               //metodi
35
36       public synchronized static boolean CheckFTP(String server,String
     usr,String inskey,int port) {
37
38           FTPSClient FTPS = new FTPSClient();
39           boolean res=false;
40           try {
41               FTPS.connect(server);
42               res= FTPS.login(usr, inskey);
43               }catch(Exception e){
44               return false;
45           }
46           try {
47               if (FTPS.isConnected()) {
48                   FTPS.logout();
49                   FTPS.disconnect();
50               }
```

```
51          } catch (IOException ex) {
52              ex.printStackTrace();
53              System.out.println("ERRORE DISCONNESSIONE FTPS");
54          }
55          return res;
57      }//end CheckFTP()
58
```

Codice 5.4.2 - CSD: la classe Dumper() - il metodo CheckFTP()

Nel codice 5.4.2 è contenuto il metodo CheckFTP(). Tale metodo, essendo statico, può essere chiamato anche senza aver prima creato un oggetto di tipo dumper su cui invocarlo: questo ha permesso di inserire la funzione di check nella parte iniziale del Main (come visto nel capitolo 5.2) in maniera indipendente dalla creazione di variabili dumper in memoria.

Prima di tutto viene creata una variabile di tipo FTPSClient(), definita in org.apache.commons.net (vedi capitolo 4.4), e una variabile boolean res che assumerà il valore *true* o *false,* a seconda che il controllo abbia esito positivo o negativo. Il valore di tale variabile booleana è settato nella struttura *try-catch* che racchiude i tentativi di accesso e disconnessione dal server FTP.

5.4.2) Dumper() : il metodo SaveRemoteFile()

```
70      public synchronized int SaveRemoteFile(String filename, String
    FTPname, String folderFTPPath, Ini IniFile) {
71          String FTPdir=IniFile.get("FTP_LOGIN", "PathFTP");
72
73          FTPS=new FTPSClient(protocol);
74          try {
75              FTPS.connect(this.hostFTP, this.Port);
76              FTPS.login(this.usr, this.psw);
77              FTPS.enterLocalPassiveMode();
78              FTPS.setFileType(FTP.BINARY_FILE_TYPE);
79              //testa esistenza cartella specificata in CONFIG.ini)
80              FTPS.changeWorkingDirectory(FTPdir);
81              int returnCodeA = FTPS.getReplyCode();
                //se la directory specificata in ini non esiste
82              if (returnCodeA == 550) {

83                  System.out.println("Check della directory:  "+ FTPdir);
84                  FTPS.makeDirectory(FTPdir);
85              }
86              FTPS.changeWorkingDirectory(FTPdir);
88              //carico il file usando inputstream
89              File LocalFile = new File(filename);
90              String RemoteFile = FTPname;
91              InputStream inputStream = new FileInputStream(LocalFile);
92              System.out.println("Caricamento di: +FTPdir+"/"+RemoteFile);
93              boolean done = FTPS.storeFile(RemoteFile, inputStream);
94              inputStream.close();
```

```
95              if (done) {
96                  System.out.printf(" -> File caricato correttamente!\n");
97              }
98          } catch (IOException ex) {
99              System.out.println("Error: " + ex.getMessage());
100             ex.printStackTrace();
101             return 1;
102         } finally {
103             try {
104                 if (FTPS.isConnected()) {
105                     FTPS.logout();
106                     FTPS.disconnect();
107                 }
108             } catch (IOException ex) {
109                 ex.printStackTrace();
110                 System.out.println("ERRORE DISCONNESSIONE FTPS");
111                 return 1;
112             }
113         }// End finally
114         return 0;
115     }// End metodo SaveRemoteFile()
```

Codice 5.4.3 - CSD: la classe Dumper() - il metodo SaveRemoteFile()

Nelle righe 77-78, vengono settati i parametri relativi alla connessione passiva al server (utilizzata per evitare il filtraggio dei dati da parte di eventuali firewall presenti) e al tipo di file da trasmettere (file di tipo *binary*).

Successivamente, nelle righe 79-86 viene testata l'esistenza della directory specificata in cui salvare il file: nel caso in cui tale directory non esiste viene creata, prima di proseguire con l'upload.

Quindi, dopo aver inizializzato le variabili di tipo File(),il FileInputStream() ad esso associato, e la stringa contenente il nome del file da salvare su server, alla riga 93 viene chiamato il metodo FTPS.storeFile(), definito nella libreria di *Apache*, che si occupa di salvare effettivamente il file. Inoltre, alla stessa riga viene salvato il valore booleano restituito da tale metodo, utile al fine di segnalare all'utente eventuali anomalie durante il caricamento del file.

A questo punto, un'ultima struttura finally-try-catch si occupa di chiudere le connessioni aperte e catturare eventuali eccezioni lanciate durante tale operazione.

5.5) CSD: la Classe ThreadDumper()

```
1 package CSD;
2
3 import java.io.FileReader;
4
5 import org.ini4j.Ini;
6
7 public class ThreadDumper implements Runnable{
8
9   public Dumper dumper;
10  public String hostFTP;
11  public String usr;
12  public String psw;
13  public int Port;
14  public String FILEin; //path file input da copiare
15  public String FTPout;  //nome nuovo file output su FTP
16  public String folderFTPPath; //path della cartella in cui salvare
17
18   public ThreadDumper(String host,String user, String password,int
   PORT,String Inpath, String outN, String PathFD) {
19      hostFTP=host;
20      usr=user;
21      psw=password;
22      Port=PORT;
23      FILEin=Inpath;
24      FTPout=outN;
25      folderFTPPath=PathFD;
26  }
27
28  public synchronized void run() {
29      Ini INI=null;
30      try {
31          INI= new Ini(new FileReader("CONFIG.ini"));
32      }catch (Exception e) {
34          System.out.println("Errore nell'apertura di CONFIG.ini");
35      }
37      dumper=new Dumper(hostFTP, usr, psw, Port);
38      synchronized (dumper) {
39          dumper.SaveRemoteFile(FILEin, FTPout, folderFTPPath,INI );
40      }
42   }
43 }
```

Codice 5.5.1 - CSD: la classe ThreadDumper()

Questa classe è responsabile del savataggio in modalità thread, come visto nel capitolo 5.2, durante la presentazione del Main(). Per realizzare questa funzione in maniera semplice, è stato scelto di definire una classe che implementa l'interfaccia runnable e utilizza la classe dumper già introdotta nel capitolo 5.4. Per questo scopo è stato inoltre deciso di dichiarare *synchronized* il metodo run(): in tal modo è possibile eseguire su una sola volta per istanza il metodo, fornendo un'ulteriore sicurezza sulla concorrenza in

aggiunta a quella fornita dal suo utilizzo nel main in concomitanza con un path sorgente dato da nome unico e dichiarato anch'esso synchronized. In tal modo è stato possibile evitare, mediante una tripla sicurezza, la complessa fase di test che avrebbe richiesto un'implementazione tecnicamente "corretta" della modalità Thread.

5.6) CSD: la classe charAnimationPrinter

Questa classe prevede un costruttore vuoto, e implementa una funzionalità aggiuntiva, a scopo puramente grafico. Utilizzando il metodo sleep(int ms_time) invocabile su oggetti di tipo Thread, è possibile, data una stringa passata al metodo print, stampare in sequenza i caratteri che compongono tale stringa, intervallandoli di un valore proporzionale a quello fornito come parametro dal chiamante del metodo (circa 5/4 di tale tempo in millisecondi). Tale valore scende a 1/4 di quello fornito nel caso in cui il carattere corrente sia uno spazio.

```
1   package CSD;
2
3   import java.util.StringTokenizer;
4
5   public class charAnimationPrinter {
7       public charAnimationPrinter() {
9       }
11      public void Print(String msg,int interval) {
12      if(interval==0) {
13       System.out.println(msg);
14      }else {
15   StringTokenizer out_msg_tokenizer=new StringTokenizer(msg, "\n");
16   try
17   {
18       for(int q=0;q<msg.length();q++){
19           String token=out_msg_tokenizer.nextToken();
20           for(int r=0;r<token.length();r++) {
21               String character=token.substring(r, r+1);
22               Thread.sleep(interval/4);
                 //aspetta 1/4 del tempo specificato
23               System.out.printf(character);
24               if(!character.equals(" ")) {
25                   Thread.sleep(interval);
                     // aspetta il tempo specificato
26               }
27           }
29       }
30      }catch (Exception e) {}
31     }
32   }
```

Codice 5.6.1 - CSD: la classe charAnimationPrinter()

5.7) CSD: il File CONFIG.ini

Qui di seguito verrà inserita una breve descrizione del file CONFIG.ini, il file avente il compito di passare a CSD l'input per la personalizzazione del comportamento del software scelto dall'utente (complemento ai commenti consultabili nel file CONFIG.ini stesso).

```
;------------------------------------------------
;---------  IMPOSTAZIONI SCANSIONE --------------
;------------------------------------------------
 [INFO]
Promisc=0
Cycle=1
CaptureUntil=2
;------------------------------------------------
;----------  IMPOSTAZIONI FILTRO ----------------
;------------------------------------------------
 [SPLIT_OPTION]
Port=22,23,24
FlowControl=0
;------------------------------------------------
;--------------  DATI LOGIN FTP -----------------
;------------------------------------------------
 [FTP_LOGIN]
HostFTP=FTP.cycliccaptureAlessioMoraschini.altervista.org
UsrFTP=cycliccaptureAlessioMoraschini
KeyFTP=XXXXXXXXXXXXXXX
PortFTP=21
PathFTP=cartellaDiCattura
;------------------------------------------------
;------------  OPZIONI FTP  --------------------
;------------------------------------------------
[FTP_OPTION]
DumpFTP=1
DumpOnlySplit=0
Threaded_version_Beta=1
```

Codice 5.7.1 - CSD: il file CONFIG.ini

Come accennato, il file di configurazione è strutturato in 4 diverse parti, utilizzate sia dal main, che dalle classi *Dumper* e *CaptureCycle*. In esso è possibile definire i parametri della scansione e cattura dei pacchetti (dump NOfilter), ovvero la modalità, la durata del singolo ciclo di scansione e upload, e la scadenza in minuti, dopo la quale CSD verrà arrestato. Nella sezione delle impostazioni del filtro è possibile specificare per quali porte creare un file separato (e filtrato) con il metodo Split() precedentemente definito (capitolo 5.3.2), oltre al tipo di filtro da applicare alla scansione di split().

Le due sezioni successive trattano i dati relativi al server FTP e alla modalità con cui vengono salvati i dati in esso: la prima parte è inserita nella sezione dei dati di login al server, mentre la seconda parte è inserita nelle opzioni FTP. Tra le opzioni di caricamento FTP, è possibile:

1. Disattivare il caricamento su server e salvare i dump solo nella cartella locale.
2. Disattivare il caricamento dei dump "NOfilter", che potrebbero risultare superflui e potenzialmente troppo pesanti per l'utilizzo che l'utente necessita.
3. Scegliere la modalità di salvataggio Thread, o sequenziale.

6) DMAD – Download Merge Analyze and Dump

In questo capitolo verrà presentata nel dettaglio l'applicazione client per l'analisi dei dati creati da CSD, il software introdotto nel precedente capitolo. Tale applicativo prende il nome di DMAD (acronimo di Download Merge Analyze and Dump), e consiste in un software ad interfaccia grafica che permette di connettersi al server FTP, scaricare, analizzare, filtrare, ed esportare un unico file contenente i pacchetti di interesse, oltre ad alcune funzioni secondarie.

6.1) DMAD: il progetto

Siccome l'applicazione DMAD racchiude diverse funzionalità molto simili a quelle presenti in CSD, è stato scelto di riutilizzare parte del codice creato (specialmente quello relativo all'accesso al server FTP, filtro dei pacchetti, la lettura da file .cap), adattandolo ai nuovi mezzi di interazione con l'utente forniti da *Swing* con l'aiuto del tool Windowbuilder, modificando/eliminando alcune componenti per soddisfare i requisiti dell'analisi, come verrà spiegato in dettaglio nel capitolo 6.

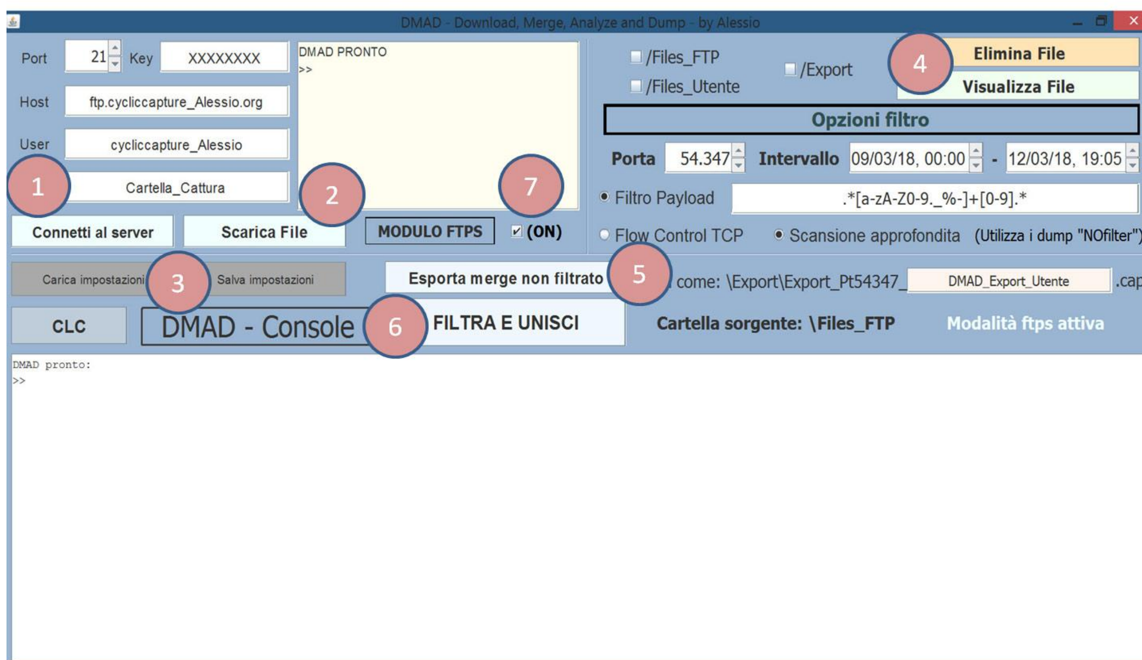

Figura 6.1 - Schermata principale e funzioni collegate ai pulsanti/selettori

Per quanto riguarda questa sezione del progetto di tesi, prima di approfondire la progettazione vera e propria del software client di analisi, sono state individuate le seguenti funzionalità:

1. Connessione al server FTP e visualizzazione dei file presenti.
2. Download dei file corrispondenti al periodo selezionato.
3. Funzionalità di salvataggio/caricamento impostazioni DMAD (da CONFIG.ini).
4. Funzionalità di visualizzazione/eliminazione file locali ed export creati.
5. Creazione file unico dai file .cap non filtrati per porta.
6. Creazione file unico dai file .cap filtrati per porta, dati i settaggi specificati.
7. Possibilità di eseguire analisi da file locali invece che scaricarli via FTPS.

Queste operazioni sono incluse nelle azioni collegate a pulsanti e selettori della GUI, come indicato in figura 6.1. La GUI è stata progettata con l'aiuto del tool Windowbuilder, e successivamente sono stati definiti manualmente degli *actionListener()* associati a tali elementi, in cui è stato inserito il codice relativo alle funzioni individuate. All'interno degli actionListener, inoltre, sono richiamate le classi CaptureCycle(), CapMerger(), e Dumper().

Ed ecco lo schema delle classi presenti nell'applicazione DMAD, con una breve modellazione grafica aggiuntiva sull'interazione con l'utente e sulla lettura/scrittura del file di configurazione (da notare a tal proposito la bidirezionalità della freccia).

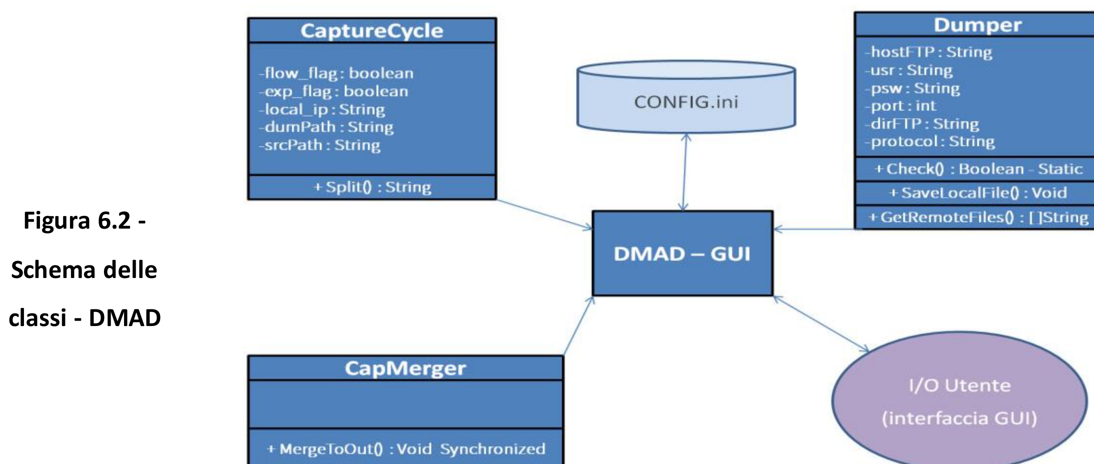

Figura 6.2 - Schema delle classi - DMAD

6.2) DMAD: la GUI (Graphic User Interface)

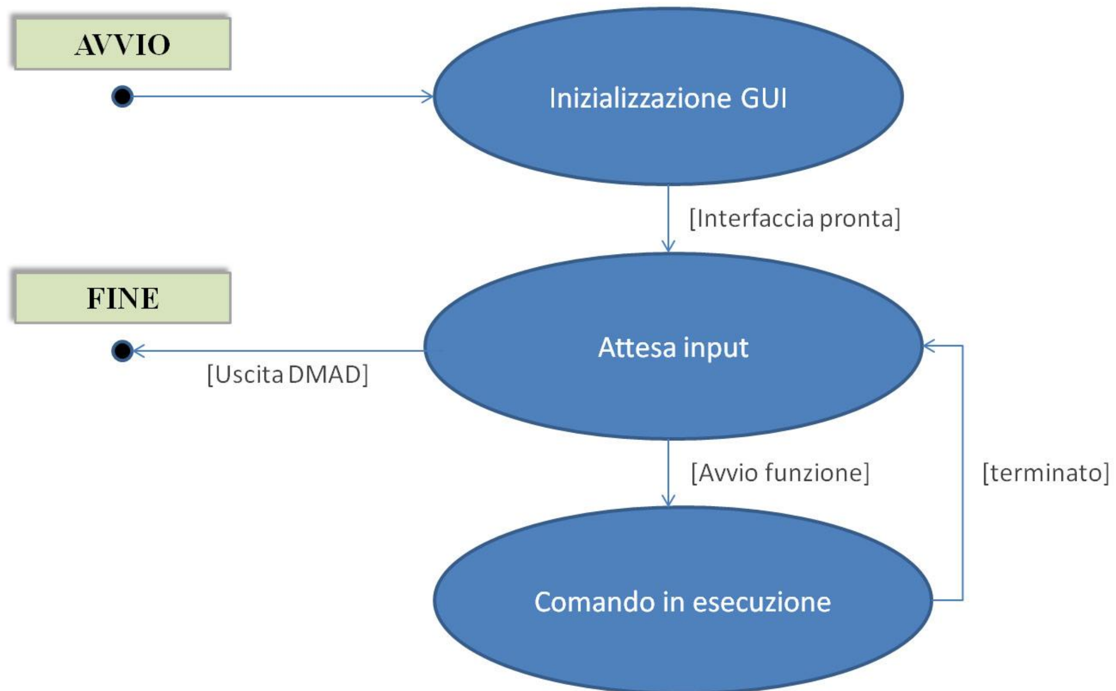

Figura 6.3 - Diagramma degli stati - DMAD(GUI)

L'interfaccia grafica racchiude la maggior parte delle funzionalità presenti nel software DMAD. Questo è dovuto al fatto che DMAD nasce come software minimale, e con poche funzionalità ripetute, e per questo sono state incluse in oggetti creati ad hoc solo alcune funzionalità chiave, minimizzando il codice e massimizzando la mantenibilità solo dove è stato reputato realmente necessario. Le classi CaptureCycle e CapMerger, in particolare, racchiudono le funzionalità necessarie al download dei file e alla loro unione in un file unico, operazioni che avrebbero contribuito alla stesura di un codice di qualità peggiore, più lungo, meno leggibile, e scomodo da modificare, se non inserite in oggetti e metodi richiamabili all'interno del codice.

Essendo realizzata direttamente con Windowbuilder, le parti del codice sorgente relative all'inizializzazione dei componenti e alla creazione della classe DMAD-GUI, sono state realizzate automaticamente dal tool di Eclipse. Il codice realizzato per la realizzazione

delle funzioni necessarie allo scopo del progetto è stato quindi inserito manualmente nel sorgente generato da Windowbuilder, attraverso la creazione di porzioni di codice associate agli elementi della GUI, principalmente sotto forma di *ActionListener()*.

Come si può vedere dalla figura 6.3, l'interfaccia grafica, una volta inizializzata, è progettata per la ricezione-esecuzione ciclica dei comandi presentati in figura 6.1 e nel capitolo 6.1. Nei capitoli a seguire, verranno prese in analisi una ad una le funzioni collegate ai pulsanti e selettori presenti nella GUI (consultabili in figura 6.1).

Per presentare all'utente i messaggi relativi all'esecuzione di tali funzioni, sono state previste due finestre, istanziate sotto forma di JTextArea(oggetto previsto dalla classe *swing*): una (csNaLog) per la visualizzazione dei file e del contenuto dei pacchetti, oltre a informazioni più dettagliate sull'esecuzione del comando; l'altra (ConsoleFTP) avente la funzione di LOG, per la visualizzazione rapida delle informazioni di supporto all'esecuzione del comando corrente.

6.2.1) GUI: la funzione LoadIni()

```
491         JButton LoadIni = new JButton("Carica impostazioni");
492         LoadIni.setBounds(7, 235, 181, 35);
493         LoadIni.addActionListener(new ActionListener() {
494             public void actionPerformed(ActionEvent e) {
495                 Ini inifile=new Ini();
496        DateFormat parser = new SimpleDateFormat ("yyyy_MM_dd_HH_mm_ss");
497                 try {
498                     Ini conFile = new Ini(new FileReader("CONFIG.ini"));
499                     inifile=conFile;
500                 }catch(Exception ex) {
501                     ConsoleFTP.append("\n>> Errore nel caricamento ");
502                 }
503                 DIR.setText(inifile.get("FTP_LOGIN", "PathFTP"));
504                 HOST.setText(inifile.get("FTP_LOGIN", "HostFTP"));
505                 USER.setText(inifile.get("FTP_LOGIN", "UsrFTP"));
       EXPR_FILTER.setText(inifile.get("SPLIT_OPTION","EXPRESSION_FILTER"));
507                 String k=inifile.get("FTP_LOGIN","KeyFTP");
508                 if(k.equals("")) {
509                     ConsoleFTP.append("\n >> Chiave FTP non rilevata ");
510                 }else {
511                     KEY.setText(k);
513                 }
514                 try {
 Data_INIZIO.setValue(parser.parse(inifile.get("SPLIT_OPTION", "Date_min")));
516
   Data_FINE.setValue(parser.parse(inifile.get("SPLIT_OPTION", "Date_max")));
517                 }catch(Exception exc) {}
519              //KEY.setText(inifile.get("FTP_LOGIN", "KeyFTP"));
520              PORTFTP.setValue(Integer.valueOf(inifile.get("FTP_LOGIN",
                 "PortFTP")));
```

```
521            portFilter.setValue(Integer.valueOf(inifile.get("SPLIT_OPTION",
               "Port")));
522                  ConsoleFTP.append("\n>> Impostazioni caricate!");
523               ConsoleFTP.setSelectionStart(ConsoleFTP.getText().length());
525              }
526          });
527          LoadIni.setFont(new Font("SansSerif", Font.PLAIN, 13));
528          LoadIni.setBackground(new Color(169, 169, 169));
529          frmCsnaBy.getContentPane().add(LoadIni);
```

Codice 6.2.1 - la GUI: la funzione LoadIni()

La funzione LoadIni() è associata al *JButton LoadIni*, riportato nella sezione codice 6.2.1. contenente le istruzioni necessarie per il salvataggio su file con estensione .ini dei valori contenuti nei campi Host, User, Key, Porta, Directory, date del filtro, porta del filtro ed espressione del filtro, se specificata.

In particolare, si notino le righe 496/514/516: esse contengono il codice per il riconoscimento delle date presenti (in formato String con rappresentazione analoga) nel file CONFIG.ini, attraverso un parser che riconosce le date in formato *String*, aventi struttura del tipo `yyyy_MM_dd_HH_mm_ss`.

6.2.2) GUI: la funzione SaveIni()

```
436  JButton SaveIni = new JButton("Salva impostazioni");
437  SaveIni.setBounds(198, 235, 181, 35);
438  SaveIni.addActionListener(new ActionListener() {
439     public void actionPerformed(ActionEvent arg0) {
440        Wini ini;
441        boolean cond=HOST.getText().equals("")||USER.getText().equals("");
442        if(!cond) {
443           try {
444               ConsoleFTP.append("\n>> Salvataggio...");
445               ini=new Wini(new File("CONFIG.ini"));
446               ini.put("FTP_LOGIN", "PathFTP",DIR.getText());
447        ini.put("SPLIT_OPTION","EXPRESSION_FILTER",EXPR_FILTER.getText());
448               ini.put("FTP_LOGIN", "HostFTP",HOST.getText());
449               ini.put("FTP_LOGIN", "UsrFTP",USER.getText());
450               if(!KEY.getText().equals("")) {
451                  int YN=JOptionPane.showConfirmDialog
                        (frmCsnaBy,JOptionPane.YES_NO_OPTION);
452                  if(YN==JOptionPane.YES_OPTION) {
453                      ini.put("FTP_LOGIN", "KeyFTP",KEY.getText());
454                      ConsoleFTP.append("\n  Chiave FTP -> CONFIG.ini\n");
455                  }else {
456                      ini.put("FTP_LOGIN", "KeyFTP","");
457                      ConsoleFTP.append("\n >> Chiave non salvata ;)\n");
458                  }
459               }else {
460                  ini.put("FTP_LOGIN", "KeyFTP",KEY.getText());
461                  ConsoleFTP.append("\n >> Chiave FTP resettata.\n");
462               }
```

```
463                 String datemin = new
SimpleDateFormat("yyyy_MM_dd_HH_mm_ss").format(Data_INIZIO.getValue());
464                 String datemax = new
SimpleDateFormat("yyyy_MM_dd_HH_mm_ss").format(Data_FINE.getValue());
465                 ini.put("SPLIT_OPTION", "Date_min",datemin);
466                 ini.put("SPLIT_OPTION", "Date_max",datemax);
468              ini.put("FTP_LOGIN", "PortFTP",(Integer)PORTFTP.getValue());
469              ini.put("SPLIT_OPTION","Port",(Integer)portFilter.getValue());
471                 ini.store();
472                 ConsoleFTP.append("\n>> Impostazioni salvate!");
473                ConsoleFTP.setSelectionStart(ConsoleFTP.getText().length());
474                 csNaLOG.setSelectionStart(csNaLOG.getText().length());
475             }catch(Exception e) {
476                 ConsoleFTP.append("\n>> Errore nel salvataggio!");
477                ConsoleFTP.setSelectionStart(ConsoleFTP.getText().length());
478             }
479          }else {
480             ConsoleFTP.append("\n>> campi HOST e USER vuoti.");
481             ConsoleFTP.setSelectionStart(ConsoleFTP.getText().length());
482                   }
483
```

Codice 6.2.2 - la GUI: la funzione SaveIni()

Il codice 6.2.2 mostra il funzionamento dell'azione collegata al pulsante "Salva impostazioni", e consiste in una serie di funzioni analoghe a quelle svolte dalla precedente *LoadIni* (capitolo 6.2.1). Come già visto diverse volte in CSD, si può leggere da file .ini con il metodo inifile.get("SECTION","ID"). Nell'ambito della progettazione di DMAD, si è riscontrata la necessità di salvare le impostazioni scelte, oltre a leggerle dal file di configurazione (in CSD non era invece prevista la funzione di scrittura su file di configurazione).

Per fare questo, si è utilizzata la classe *Wini* definita nella libreria Ini4j (vedi capitolo 4.3, invocando il metodo Winifile.put("SECTION","ID",String valore_elementoGUI) per ogni valore da aggiornare. Questa istruzione scrive il relativo valore contenuto nell'interfaccia utente di DMAD all'interno del file CONFIG (righe 445-471), all'interno della sezione specificata in "SECTION".

Inoltre, si può osservare il controllo sull'input (chiave FTP) effettuato alle righe 451-462, che si occupa di chiedere all'utente conferma per il salvataggio della chiave FTP nel file di configurazione, essendo tale file scritto in chiaro ed essendo quindi sconsigliata la pratica di salvare le password utilizzate. Nel caso in cui la password non sia specificata la funzione LoadIni() resetta l'eventuale valore presente nel file CONFIG.ini.

6.2.3) GUI: la funzione Check ()

```
253        JButton Check = new JButton("Connetti al server");
254        Check.setBounds(7, 187, 181, 33);
255        Check.addActionListener(new ActionListener() {
256            public void actionPerformed(ActionEvent arg0) {
257                ConsoleFTP.append("\n>> Lettura dati\n");
258                Check.setEnabled(false);
259                String host=HOST.getText();
260                String user=USER.getText();
261                String key=KEY.getText();
262                String FTPdir=DIR.getText();
263                int portFTP=21;
264                try {
265                    portFTP=((Integer)PORTFTP.getValue());
266                }catch(Exception e) {
269                }
272                final int p=portFTP;
273                new Thread(new Runnable() {
                   //apro un thread per non bloccare il messaggio iniziale
274                    public void run() {
276                boolean check=false;
277                Dumper dumper;
278                String []file=null;
279                try {
280                    dumper=new Dumper(host, user, key, p);
281                    check=Dumper.CheckFTP(host, user, key, p);
282                    file=dumper.GetRemoteFiles("/"+FTPdir);
283                    Check.setEnabled(true);
284                }catch (Exception e) {
289                    Check.setEnabled(true);
290                }
291                Check.setEnabled(true);
292                StringTokenizer tokenizer;
294                if(check) {
297                csNaLOG.append(">> Lettura nella cartella: "+FTPdir+"\n");
300                }
301                else {
302                    ConsoleFTP.setText(ConsoleFTP.getText()+"Fallito.");
304                    csNaLOG.append("\n>> Connessione fallita.\n");
306                }
307                for (String s:file) {
308                    tokenizer=new StringTokenizer(s, "-");
309                    String token=tokenizer.nextToken();
310                    if(token.equals("NOfilter")) {
311                        csNaLOG.append(s+": rilevato DUMP non filtrato");
312                    }else if(token.equals("Port")) {
313                        csNaLOG.append(s+": rilevato DUMP filtrato");
314                    }
316                    else {
317                        csNaLOG.append(s+" : File generico/directory");
318                    }}
320                ConsoleFTP.append("\n\n>> FINE, DMAD pronto:");
321              ConsoleFTP.setSelectionStart(ConsoleFTP.getText().length());
322                csNaLOG.setSelectionStart(csNaLOG.getText().length());
324                }}).start();  // avvio il thread appena definito}});
```

Codice 6.2.3 - la GUI: la funzione Check()

Il codice 6.2.3 presenta la funzione "Connetti al server"(1) rappresentata nella figura 6.1. Tale funzione si occupa di connettere l'utente al server FTP (dati i parametri specificati per la connessione via FTPS), ed eseguire uno scan dei file presenti all'interno della cartella specificata dall'utente nel campo "DIR" (vedi figura 6.1). Inoltre, per ogni file scansionato, DMAD riconosce se si tratta di un dump di tipo NOfilter, di un dump filtrato per una precisa porta, oppure un'altro tipo di file (o directory), come si può osservare alle righe 307-319 del codice 6.2.3.

Alla riga 273 si nota l'apertura della definizione di un oggetto della classe thread: esso è stato introdotto per evitare che la visualizzazione dell'output nella GUI restasse "bloccata" fino al termine dell'esecuzione della funzione check().

6.2.4) GUI: la funzione DownloadFiles()

Questo capitolo si occuperà di approfondire la funzionalità di salvataggio dei dump (filtrati e non) che corrispondono al periodo specificato negli appositi campi presenti sulla destra dell'interfaccia grafica presentata in figura 6.1. Tale funzionalità corrisponde al tasto di tipo JButton (definito nella libreria *Swing*), come si può osservare nelle prime righe del codice 6.2.4.

```
544 JButton DownloadDump = new JButton("Scarica File");
545 DownloadDump.setBounds(198, 187, 181, 33);
546 DownloadDump.addActionListener(new ActionListener() {
547 public void actionPerformed(ActionEvent arg0) {
548   DownloadDump.setEnabled(false);
549   ConsoleFTP.setText(ConsoleFTP.getText()+"Connessione in corso...");
550   ConsoleFTP.setSelectionStart(ConsoleFTP.getText().length());
551   String host=HOST.getText();
552   String user=USER.getText();
553   String key=KEY.getText();
554   String FTPdir=DIR.getText();
555   int portFTP=21;
556   try {
557       portFTP=((Integer)PORTFTP.getValue());
558   }catch(Exception e) {
559   PORTFTP.setValue(Integer.valueOf(21));
562   }final int p=portFTP;
563   new Thread(new Runnable() {
564      public void run() {
565         //apro un thread per non bloccare il messaggio iniziale
566         Dumper dumper=new Dumper(host, user, key, p);
567         boolean check=Dumper.CheckFTP(host, user, key, p);
568         String[] file=dumper.GetRemoteFiles("/"+FTPdir);
569         //ottengo i nomi di tutti i file contenuti della cartella
570         StringTokenizer tokenizer;
572         if(check) {
```

```
573             ConsoleFTP.append("\n>> Connessione riuscita\n");
575             csNaLOG.append(">> Lettura files in: "+FTPdir+"\n");
578         }
579         else {
580             ConsoleFTP.setText(ConsoleFTP.getText()+"fallita");
581             csNaLOG.append("\n>> Connessione fallita.\n");
582             DownloadDump.setEnabled(true);
583         }
584         ConsoleFTP.setSelectionStart(ConsoleFTP.getText().length());
585         //ora cerco i file voluti
586         int cont=0;
587         for (String s:file) {
588             cont++;
589             if(cont>=file.length)
590             {
591                 DownloadDump.setEnabled(true);
592             }
593             tokenizer=new StringTokenizer(s, "-");
594             String type=tokenizer.nextToken();
595             //inizializzo variabili per il confronto della data con
596             //l'intervallo del filtro
597             DateFormat df = new SimpleDateFormat ("yyyy_MM_dd.HH_mm");
598             df.setLenient (false);
599             Date inizio=(Date)Data_INIZIO.getValue();
600             df.format(inizio);
601             Date fine=(Date)Data_FINE.getValue();
602             df.format(fine);
603             // Ora posso iniziare a distinguere se file filtrato
604             if(type.equals("NOfilter")) {
605                 tokenizer.nextToken();
606                 String fileDate="";
607                 // ricavo la data del dump e cancello estensione
608                 fileDate=tokenizer.nextToken();
610                 fileDate=fileDate.substring(0,(fileDate.length()-4));
612                 Date d=new Date();
613                 try {
614                    d = df.parse (fileDate);
615                 }catch(Exception e) {
616                 csNaLOG.append("\nErrore nella lettura ");
617                 ConsoleFTP.setText("Errore lettura intervallo");
618                 DownloadDump.setEnabled(true);
620              }
622          boolean condizione=d.compareTo(inizio)>=0&&d.compareTo(fine)<=0;
623              if(condizione) {
624                 csNaLOG.append("\n >> File trovato:"+s+"\n");
626                 File local=new File("Files_FTP/"+s);
627                 // se file esiste non lo scrivo
628                 if (local.exists()) {
629                  csNaLOG.append("Esistente, download non necessario!");
630              ConsoleFTP.append(s+">> file esistente in Files_FTP\n\n");
631                  csNaLOG.setSelectionStart(csNaLOG.getText().length());
632              ConsoleFTP.setSelectionStart(ConsoleFTP.getText().length());
633                 //altrimenti avviso del corretto salvataggio
634                 }else if(dumper.saveLocalFile(FTPdir, s, "Files_FTP/"+s))
                    {
635                   csNaLOG.append(s+" -> Salvato in locale. ");
636                   ConsoleFTP.append(s+"\n>> File salvato\n\n");
```

```
637                     csNaLOG.setSelectionStart(csNaLOG.getText().length());
638                   ConsoleFTP.setSelectionStart(csNaLOG.getText().length());
639                   }
640                }
642                //se file splittato
643                }else if(type.equals("Port")){
644                tokenizer.nextToken(); //porta del file splittato
645                tokenizer.nextToken();// IP
646                String fileDate="";
647                // ricavo la data del dump e cancello l'estensione .cap
648                fileDate=tokenizer.nextToken();
650                fileDate=fileDate.substring(0,(fileDate.length()-4));
652                Date d=new Date();
653                try {
654                   d = df.parse (fileDate);
655                }catch(Exception e) {
656                   csNaLOG.append("Errore nella lettura dell'intervallo");
657                }
659            boolean condizione=d.compareTo(inizio)>=0&&d.compareTo(fine)<=0;
664                if(condizione) {
665                csNaLOG.append("\n >> File trovato:"+s+"\n");
666                File local=new File("Files_FTP/"+s);
667                // se file esiste non lo scrivo
668                if (local.exists()) {
669                csNaLOG.append("File esistente,download non necessario!\n");
670                ConsoleFTP.append(s+"\n>> file esistente in Files_FTP\n\n");
671                csNaLOG.setSelectionStart(csNaLOG.getText().length());
672                ConsoleFTP.setSelectionStart(ConsoleFTP.getText().length());
673                }else if(dumper.saveLocalFile(FTPdir, s, "Files_FTP/"+s))
                   {
674                     csNaLOG.append(s+"-> Salvato in locale.\n");
675                     ConsoleFTP.append(s+">> File salvato\n\n");
676                     csNaLOG.setSelectionStart(csNaLOG.getText().length());
677                ConsoleFTP.setSelectionStart(ConsoleFTP.getText().length());
678                   }
679                 }
681              }
682           }
683           csNaLOG.append("\n\n>> FINE, DMAD pronto:");
684           ConsoleFTP.append("\n\n>> TERMINATO!");
685           csNaLOG.setSelectionStart(csNaLOG.getText().length());
686           ConsoleFTP.setSelectionStart(ConsoleFTP.getText().length());
687           DownloadDump.setEnabled(true);
688         }
689       }).start();
691     }
692  });
```

Codice 6.2.4 - la GUI: la funzione DownloadFiles()

Nelle righe 548-562 vengono letti i valori per l'accesso al server FTP, specificati negli appositi campi della GUI.

Dopodichè, nelle successive righe fino alle 585 avviene la connessione al server: a seconda del valore booleano restituito dalla funzione check() definita nella classe

Dumper (descritta nel capitolo 6.4), il codice successivo si comporterà in maniera differente, segnalando un errore nel caso di connessione non riuscita, o iniziando il controllo dei file presenti in caso di connessione instaurata correttamente. Alla riga 568 vengono letti i nomi dei file contenuti nella cartella, e salvati in un array di String, utilizzato nel successivo controllo atto a verificare se il singolo file deve essere scaricato o meno.

Alle righe 589-592, si può notare la riabilitazione del pulsante JButton che rappresenta la funzione "Scarica file". Questo meccanismo è stato introdotto per evitare che l'utente potesse avviare nuovamente il comando prima del completamento.

Quindi, grazie all'aiuto della nota classe StringTokenizer prevista dalle libreria standard del JDK utilizzato, inizia un ciclo *for* che scansiona tutte le stringhe precedentemente salvate nell'array rappresentante i file contenuti nella cartella DIR sul server FTP selezionato. Viene sfruttata la struttura del nome dei file creati da CSD (in particolare la presenza del carattere "-" utilizzato per ottenere il tipo di dump, l'ip della VulnBox, e la data di inizio dello scan): per ogni file, dopo aver raccolto queste informazioni viene effettuato un confronto con i parametri dell'intervallo specificato dall'utente, e se il file corrisponde, oltre a non essere già presente nella cartella locale, inizia il download di tale file.

Il download dei file selezionati è effettuato con la chiamata al metodo SaveLocalFile(), presente nella classe Dumper(). Tale metodo sarà approfondito nel capitolo 6.4.

6.2.5) GUI: la funzione MergeNofilter()

Questa funzionalità è stata inserita per permettere l'esportazione di un unico file senza filtri applicati (contenente quindi l'intero traffico di rete registrato da CSD), a partire dai file precedentemente scaricati (o caricati manualmente; per appronfondire questa funzionalità consultare il capitolo 6.2.8, selezionando quelli che corrispondono al periodo specificato dall'utente. Tali file vengono aggiunti a un unico file .cap, che rappresenta un unione dei pacchetti catturati nell'intero periodo, e originariamente salvati in diversi file .cap. Questa operazione è resa possbile grazie alla classe CapMerger, o più precisamente grazie al metodo statico MergeToOut(). Per appronfondire questa funzionalità, consultare il capitolo 6.5.

```
1008  JButton MergeExports = new JButton("Esporta merge non filtrato");
1009  MergeExports.setBounds(422, 234, 268, 33);
1010  MergeExports.addActionListener(new ActionListener() {
1011      public void actionPerformed(ActionEvent arg0) {
1012          String sourcefolder;
1014          if(FTPON_OFF.isSelected()) {//se modulo FTP attivo->Files_FTP
1015              sourcefolder="Files_FTP";
1016              csNaLOG.append("\n\n>> Lettura cartella Files_Utente: \n");
1017          }else {//altrimenti uso i file caricati dall'utente
1018              sourcefolder="Files_Utente";
1019              csNaLOG.append("\n\n>> Lettura cartella Files_FTP: \n");
1020          }
1021          File cartellaSrc=new File(sourcefolder);
1022          String []filesSorgenti=cartellaSrc.list();
1023          for(String s:filesSorgenti) {
1024              csNaLOG.append("\n - "+s);
1025          }
1026          if(filesSorgenti.length==0) {
1027              csNaLOG.append(" >> Nessun file nella cartella!\n");
1028              csNaLOG.append("EXPORT TERMINATO.\n");
1029              ConsoleFTP.append("\n\n>> EXPORT TERMINATO\n");
1030              csNaLOG.setSelectionStart(csNaLOG.getText().length());
1031             ConsoleFTP.setSelectionStart(ConsoleFTP.getText().length());
1032          }
1035          String exportName="Export/Export_NOfilter_"+
                ExportUserPath.getText()+".cap";
1036          exportMSG.setText("Salva come: \\Export\\Export_NOfilter_");
1037          boolean firstappend=false;
            //usato per evitare l'append del global header la prima volta
1038          for (String s:filesSorgenti) {
1039
1040              StringTokenizer tokenizer=new StringTokenizer(s, "-");
1041              String type=tokenizer.nextToken();
1042              //inizializzo variabili per il confronto della data
1044              DateFormat df = new SimpleDateFormat ("yyyy_MM_dd.HH_mm");
1045              df.setLenient (false);
1046              Date inizio=(Date)Data_INIZIO.getValue();
1047              df.format(inizio);
1048              Date fine=(Date)Data_FINE.getValue();
1049              df.format(fine);
1050              // Ora posso distinguere file filtrato o non filtrato
1051              if(type.equals("NOfilter")) {
1052                  tokenizer.nextToken();
1053                  String fileDate="";
1054                  // ricavo la data del dump e cancello l'estensione .cap
1055                  fileDate=tokenizer.nextToken();
1057                  fileDate=fileDate.substring(0,(fileDate.length()-4));
1059                  Date d=new Date();
1060                  try {
1061                      d = df.parse (fileDate);
1062                  }catch(Exception e) {
1063                      csNaLOG.append("Errore lettura intervallo");
1064                  }
1068          boolean condizione=d.compareTo(inizio)>=0&&d.compareTo(fine)<=0;
1069                  if(condizione) {
1070                      csNaLOG.append(s+" aggiunto a "+exportName);
1071      CapMerger.MergeToOUT((sourcefolder+"/"+s), exportName, firstappend);
```

```
1072                    if(firstappend==false) {
1073                        firstappend=true;
1074                    }
1075                }
1076                if(!condizione) {
1077                    csNaLOG.append(s+" non soddisfa i requisiti");
1078                 csNaLOG.setSelectionStart(csNaLOG.getText().length());
1079                }
1080            }}
1081        csNaLOG.append(" EXPORT TERMINATO.");
1082        ConsoleFTP.append("\n\n>> EXPORT TERMINATO\n");
1083        }
1086    });
```

Codice 6.2.5 - la GUI: la funzione MergeNoFilter()

Le prime righe di codice (1012-1022), servono per selezionare la cartella sorgente in cui sono presenti i file .cap creati da CSD scaricati, o caricati manualmente.

Quindi, inizia la scansione dei file contenuti nella cartella sorgente individuata, e viene effettuato un primo controllo sull'eventualità che tale cartella non contenga nessun file, in tal caso viene mostrato un messaggio di avviso all'utente e il comando termina.

Nel caso in cui siano invece stati rilevati dei file, viene effettuato un controllo per ognuno di essi: se il file è un dump di tipo "NOfilter", e tale file corrisponde a un dump creato da CSD nel periodo di interesse, allora viene aggiunto al file *Export* nella cartella /Export (vedere le righedi codice 1051-1080).

Alla riga 1037 si può notare l'inizializzazione della variabile *boolean firstAppend=false*, tale variabile è utilizzata per riconoscere se si tratta di un dump esistente a cui vanno aggiunti pacchetti (e in tal caso omette i primi 24 byte rappresentanti il global header del file .cap, per non corrompere il file *Export*), o di un dump non esistente (e che va quindi creato copiando tutti i byte del .cap sorgente). Questo aspetto verrà approfondito nel capitolo 6.5 relativo alla classe CapMerger().

6.2.6) GUI: la funzione FilterAndMerge()

```
775         JButton btnFiltracartellaFtp = new JButton("FILTRA E UNISCI");
776         btnFiltracartellaFtp.addActionListener(new ActionListener() {
777             public void actionPerformed(ActionEvent arg0) {
778                 String sourcefolder;
779           Vector <PcapPacket> packetBlacklist = new Vector<PcapPacket>();
781                 //se modulo FTP attivo uso files_FTP
782                 if(FTPON_OFF.isSelected()) {
783                     sourcefolder="Files_FTP";
784                     csNaLOG.append("Lettura cartella Files_Utente:\n\n");
785                 }else {//altrimenti uso i file caricati dall'utente
786                     sourcefolder="Files_Utente";
```

```
787                         csNaLOG.append("Lettura cartella Files_FTP:\n\n");
788                     }
789                     File cartellaSrc=new File(sourcefolder);
790                     String []filesSorgenti=cartellaSrc.list();
791                     for(String s:filesSorgenti) {
792                         csNaLOG.append(" - "+s+"\n");
793                     }
794                     if(filesSorgenti.length==0) {
795                         csNaLOG.append(" >> Nessun file nella cartella!\n");
796                         csNaLOG.append("EXPORT TERMINATO.\n\n");
797                         ConsoleFTP.append("\n>> EXPORT TERMINATO\n");
800                     }
801                     btnFiltracartellaFtp.setEnabled(false);
802                     new Thread(new Runnable() {
803                         public void run() {
804                      //apro un thread per non bloccare il messaggio iniziale
805                             int PORTAscan=0;
807                             try {
808                                 PORTAscan=((Integer)portFilter.getValue());
809                             }catch(Exception e)
                                { btnFiltracartellaFtp.setEnabled(true);
                                }
810                          boolean nessunFile=true;
811                          String exportName2="Export/Export_Filtered_"+
                             ExportUserPath.getText()+".cap";
812                          String exportName="Export/Export_Pt_"+
                             PORTAscan+"_"+ExportUserPath.getText()+".cap";
813                          exportMSG.setText("Salva come:"+PORTAscan);
814                          boolean firstappend=false;
                 //usato per evitare l'append del global header la prima volta
815                          CaptureCycle cicloSplit;
816                          for (String s:filesSorgenti) {
818                        StringTokenizer tokenizer=new StringTokenizer(s, "-");
819                          String type=tokenizer.nextToken();
820                          //inizializzo variabili per il confronto della data
822                    DateFormat df = new SimpleDateFormat ("yyyy_MM_dd.HH_mm");
823                         df.setLenient (false);
824                         Date inizio=(Date)Data_INIZIO.getValue();
825                         df.format(inizio);
826                         Date fine=(Date)Data_FINE.getValue();
827                         df.format(fine);
828                         StringBuilder errbuf = new StringBuilder();
```

Codice 6.2.6 - la GUI: la funzione FilterAndMerge()

Il codice 6.2.6 rappresenta la parte iniziale della funzione *Filter and Merge()*. Nella fase iniziale vengono inizializzate le variabili utilizzate nell'*ActionListener()*, tra cui il vettore destinato a contenere i pacchetti (in caso di Flow Control tcp attivo), e la variabile *sourceFolder*, inizializzata con il valore della cartella contenente i file sorgenti da filtrare. Tale cartella è diversa a seconda che il modulo FTP sia selezionato o meno, come è possibile osservare nelle righe 778-789.

Quindi, inizia la definizione di un thread contenente il resto del metodo (per non bloccare il rendering dell'output iniziale sulla GUI), e dopo aver ottenuto un elenco dei file contenuti nella cartella *sourceFolder,* inizia un ciclo *for* che si occupa di scansionare i file presenti nell'elenco.

Nelle righe 816-828 si trovano le prime istruzione di tale scansione, finalizzate a definire il tipo di file (a seconda del nome e dell'estensione rilevata) e altri parametri utili per la successiva fase di filtro dei pacchetti contenuti nei file trovati.

```
830      // Ora posso  distinguere se file filtrato per porta o non filtrato
831
if(type.equals("NOfilter")&&(DEEP_SCAN_FLAG.isSelected()||!PORT_YN.isSelected
())) {
832        String IP=tokenizer.nextToken();
833        String fileDate="";
834        // ricavo la data del dump e cancello l'estensione .cap
835        fileDate=tokenizer.nextToken();
837        fileDate=fileDate.substring(0,(fileDate.length()-4));
839        Date d=new Date();
840        try {
841            d = df.parse (fileDate);
842        }catch(Exception e) {
843        btnFiltracartellaFtp.setEnabled(true);
844       csNaLOG.append("Errore nella lettura :( \n");
845       ConsoleFTP.append("\n>> Errore lettura intervallo :( \n");
848       }
850       boolean condizione=d.compareTo(inizio)>=0&&d.compareTo(fine)<=0;
851       if(condizione) {
853           cicloSplit=new CaptureCycle(errbuf, sourcefolder, "Temp", IP,
FLOW_CONTR_FLAG.isSelected(), EXPR_FLAG.isSelected(), EXPR_FILTER.getText());
855           if(PORT_YN.isSelected()) {
856       String splitResult=cicloSplit.Split(s, PORTAscan, packetBlacklist);
857       CapMerger.MergeToOUT(splitResult, exportName, firstappend);
858           }
859           else {
861               String splitResult=cicloSplit.Filter(s, packetBlacklist);
862              CapMerger.MergeToOUT(splitResult, exportName2, firstappend);
864           }
866           nessunFile=false;
867           csNaLOG.append(s+"soddisfa i requisiti del filtro :D");
868           ConsoleFTP.append("\n>> file salvato");
871           if(firstappend==false) {
872               firstappend=true;
873           }
874        }
875        if(!condizione) {
876           csNaLOG.append(s+" non soddisfa i requisiti del filtro :(\n");
877        }
878     }//end if(equals("NOfilter"))
879     //L'else riguarda il caso file splittato per porta(senza deep scan)
882     else
if(type.equals("Port")&&!DEEP_SCAN_FLAG.isSelected()&&PORT_YN.isSelected()){
```

```
883         String port=tokenizer.nextToken(); //porta del file splittato
884         csNaLOG.append("\n -> "+s+": rilevato DUMP splittato su: "+port);
885         String IP=tokenizer.nextToken();// IP
886         String fileDate="";
887         // ricavo la data del dump e cancello l'estensione .cap
888         fileDate=tokenizer.nextToken();
890         fileDate=fileDate.substring(0,(fileDate.length()-4));
892         Date d=new Date();
893         try {
894             d = df.parse (fileDate);
895         }catch(Exception e) {
896             btnFiltracartellaFtp.setEnabled(true);
901         }
903         boolean condizione=d.compareTo(inizio)>=0&&d.compareTo(fine)<=0;
904         int userFilter=0;
905         try {
906             userFilter=((Integer)portFilter.getValue());
907         }catch(Exception e) {btnFiltracartellaFtp.setEnabled(true);}
908         if(condizione&&userFilter==Integer.parseInt(port)) {
911             cicloSplit=new CaptureCycle(errbuf, sourcefolder, "Temp", IP,
FLOW_CONTR_FLAG.isSelected(), EXPR_FLAG.isSelected(), EXPR_FILTER.getText());
912         String splitResult=cicloSplit.Split(s, PORTAscan, packetBlacklist);
913             CapMerger.MergeToOUT(splitResult, exportName, firstappend);
914             nessunFile=false;
915             csNaLOG.append("\n>> Il file soddisfa i requisiti del filtro");
916             ConsoleFTP.append("\n>> file salvato");
919             if(firstappend==false) {
920                     firstappend=true;
921          }  }
925     }}
```

Codice 6.2.7 - la GUI: la funzione FilterAndMerge()

Il codice 6.2.7 rappresenta la fase di riconoscimento e filtro dei file rilevati e positivi al controllo effettuato. Alle righe 831 e 882 si possono notare le due dichiarazioni di inizio delle strutture condizionali *if* che controllano se il file deve essere riconosciuto un file non filtrato come sorgente, oppure un file risultante dal filtro di CSD. Quindi a seconda che il valore di porta sia specificato o meno (campo PORT_YN, che rappresenta il relativo selettore presente nella GUI), viene eseguito il metodo Filter() oppure il metodo Split(): il primo si occupa di eseguire il filtro nello stesso modo di quello presente in CSD, mentre il secondo si occupa di effettuare l'operazione di filtro senza tenere conto dei valori contenuti in *tcp.dst* e *tcp.src*.

Quindi, nel caso in cui il tipo di file venga riconosciuto, e i valori della data superino i parametri del filtro (oltre a eventuali parametri aggiuntivi, come ad esempio un'espressione regolare di filtro per la payload tcp o l'impostazione *Flow_control* attiva), il file viene creato (prima iterazione), o aggiunto (iterazioni successive) al file di

export, con la chiamata al metodo statico *CapMerger.MergeToOut()* (righe 857/862/913).

```
931      csNaLOG.append("inizio analisi Export ");
932      ConsoleFTP.append("Inizio visualizzazione");
935      //ora visualizzo il file creato
936      if(nessunFile) { // se nessun file corrisponde al filtro
937         csNaLOG.append("Nessun file corrispondente al filtro :(");
939         btnFiltracartellaFtp.setEnabled(true);
942      }
943      else { // se invece sono state trovate corrispondenze
944          StringBuilder buff=new StringBuilder();
945          if(!PORT_YN.isSelected()) {
946               btnFiltracartellaFtp.setEnabled(true);
947               JOptionPane.showMessageDialog(frmCsnaBy, (new
          File(exportName2)).getName()+": Completato - "+ (new
          File(exportName2)).length()+" Byte di dati");
948               return;
949               }
950               final Pcap reader = Pcap.openOffline(exportName, buff);
951               if (reader == null) {
952                   csNaLOG.append("Errore nella lettura dell'Export");
954               }
956               reader.loop(-1, new JPacketHandler<StringBuilder>() {
957               //inizializzo il tipo dei pacchetti di riferimento che user
958               //riconoscere se salvare il pacchetto in nextpacket()
959                 final Tcp tcp = new Tcp();
960                 final Udp udp = new Udp();
961                 final Ip4 ip = new Ip4();
962                 int i=0;
963              public void nextPacket(JPacket packet, StringBuilder errbuf)
964                 {
965                    i++;
966                    if(packet.hasHeader(tcp)&&packet.hasHeader(ip)) {
967                        packet.getHeader(tcp);
968                        packet.getHeader(ip);
973               ConsoleFTP.append("\n-> Pacchetto(TCP) numero: #"+ i);
987               //altre informazioni, output sulla console omesso
988            }else if(packet.hasHeader(udp)) {
989               packet.getHeader(udp);
990               packet.getHeader(ip);
991               ConsoleFTP.append("\n-> Pacchetto(UDP) numero: #"+ i);
           }
1003         }//end nextpacket()
1004      }, buff);//end loop scansione file
1005     reader.close(); //chiusura pcap
1006     if(firstappend) {
1007         File export=new File(exportName);
1008         if (export.length()<1024) {//byte
1009            JOptionPane.showMessageDialog(frmCsnaBy, "Completato, il file"
          +export.getName()+" ha "+export.length()+" Byte di dati");
1010  }else if(export.length()>=1024 && export.length()<(1024*1024)) {//kbyte
1011            JOptionPane.showMessageDialog(frmCsnaBy, "Completato, il file"
          +export.getName()+" ha "+export.length()/1024+" kByte di dati");
1012         }else {//MByte
1013            JOptionPane.showMessageDialog(frmCsnaBy, "Completato, il file"
```

```
      +export.getName()+" ha "+export.length()/(1024*1024)+" MByte di dati");
1014           }
1015         }
1017         ConsoleFTP.append("FILTRA! terminato ");
1020         btnFiltracartellaFtp.setEnabled(true);
1022       }}
1024     }).start();
1027 }});
```

Codice 6.2.8 - la GUI: la funzione FilterAndMerge()

Il codice 6.2.8 fa uso di classi e metodi presenti nella libreria JnetPcap, già visti nella definizione delle classi appartenenti a CSD (capitolo 5), al fine di visualizzare i pacchetti contenuti nel file di export risultante dalla precedente fase di filtraggio dei file sorgenti.

Inoltre, come si può notare alle righe 945-949, è stato scelto di non fornire in output i dati di interesse dei pacchetti, nel caso in cui non sia specificata una precisa porta: in questo caso il filtro non effettua controlli sul valore di porta tcp/udp, ed essendo quindi generati molti più pacchetti, la fase di analisi risulterebbe molto rallentata dalla stampa dei valori nella console. In tal caso, viene semplicemente mostrata una nuova finestra contenente l'avviso che il metodo è terminato, oltre ad alcune informazioni sul file creato (dimensione, nome). Tale finestra di dialogo è stata implementata mediante la classe *JOptionPane.showMessageDialog(),* facente parte della libreria *Swing*.

6.2.7) GUI: visualizzazione ed eliminazione dei file locali

Queste due funzionalità sono state implementate in maniera molto simile tra di loro, per questo motivo verrà presentato solo il codice relativo alla funzione di visualizzazione dei file.

```
1284   JButton btnVediFile = new JButton("Visualizza File");
1285   btnVediFile.addActionListener(new ActionListener() {
1286       public void actionPerformed(ActionEvent arg0) {
1288           File sourcefolder1=new File("Files_FTP");
1289           File sourcefolder2=new File("Files_Utente");
1290           File sourcefolder3=new File("Export");
1292
if(FTP_DELETE_FLAG.isSelected()||EXPORT_DELETE_FLAG.isSelected()||USER_DELETE
_FLAG.isSelected()) {
1293       new Thread(new Runnable() {
1294          public void run() {
1296             try {
1297               if(FTP_DELETE_FLAG.isSelected()) {
1298                  File []list1=sourcefolder1.listFiles();
```

```
1299                ConsoleFTP.append(sourcefolder1.getName());
1301                if(list1.length==0) {
1302                   csNaLOG.append("Nessun file");
1306                }else {
1307                   ConsoleFTP.append("Lettura directory:
                       "+sourcefolder1.getName()+"\n");
1308                   for(File f:list1) {
1309                      if(f.isDirectory()) {
1310                         csNaLOG.append("\n >> Directory:");
1311                      }
1312                      else if(f.isFile()) {
1313                         String fname=f.getName();
1314                         String extension=fname.substring
                             (fname.length()-4,fname.length());
1315                         if(!extension.equals(".cap")) {
1316                           csNaLOG.append("File generico: "+f.getName());
1317                         }else {
1318                           csNaLOG.append("File .cap");
1319                         }
1321                      }
1324                 }
1325              }
1326           }
```

Codice 6.2.9 - la GUI: visualizzazione/eliminazione dei file

Come si può notare dalle prime righe del codice 6.2.9, prima di tutto viene effettuato un controllo sui *flag* corrispondenti ai selettori del filtro, per verificare se almeno una cartella è stata selezionata. Quindi, inizia una serie di 3 strutture condizionali *if* che hanno lo scopo di visualizzare i file per la relativa cartella corrispondente (Files_FTP,Files_Utente, o Export).

Alle righe 1297-1326 è possibile osservare la prima di tali strutture condizionali, costituita dal controllo preventivo sull'esistenza di file nella cartella(riga 1301), e dal successivo ciclo iterativo *for,* che si occupa di scansionare i file rilevati visualizzandone il contenuto (righe 1308-1321).

Le operazioni eseguite per le altre cartelle sono analoghe a quelle descritte, e il codice relativo ad esse verrà quindi omesso.

Per quanto riguarda la funzione di eliminazione dei file, la struttura del codice è molto simile, con l'unica differenza che invece di visualizzare i file contenuti, viene effettuata la loro cancellazione con il comando *fileObject.delete().*

6.2.8) GUI: il selettore FTPON/OFF

Questo capitolo non rappresenta una vera e propria funzionalità, quanto più un aspetto cruciale relativo alla praticità e versatilità dell'utilizzo del software DMAD.

Il selettore *FTPON/OFF* è un componente immerso nella GUI (consultare figura 6.1), e grazie a questo elemento è stato possibile implementare in maniera intuitiva lo switch tra la modalità "Online" e la modalità "Offline":

- Nella modalità "Online" (FTPON) è possibile visualizzare e scaricare i file presenti sul server FTP. In questa modalità le funzioni di filtro e merge dei dump non filtrati utilizzano i dati contenuti nella cartella Files_FTP.
- Nella modalità "Offline" (FTPOFF) le funzioni di filtro e merge dei dump non filtrati utilizzano i dati contenuti nella cartella Files_Utente.

A seguire il codice relativo all'actionListener() associato al selettore:

```
729         JCheckBox FTPON_OFF = new JCheckBox("(ON)");
730         FTPON_OFF.setForeground(SystemColor.activeCaptionText);
731         FTPON_OFF.setBounds(558, 184, 77, 38);
732         FTPON_OFF.setFont(new Font("Tahoma", Font.BOLD, 16));
733         FTPON_OFF.addActionListener(new ActionListener() {
734             public void actionPerformed(ActionEvent arg0) {
735                 if(FTPON_OFF.isSelected()) { //accendi
736                     ConsoleFTP.append("\n Modulo FTPS attivato");
737                     FTPON_OFF.setText("(ON)");
738                     lblFtpsLog.setForeground(Color.black);
739                     USER.setEnabled(true);
740                     PORTFTP.setEnabled(true);
741                     HOST.setEnabled(true);
742                     KEY.setEnabled(true);
743                     lblFtpsLog.setText("MODULO FTPS");
744                     DIR.setEnabled(true);
745                     Check.setEnabled(true);
746                     DownloadDump.setEnabled(true);
747                     FTPonoff.setForeground(new Color(240, 255, 255));
748                     FTPonoff.setText("Modalit�tps attiva");
749                     msg.setText("Cartella sorgente: \\Files_FTP");
750 
751                 }else { //spegni
752                     ConsoleFTP.append("Modulo FTPS disattivato\n");
753                     lblFtpsLog.setForeground(Color.gray);
754                     FTPON_OFF.setText("(OFF)");
755                     USER.setEnabled(false);
756                     lblFtpsLog.setText("MODULO FTPS");
757                     PORTFTP.setEnabled(false);
758                     HOST.setEnabled(false);
759                     KEY.setEnabled(false);
760                     DIR.setEnabled(false);
761                     Check.setEnabled(false);
```

```
762                         DownloadDump.setEnabled(false);
763                         FTPonoff.setForeground(Color.blue);
764                         FTPonoff.setText("Modalit�tps disattivata");
765                         msg.setText("Cartella sorgente: \\Files_Utente");
766 
767                     }
768                 }
769             });
770             FTPON_OFF.setSelected(true);
```

Codice 6.2.10 - la GUI: il selettore FTPON/OFF

6.3) DMAD: la classe CaptureCycle()

Il codice della classe CaptureCycle, utilizzato per le funzioni di filtro ed esportazione dei dump, è molto simile a quello presente in CSD. Per questo motivo, nella stesura di del codice di questa classe, si è scelto di riutilizzare il sorgente dell'applicazione CSD, con qualche piccola modifica:

- In DMAD, a differenza di CSD, è possibile specificare i parametri del filtro in maniera indipendente: non ci sono limitazioni di combinazioni eseguibili tra i parametri forniti (ad esempio, in DMAD è possibile eseguire un filtro dei pacchetti contenenti espressioni regolari, con l'opzione Flow_control attiva o disattiva a seconda della volontà dell'utente).
- In DMAD è previsto un metodo aggiuntivo Filter(), molto simile al metodo Split(), ma invece di applicare il filtro sui pacchetti in transito su una porta, vengono considerati tutti i pacchetti rilevati come input del filtro (in CSD è invece possibile applicare le opzioni del filtro solo sulle porte selezionate, o in alternativa non applicare affatto il filtro utilizzando i dump "NOfilter").

Il codice di questa classe è omesso per i motivi sopra citati.

6.4) DMAD: la classe Dumper()

Il codice 6.3.1 mostra la dichiarazione di campi e costruttori della classe Dumper(), la classe creata con lo scopo di stabilire una connessione al server, visualizzare i file presenti sul server, e scaricarli nella cartella locale Files_FTP.

Tale codice è analogo a quello mostrato nel caso della funzione Dumper() già approfondita nell'analisi dell'applicazione CSD.

```
12 public class Dumper {
13
14  // campi
15
16  public String hostFTP;
17  public String usr;
18  public String psw;
19  public int Port;
21  public FTPSClient FTPS;
22  public String protocol = "TLS";
23
24  //costruttori
25
26  public Dumper(String host,String user, String password,int PORT) {
27      this.hostFTP=host;
28      this.usr=user;
29      this.psw=password;
30      this.Port=PORT;
31  }
```

Codice 6.4.1 - la classe Dumper()

Per quanto riguarda la sicurezza della connessione, è stato scelto di utilizzare il successore del protocollo SSL, ovvero TLS.

A differenza della classe Dumper() presente nell'applicazione CSD, questa volta sono stati definiti 3 metodi:

- boolean CheckFTP() : questo metodo statico si occupa di controllare la connessione al server FTP, tramite protocollo TLS. Come nel metodo CheckFTP visto precedentemente, viene restituito un valore booleano corrispondente all'esito della connessione. Il codice di questo metodo è omesso in quanto identico a quello già approfondito nel capitolo 5.4.1.

- boolean saveLocalFile() : questo metodo è stato creato per implementare la funzionalità di download dei file presenti in una cartella presente sul server FTP, il cui nome viene passato come parametro di tipo String. Gli altri parametri letti dalla funzione saveLocalFile() sono il nome del file da scaricare e il percorso in cui salvarlo in locale, in formato String.

```
58  // metodo per il salvataggio da server su cartella locale
59  public boolean saveLocalFile(String FTPdir,String RemoteName, String
localOutputPath) {
61      FTPS=new FTPSClient(protocol);
62         try {
63             FTPS.connect(this.hostFTP, this.Port);
64             FTPS.login(this.usr, this.psw);
65             FTPS.enterLocalPassiveMode();
66             FTPS.setFileType(FTP.BINARY_FILE_TYPE);
```

```
67              //testa esistenza cartella specificata in CONFIG.ini)
68              FTPS.changeWorkingDirectory(FTPdir);
69              int returnCodeA = FTPS.getReplyCode();
70              if (returnCodeA == 550) {//se la directory specificata
71               return false;
72              }
73              File LocalFile = new File(localOutputPath);
74              OutputStream outputStream = new FileOutputStream(LocalFile);
76              boolean done = FTPS.retrieveFile(RemoteName, outputStream);
77              outputStream.close();
78              return done;
80          } catch (IOException ex) {
81               return false;
82          } finally {
83              try {
84                  if (FTPS.isConnected()) {
85                      FTPS.logout();
86                      FTPS.disconnect();
87                  }
88              } catch (IOException ex) {
90          }
91      }
93  }
```

Codice 6.4.2 - la classe Dumper() - il metodo saveLocalFile()

- String[] GetRemoteFiles() : questo metodo si occupa di scansionare tutti i file presenti all'interno di una cartella (presente sul server FTP) specificata come parametro del metodo, e, dopo averli salvati in un'array di tipo String, li restituisce sotto tale forma al chiamante del metodo.

```
94  //metodo che ritorna un array contenente i file rilevati nella directory
95  public String[] GetRemoteFiles(String FTPdir){
96      String [] alldumps;
97      FTPS=new FTPSClient(protocol);
98         try {
99              FTPS.connect(this.hostFTP, this.Port);
100             FTPS.login(this.usr, this.psw);
101             FTPS.enterLocalPassiveMode();
102             FTPS.setFileType(FTP.BINARY_FILE_TYPE);
103             //testa esistenza cartella specificata in CONFIG.ini)
104             FTPS.changeWorkingDirectory(FTPdir);
105             int returnCodeA = FTPS.getReplyCode();
106             if (returnCodeA == 550) {//se la directory non esiste
107              return null;
108             }
109             alldumps = FTPS.listNames(FTPdir);
111             return alldumps;
112         } catch (IOException ex) {
113              return null;
114         } finally {
115             try {
116                 if (FTPS.isConnected()) {
117                     FTPS.logout();
118                     FTPS.disconnect();
```

```
119                 }
120             } catch (IOException ex) {
121                 return null;
122     }}
```

Codice 6.4.3- la classe Dumper() - il metodo GetRemoteFiles()

6.5) DMAD: la classe CapMerger()

Questa classe, già anticipata nel capitolo 6.2, si occupa di unire i file sorgenti prodotti da CSD in un unico file, raggruppando tutti i pacchetti contenuti nei vari file di origine selezionati. Per fare questo, è stata implementata una funzionalità di copia dei file, con l'aggiunta di un parametro *boolean* append:

- Quando *append==false* il metodo lavora in sovrascrittura, scrivendo tutti i byte del file sorgente a partire dal primo.
- Quando *append==true* il metodo lavora in scrittura sequenziale, scrivendo tutti i byte del file sorgente a partire dal numero 25 (i primi 24 Byte rappresentano il global header del file .cap, e sono quindi bypassati per evitare la corruzione del file, e la conseguente perdita di leggibilità delle informazioni in esso contenute). Per ulteriori informazioni è possibile consultare il sito web di wireshark al seguente URL: *https://wiki.wireshark.org/Development/LibpcapFileFormat*

```
12  public CapMerger() {   //Costruttore nullo    }
22  public static void MergeToOUT(String inputPath1, String OutputPath,
boolean append) {
24      File in1=new File(inputPath1);
25      File out=new File(OutputPath);
26      FileOutputStream fOut;
27      FileInputStream fin1;
29      try {
30          fin1 = new FileInputStream(in1);
31          if (append) {
32              fin1.getChannel().position(globheader);
33          }//scarta i primi 24 byte(header)
35          fOut = new FileOutputStream(out,append);
36          FileChannel sourceChannel1 = fin1.getChannel();
37          FileChannel destChannel = fOut.getChannel();
38          destChannel.transferFrom(sourceChannel1, destChannel.size(),
         sourceChannel1.size());
39          sourceChannel1.close();
40          destChannel.close();
41          fin1.close();
42          fOut.close();
43      }catch(Exception e) {   }
```

Codice 6.5.1 - la classe CapMerger()

7) Introduzione all'uso

Dopo aver introdotto e presentato nel dettaglio le due applicazioni oggetto di questo documento di tesi, nel seguente capitolo verranno approfondite le principali modalità di utilizzo dei due software, accompagnate da esempi pratici e immagini tratte dai test effettuati.

7.1) Introduzione all'uso - CSD

- **Installazione prerequisiti e avvio CSD:**
    - installare WinPcap/libPcap e jnetpcap: a seconda del proprio sistema operativo, per installare jnetpcap seguire le istruzioni contenute nel file RELEASE_NOTES.txt, presente nella cartella principale di JnetPcap.
    - Su sistemi windows, copiare la cartella contenente CSD sul disco locale, aprirla ed eseguire il file launcher.bat.
    - Su Unix/Linux, dopo aver installato libPcap-Jnetpcap, è sufficiente copiare la cartella CSD su disco e avviare il jar da terminale
    - Per evitare problemi con l'utilizzo della memoria da parte di CSD, utilizzare il comando -Xmx1024m (serve per riservare 1 GB di ram a CSD) durante l'avvio da terminale. In caso di necessità variare il parametro a seconda del proprio caso.

      Digitare: VulnBox >> java -Xmx1024m -jar CSD-v1.3b.jar

In caso di problemi, eseguire installer.bat ed eventualmente verificare che jnetpcap sia installato correttamente (batch funzionante solo su sistemi windows). Se il problema persiste, verificare che non sia presente un firewall o un'impostazione di sicurezza che blocchi la comunicazione con la porta del server FTP, o l'accesso al dispositivo di rete.

- **Impostazione del file CONFIG.ini:**
    - CSD è predisposto per reperire i parametri dello scan e della connessione FTP, leggendo i parametri impostati nel file CONFIG. In ogni sezione sono presenti dei parametri relativi alla funzione svolta. (per esempio HostFTP= FTP.server.org).

Modificando il valore a destra di "=" è possibile personalizzare il funzionamento di CSD a seconda delle proprie preferenze.

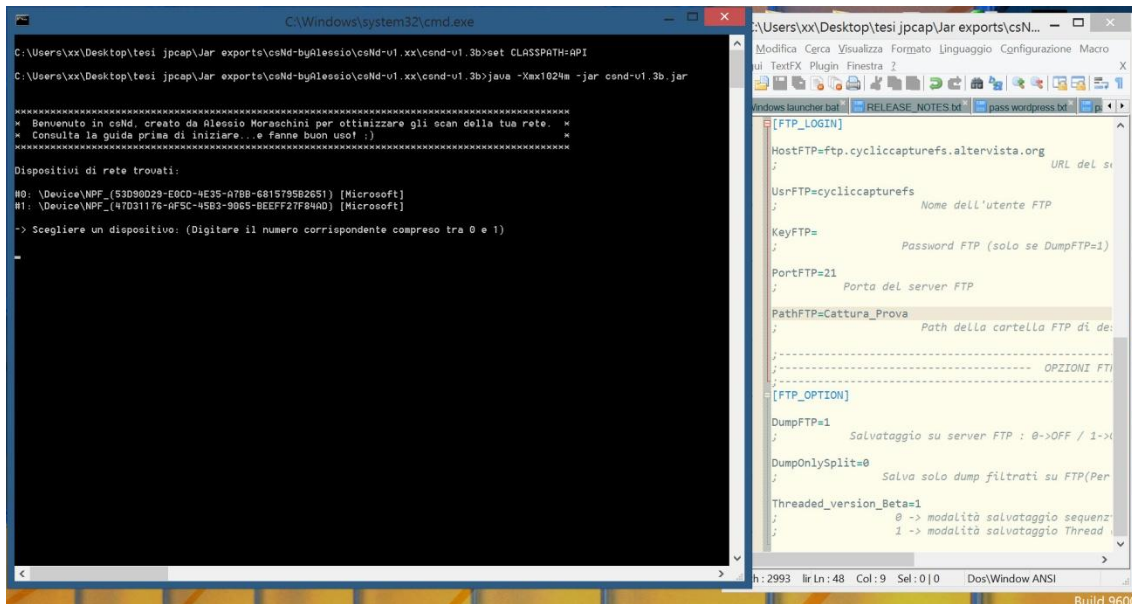

Figura 7.1.1 - Schermata di richiesta selezione device di cattura (Sullo sfondo il file CONFIG.ini)

- **Avvio :**

  - Avviare il software seguendo la modalità indicata precedentemente, a seconda del proprio sistema operativo.
  - Se il file CONFIG.ini è impostato correttamente e non vengono riscontrati errori, il software chiederà di scegliere un dispositivo, altrimenti verrà segnalato l'errore critico.

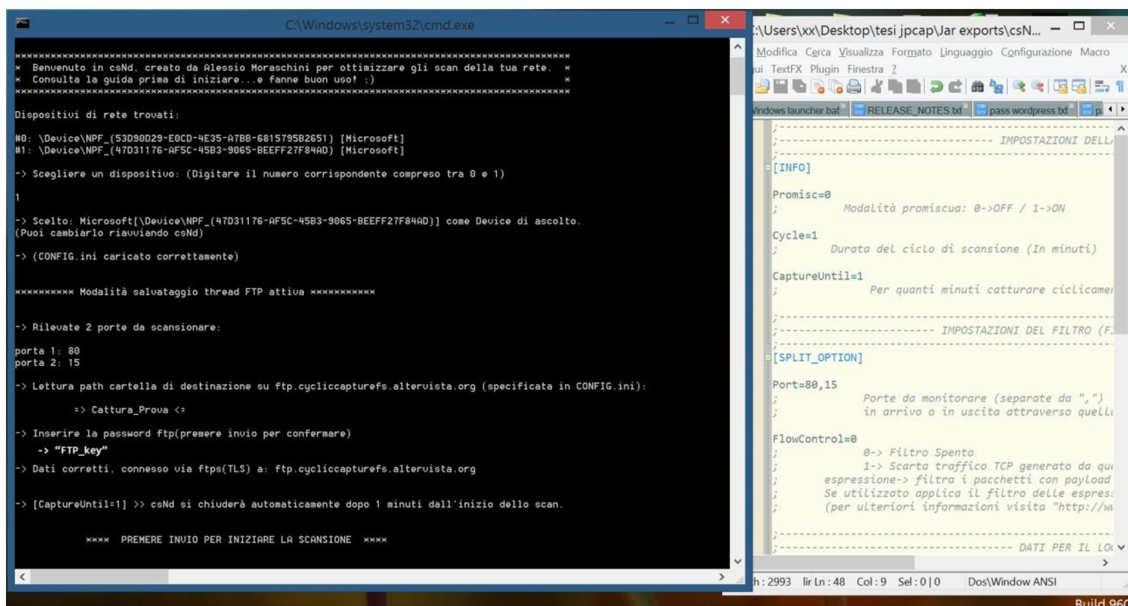

Figura 7.1.2 - Inserimento KeyFTP mancante e completamento inizializzazione.

- In caso di campo FTPkey vuoto, la password verrà richiesta durante l'esecuzione di CSD (solo se il salvataggio FTP è abilitato), altrimenti verrà segnalata la connessione avvenuta.
- Se tutto è andato correttamente ed è stata selezionata la corretta scheda di rete (assicurarsi di selezionare quella in uso durante le analisi!), CSD termina il caricamento delle opzioni di scan, split, e upload, restando in attesa di conferma per l'avvio della scansione: premendo il Tasto invio, si inizieranno a vedere i messaggi relativi alla cattura del singolo pacchetto e del tempo trascorso/residuo (Per il ciclo di scansione corrente).

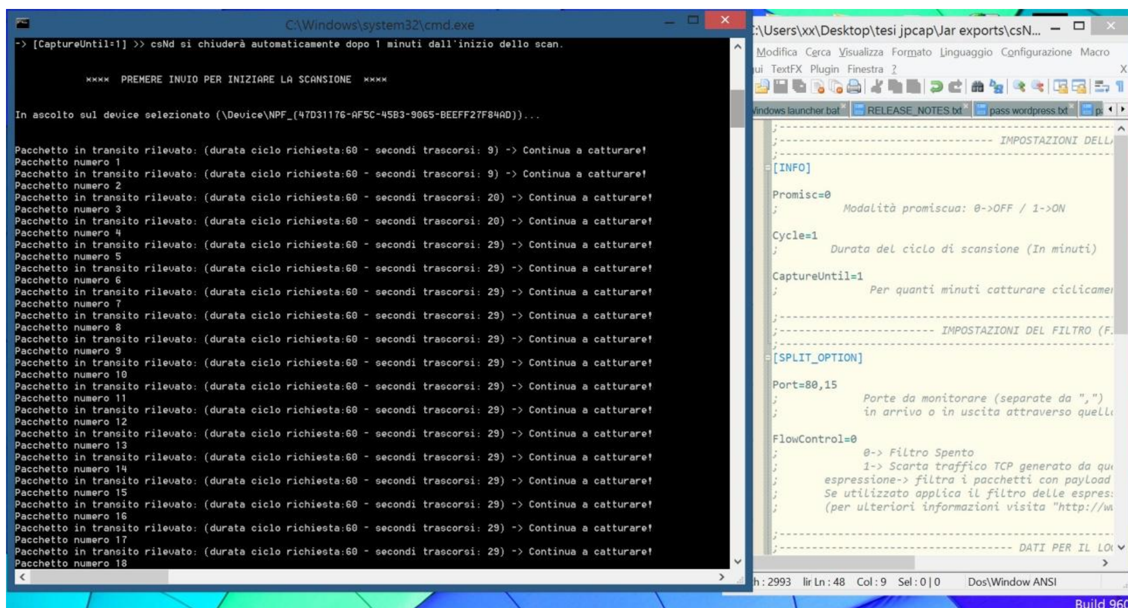

Figura 7.1.3 - Ciclo di cattura avviato

- Al termine del ciclo di scansione, CSD effettua una copia locale di dump non filtrato (NOfilter-....cap) nella cartella /Dump/Local_capture. Quindi, inizia il salvataggio su server FTP (con FTPS) del dump non filtrato. A questo punto, a seconda della modalità (threaded_beta/sequenziale) specificata nel file CONFIG, CSD resta in attesa del caricamento (sequenziale), oppure inizia a caricare il file usando un thread parallelo, permettendo a CSD di proseguire con lo split del dump non filtrato mentre l'upload viene completato.

- Nella fase di Split il dump presente in Local_capture viene analizzato (vengono controllate le opzioni del filtro specificate in CONFIG.ini), e per ogni porta specificata, viene creato e copiato in /Dump/Local_capture un Dump fitrato per quella porta con le impostazioni correnti.

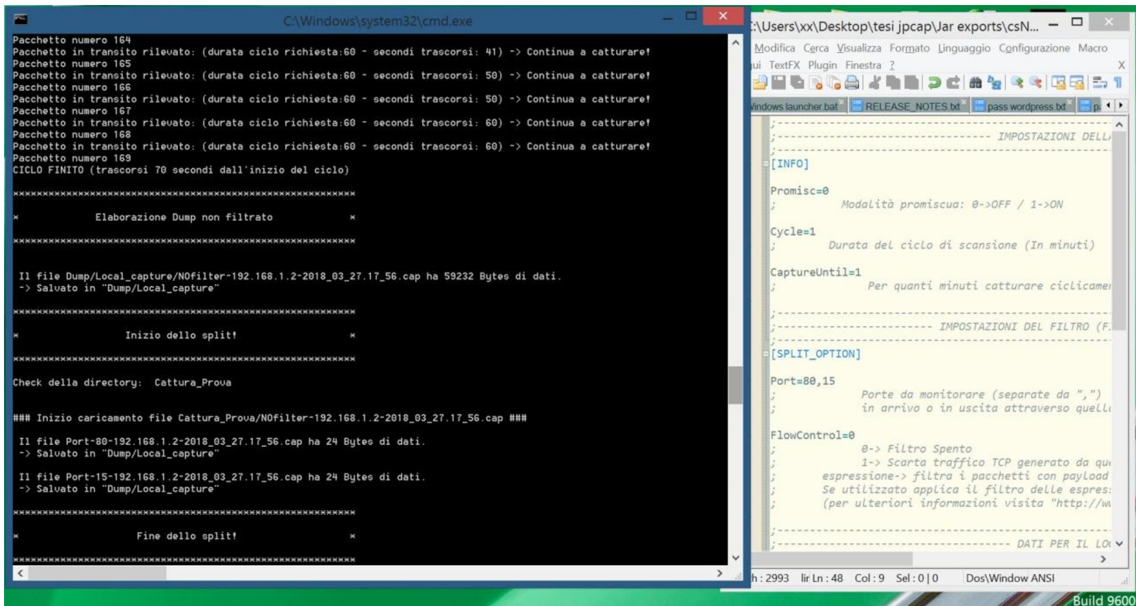

Figura 7.1.4 - Termine ciclo - elaborazione file "NOfilter - elaborazione file filtrato (fase di split)

- Quindi, il ciclo viene ripetuto fino al termine dei *minutiDiCattura* specificati in CONFIG.ini, caricando ogni volta i relativi dump nella cartella folderFTPath specificata nel file CONFIG (se *folderFTPath* non esiste a quel path di destinazione, verrà creata da CSD durante l'esecuzione).

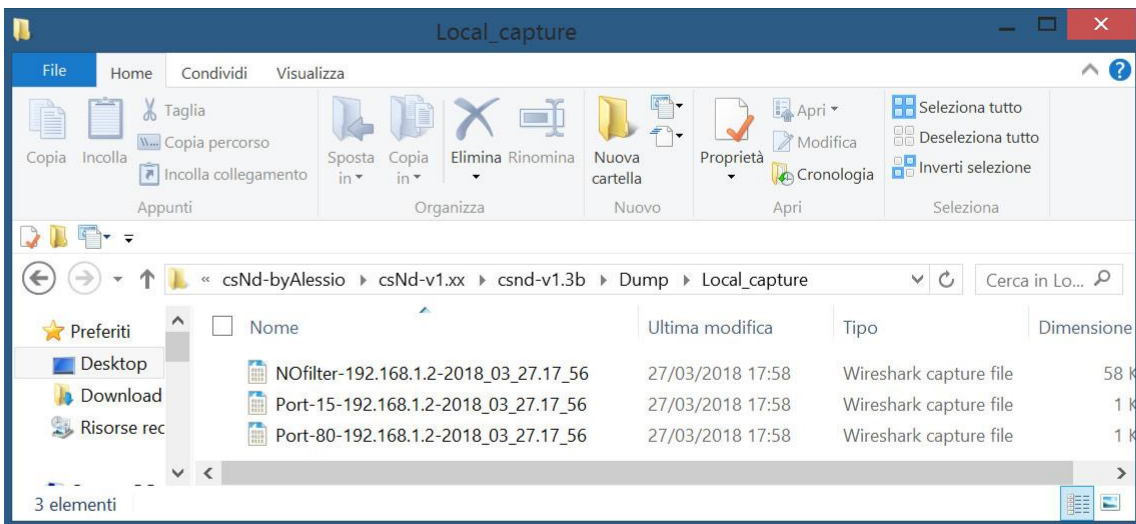

Figura 7.1.5 - I File locali di backup salvati nella cartella Local_capture

Sito remoto: /Cattura_Prova

- /
  - Cattura_Prova
  - VulnBoxCaptureP2

Nome file	Dimension...	Tipo file	Ultima modifica	Perme...	Propri...
..					
Port-80-192.168.1.2-2018_03_27.17_56.cap	24	Wireshark capt...	27/03/2018 18:01:04	0664	33917...
Port-15-192.168.1.2-2018_03_27.17_56.cap	24	Wireshark capt...	27/03/2018 18:01:04	0664	33917...
NOfilter-192.168.1.2-2018_03_27.17_56.cap	59.232	Wireshark capt...	27/03/2018 18:01:03	0664	33917...

3 file - dimensione totale: 59.280 byte

Figura 7.1.6 - I file caricati su server (Visualizzati con FileZilla Client)

NB - E' fortemente sconsigliato cambiare il nome dei file creati da CSD, in quanto tali nomi sono utilizzati dall'aplicazione DMAD per il riconoscimento del periodo, IP VulnBox, e porta relativi al dump. Cambiando il nome del file DMAD potrebbe non funzionare correttamente o non funzionare affatto.

Per queste ragioni, si consiglia di limitare le operazioni sui file creati al semplice spostamento -nel caso di caricamento manuale dei file- da Local_Capture all'apposita cartella nella directory sorgente di DMAD (Questa funzionalità verrà approfondita nel capitolo 7.2), o eliminazione nel caso in cui i files non risultino più necessari.

C:\Windows\system32\cmd.exe

```
***********************************************************
*                     Fine dello split!                    *
***********************************************************

*** Attesa completamento salvataggio FTP ***...
-> Thread residui in elaborazione: 3
*** Thread residui in elaborazione: 3 .
*** Thread residui in elaborazione: 3 . .
 -> File caricato correttamente!

### Inizio caricamento file Cattura_Prova/Port-80-192.168.1.2-2018_03_27.17_56.cap ###

### Inizio caricamento file Cattura_Prova/Port-15-192.168.1.2-2018_03_27.17_56.cap ###
*** Thread residui in elaborazione: 3 . . .
 -> File caricato correttamente!
 -> File caricato correttamente!
0 Thread in lavorazione...

:) :) :) :) :) :) :) :) :) :) :) :) :) :) :) :) :) :) :) :) :)

-> csNd terminato!

--------- GRAZIE PER AVER USATO QUESTO SOFTWARE ;) --------
```

Figura 7.1.7 - Attesa completamento upload e terminazione CSD

7.2) Introduzione all'uso - DMAD

In questo capitolo verrano approfonditi gli aspetti relativi alla configurazione ed utilizzo del software di analisi dei file di cattura generati da CSD, ovvero l'applicazione DMAD.

Per quanto riguarda la fase di installazione, la fase di installazione di LibPcap/WinPcap e JnetPcap è identica a quella presentata nel capitolo 7.1, e verrà quindi omessa.

Una volta installate le librerie necessarie, dovrebbe essere sufficiente un doppio click sul file DMAD.jar per avviare il software. In caso ciò non bastasse, è possibile avviare il software direttamente da terminale (dopo aver controllato la corretta installazione di LibPcap e Jnetpcap, e la loro presenza nel PATH di sistema) con il seguente comando:

- PC-Analisi >> java -Xmx1024m -jar DMAD.jar

Come anticipato precedentemente, si tratta di un'applicazione *standalone* progettata per il download, analisi, unione e filtraggio dei file .cap catturati sulla VulnBox, tramite l'interfaccia grafica presentata in figura 7.2.1.

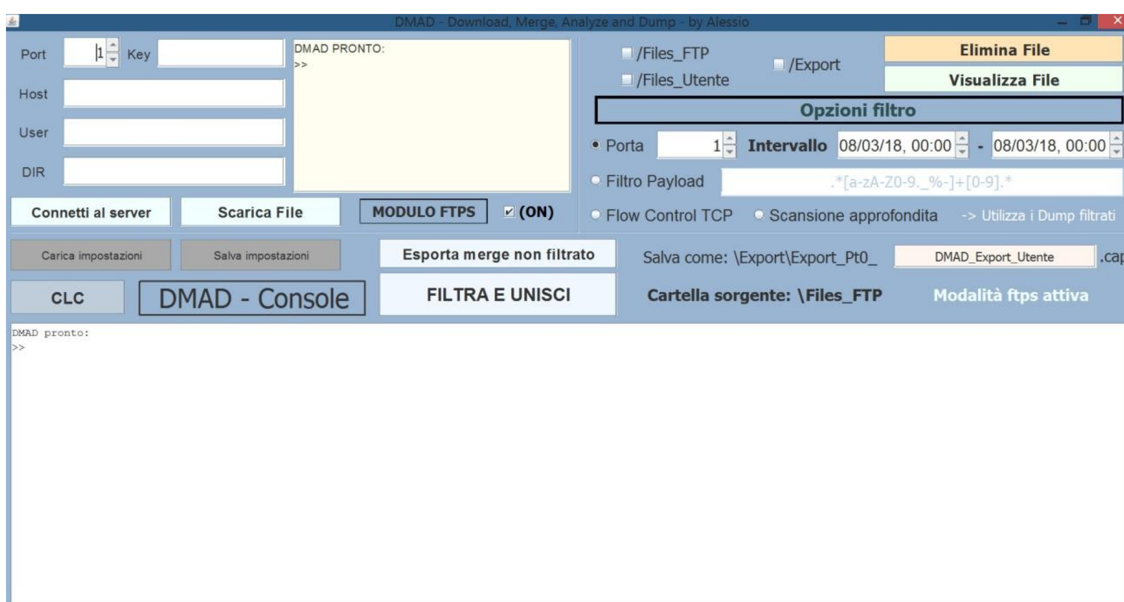

Figura 7.2.1 - DMAD - Schermata iniziale

Ecco una breve introduzione ai comandi e campi visibili una volta avviato il software:

- Nella parte sinistra in alto, è possibile trovare i campi in cui immettere i valori relativi al server FTP in cui risiedono i file salvati dall'applicazione CSD. Inoltre, in questa sezione è possibile trovare le funzioni necessarie alla

connessione al server e al download dei file in esso contenuti (alla directory specificata nel campo DIR). Il selettore "MODULO FTPS" (ON/OFF) serve per specificare la cartella sorgente nel caso in cui vengano chiamate le funzioni "Esporta merge non filtrato" ed "Filtra ed unisci". Sotto il modulo FTP, è possibile trovare le funzioni di caricamento e salvataggio delle impostazioni di DMAD, oltre al tasto CLC, utile per cancellare il contenuto delle aree di testo per la visualizzazione dei messaggi di interazione con l'utente.

- Nella metà inferiore della schermata, si trova la finestra principale in cui avvengono le comunicazione del software all'utente: qui verranno visualizzati i risultati delle operazioni effettuate, i pacchetti rilevati, e le varie informazioni relative al comando in esecuzione.
- Nella parte destra in alto si trovano le funzioni di visualizzazione e cancellazione dei file locali dell'applicazione. Sotto di esse si trovano i parametri del filtro: qui è possibile specificare la porta (una in particolare o tutte le porte) tcp/udp, le date iniziale e finale, le opzioni di controllo espressioni regolari, Flow_control e scansione approfondita (utilizza i file NOfilter in fase di filtro, permettendo di filtrare traffico su porte non specificate in fase di cattura, e per cui non esiste quindi uno specifico file "Port-n-ip-date.cap").
- Nella parte centrale è possibile trovare la finestra di LOG in cui vengono comunicate le informazioni chiave relative alle operazioni effettuate, oltre ad eventuali messaggi di errore. Subito sotto di essa si trovano i tasti collegati alle funzioni di merge e filtro dei file sorgenti scaricati (o caricati manualmente in /Files_utente).

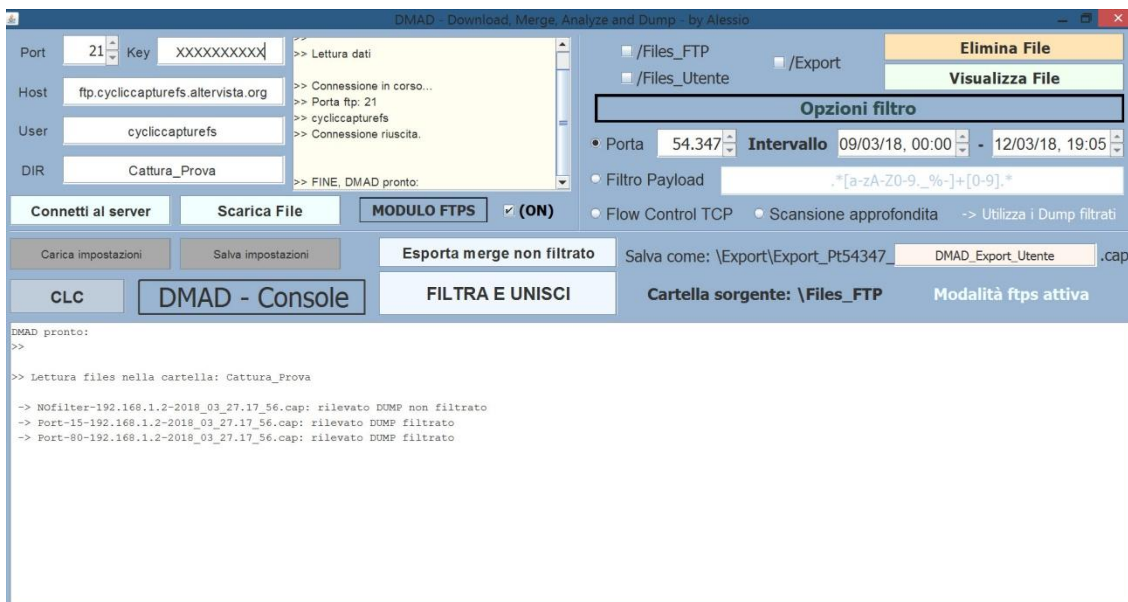

Figura 7.2.2 - DMAD - impostazioni caricate e connessione FTP effettuata

Guida all'utilizzo del software:

1. Il primo passo da effettuare, nel caso in cui si sia già utilizzato il software e si vogliano caricare i settaggi salvati, che nel caso in cui si voglia salvare le impostazioni inserite per la prima volta, è utilizzare i tasti "Carica impostazioni" e "Salva impostazioni".

2. A questo punto, nel caso in cui si desideri scaricare i file dal server FTP (è richiesta l'immissione dei valori corretti nei campi relativi alla connessione FTPS), è possibile connettersi al server e visualizzare i file presenti all'interno della cartella "DIR" nel server FTP specificato nel campo "Host". E' possibile vedere un'esempio del risultato dell'esecuzione dei comandi descritti ai punti 1. e 2. nella figura 7.2.2. Si ricorda che prima di effettuare il download dei file è necessario selezionare il periodo di interesse, siccome DMAD scarica solo i file appartenenti al periodo specificato, al fine di ottimizzare al massimo l'utilizzo della rete e il tempo di risposta del programma.

3. Quindi, cliccare il tasto "Scarica File" per effettuare il download dei file corrispondenti al filtro, nel caso in cui essi non sia già presenti all'interno della cartella Files_FTP: in tal caso viene avvisato che il file è già presente, e la fase di download passa al file successivo (figura 7.2.3).

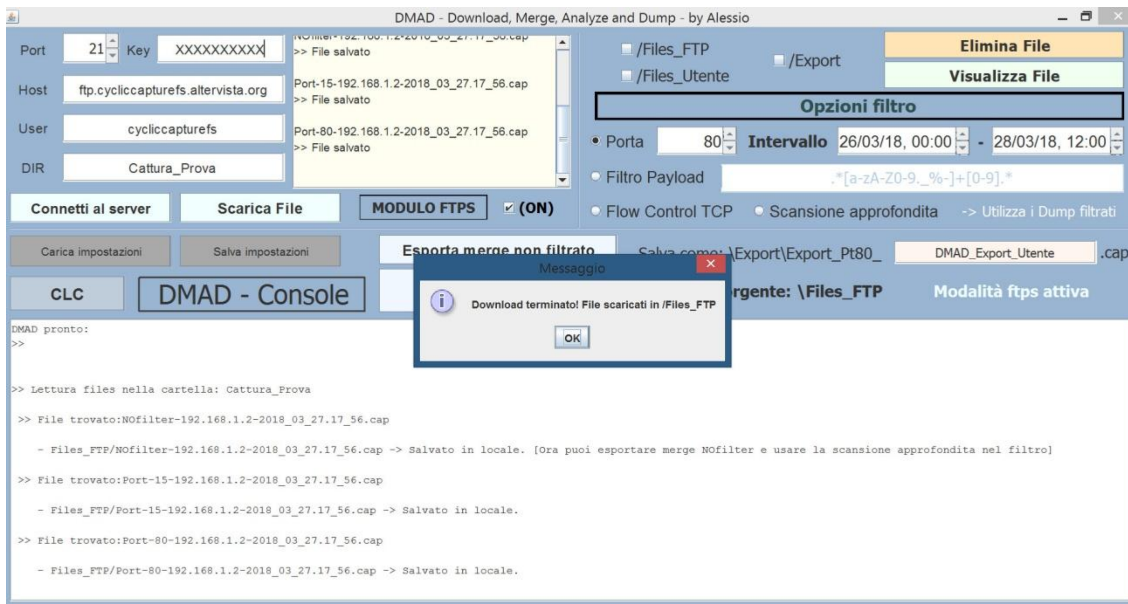

Figura 7.2.3 - DMAD - Filtro data settato e Download dei file terminato

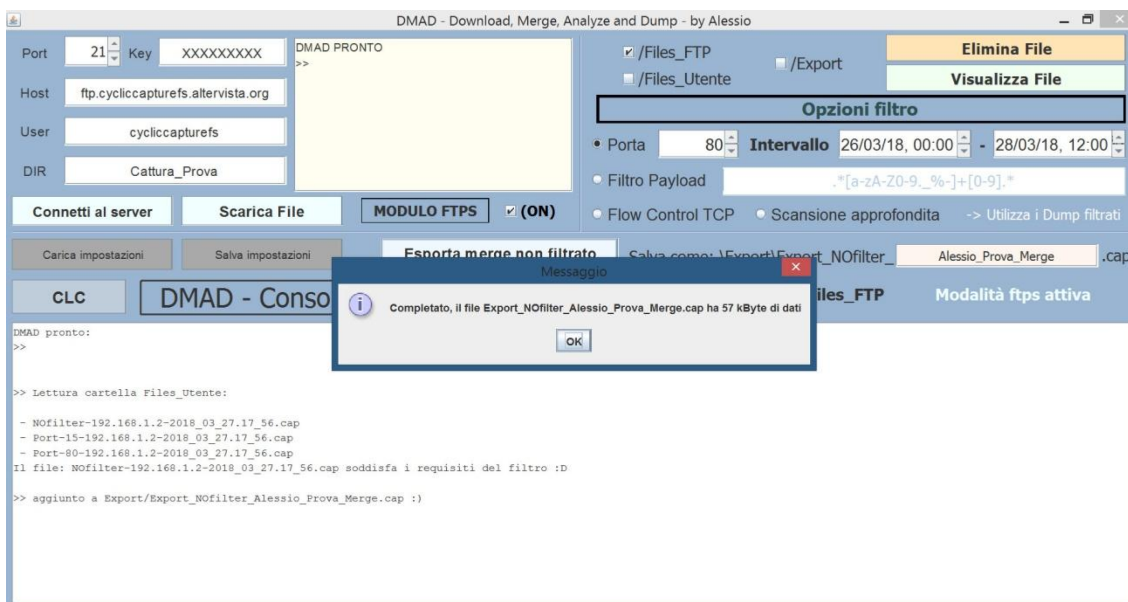

Figura 7.2.4 - DMAD - Merge dei file non filtrati

A questo punto i file scaricati (o analogamente copiati manuamente in /Files_utente, nel caso di caricamento manuale dei file. Ricordarsi in tal caso di deselezionare il Modulo FTPS, spuntando il relativo selettore presente al centro dell'interfaccia) sono pronti per essere processati da DMAD. Cliccando sul tasto "Esporta Merge non filtrato" è possibile creare un nuovo file (avente il nome specificato nella parte destra dell'interfaccia, modificabile dall'utente) contenente tutti i pacchetti rilevati all'interno dei file "NOfilter" corrispondenti al

periodo specificato. Il nome dei file che rappresentano tale tipo di export, inizia con "NOfilter", e contiene l'intero traffico catturato (Figura 7.2.4)

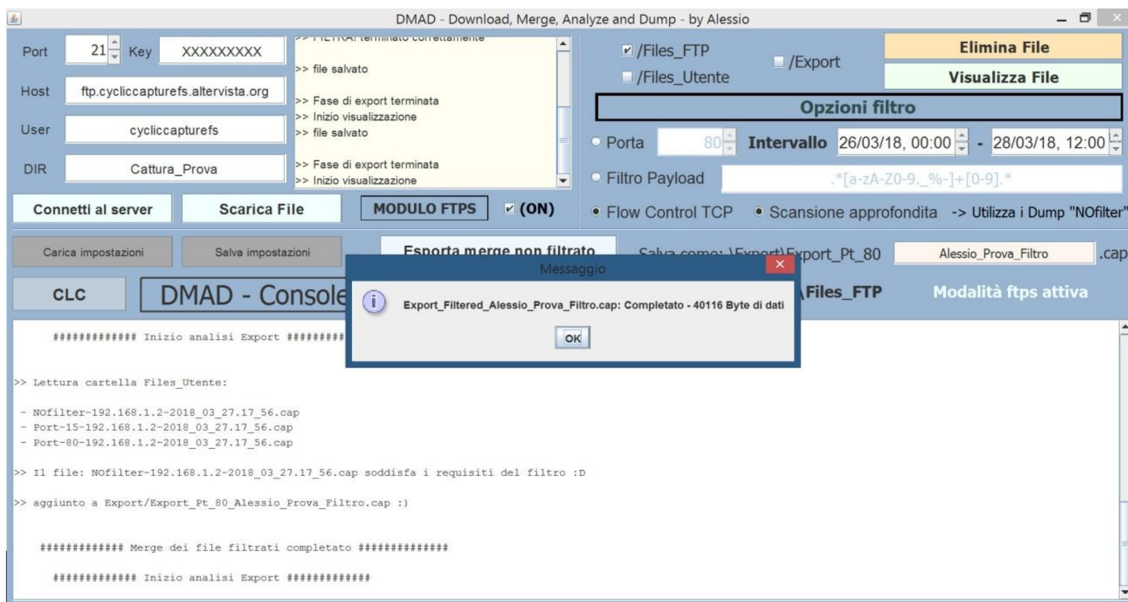

Figura 7.2.5 - DMAD - output del comando "Filtra e unisci"

4. Cliccando il tasto "Filtra e unisci", è possibile applicare i parametri del filtro (specificati sulla destra della schermata) ed esportare un'unico file contenente i pacchetti rilevati a partire dai file sorgenti e corrispondenti al filtro settato. E' possibile vedere i risultati di questa operazione in figura 7.2.5. Quindi, i file creati si possono trovare all'interno della cartella /Export, e sono compatibili con i principali software presenti sul mercato per la lettura di file di cattura con estensione .cap (es. wireshark). E' possibile in ogni momento visualizzare ed eventualmente eliminare i file presenti nelle cartelle locali di DMAD utilizzando le funzioni presenti in alto a destra, dopo aver spuntato le cartelle a cui applicare il comando (vedere figura 7.2.6).

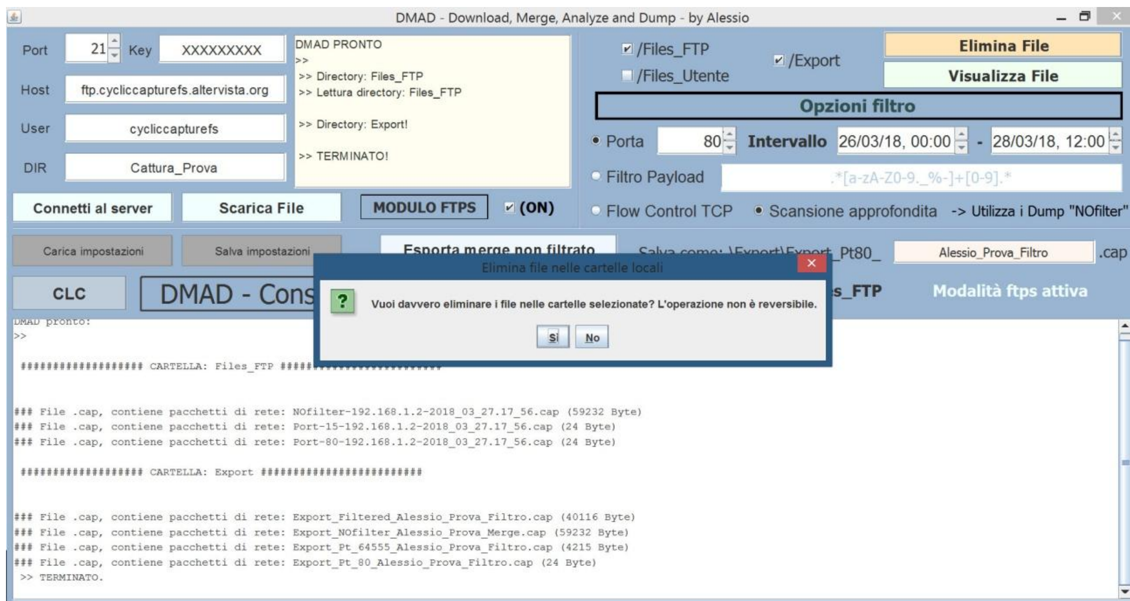

Figura 7.2.6 - DMAD - Visualizza file (sfondo) - richiesta di conferma (Elimina File)

8) Conclusioni e sviluppi futuri

Dopo aver testato le diverse modalità di utilizzo, e le principali combinazioni possibili per gli input inseriti, si può affermare che le applicazioni CSD e DMAD soddisfano gli obiettivi proposti, oltre ad aggiungere funzionalità extra per la personalizzazione sia della cattura che dell'analisi del traffico di rete.

Durante la finalizzazione del progetto e l'esecuzione dei test, sono state indivuate alcune funzionalità utili da implementare in eventuali sviluppi futuri dei due applicativi:

1 - CSD

- Sarebbe interessante, al fine di ottimizzare la *user experience* fornita dal software, aggiungere ulteriori possibilità di configurazione del filtro, tra cui la possibilità di specificare protocollo, campi dell'header, filtro payload esteso ad altri campi, ecc. In particolare, sarebbe interessante estendere la funzione Flow_control, per escludere il traffico generato dalla VulnBox anche nel caso di pacchetti di protocolli diversi dal tcp.
- Siccome è stata notata una certa macchinosità nell'utilizzo manuale del file CONFIG per il settaggio dello scan di rete, sarebbe utile prevedere l'apertura di una finestra (GUI) all'avvio di CSD, in cui settare tramite interfaccia grafica i vari parametri del software.
- In caso di Upload di file di grosse dimensioni, l'utente potrebbe nutrire alcune perplessità sulla percentuale di completamento dell'operazione. Per questo motivo sarebbe utile rendere il software più interattivo, inserendo codice (nel main o nella classe Dumper) per la visualizzazione periodica delle informazioni sugli upload in corso (nome file, byte caricati, byte residui, % completamento).

2 - DMAD

- Al fine di rendere più comprensibile l'interfaccia grafica, sarebbe utile aggiungere una barra orizzontale in cui inserire i vari menu a tendina in cui inserire i comandi eseguibili, ordinandoli in maniera più efficace (specialmente nel caso in cui vengano inserite nuove funzionalità).
- In analogia alla proposta effettuata per il software CSD, sarebbe utile implementare una funzionalità di aggiornamento sull'avanzamento del download dei file dal server FTP (e magari anche per il filtro/visualizzatore dei pacchetti).
- Siccome il salvataggio delle password all'interno del file di configurazione comprometterebbe la sicurezza del server stesso (e, potenzialmente, a tutte le macchine che si connettono o comunicano con tale server). DMAD, pur prevedendo il salvataggio delle password, scoraggia l'utente al salvataggio della password, e la resetta in caso di risposta negativa alla richiesta di conferma. Tuttavia, sarebbe utile implementare una funzionalità di crittografia delle password e dei dati contenuti all'interno del file di configurazione (preferibilmente con un sistema a doppia chiave, puubblica e privata).
- La GUI, e gli elementi in essa presenti, sono stati creati utilizzando un layout a coordinate assolute, ed è quindi statica (non ridimensionabile dall'utente). E' stato notato che ciò potrebbe causare problemi nell'utilizzabilità del software, in caso di monitor utilizzati aventi basse risoluzioni. Per questo si consiglia di ridefinire il layout della GUI utilizzando coordinate relative, o altri metodi che permettono di definire un layout di tipo *responsive,* capace di adattarsi alle necessità di visualizzazione di un maggior numero di utenti.
- Al fine di ampliare le potenzialità di analisi del software, sarebbe utile implementare delle funzionalità di *Intrusion Detection,* aggiungendo codice per l'analisi, decodifica e riconosciumento dei dati contenuti all'interno degli header dei protocolli rilevati, e confrontandoli con quelli tipici delle tipologie di attacco conosciute, creando e aggiornando una struttura dati contenente i valori associati a pacchetti malevoli.

- Al fine di ottimizzare l'efficacia del sistema di IDT proposto al punto precedente, sarebbe utile implementare una funzionalità di decodifica dei pacchetti criptati su rete Wlan, specificando la chiave WPA utilizzata dal sistema VulnBox. In alternativa, nel caso di traffico rilevato su macchine diverse dalla VulnBox, sarebbe necessario un metodo di cattura dei 4 handshake EAPOL utilizzati dai protocolli WPA/WPA2 per la determinazione della chiave di cifratura. In tal modo sarebbe possibile ampliare il numero e la tipologia di attacchi che DMAD potrebbe riconoscere, aumentando l'efficacia del sistema di analisi del traffico.
- Con l'utilizzo di file di cattura di grandi dimensioni, o periodi di analisi molto lunghi, l'esecuzione dei comandi di "Filtra e unisci" ed "Esporta dump non filtrati" potrebbe risultare rallentata: a tal fine risulterebbe utile un ottimizzazione dell'efficienza del codice, oltre a una separazione della fase di applicazione del filtro da quella di visualizzazione dei pacchetti contenuti, inserendole in due tasti separati.

Riferimenti bibliografici

[1] Html.it - Introduzione al protocollo FTP - 19 luglio 2006.
Http://www.html.it/articoli/introduzione-al-protocollo-FTP-1/

[2] Codejava.net - Upload and download file using FTP in Java. 13 agosto 2017.
http://codejava.net/java-se/networking/FTP/java-FTP-file-upload-tutorial-and-example
http://codejava.net/java-se/networking/FTP/java-FTP-file-download-tutorial-and-example

[3] Inetdaemon.com - TCP and 3-way-handshake working principles. 19 maggio 2018.
http://www.inetdaemon.com/tutorials/internet/tcp/3-way_handshake.shtml

[4] Wikipedia - UDP Protocol. 11 maggio 2018.
https://it.wikipedia.org/wiki/User_Datagram_Protocol

[5] Wireshark.org - ".cap" file header structure. 23 agosto 2015.
https://wiki.wireshark.org/Development/LibpcapFileFormat

[6] WinPcap - Open source library for Net scanning in Windows. 8 marzo 2013
https://www.winpcap.org/install

[7] JnetPcap - Java wrapper for libpcap library native calls. 2017.
http://jnetpcap.com

[8] Eclipse Oxygen - IDE for java software develoopment. 28 giugno 2018.
https://projects.eclipse.org/releases/oxygen

[9] Eclipse WindowBuilder - Tool for UI creation design. 10 febbraio 2011.
https://www.eclipse.org/windowbuilder/

[10] Ini4j - Java library for ".INI" file reading and writing. 11 marzo 2005.
http://ini4j.sourceforge.net/

[11] Html.it - espressioni regolari in Java. 22 settembre 2008.
http://www.html.it/articoli/espressioni-regolari-in-java-1/

[12] Altervista.org - free hosting and FTP server. 25 gennaio 2016.
https://it.altervista.org/

[13] Wikipedia - sicurezza informatica. 22 gennaio 2018.
https://it.wikipedia.org/wiki/Sicurezza_informatica