INDICE

INTRODUZIONE ED OBIETTIVO

I*NTRODUZIONE*: L'esercizio consiste nella realizzazione di un simulatore che invia **datagram UDP di 1 KB** verso un indirizzo e una porta specificati, con l'obiettivo di osservare il comportamento del traffico di rete e le evidenze prodotte sul sistema ricevente.
Lo script acquisisce i parametri necessari, costruisce payload di lunghezza definita con contenuto casuale e li trasmette tramite **socket UDP**; durante l'esecuzione vengono forniti messaggi di stato che permettono di correlare facilmente quanto mostrato a schermo con le tracce catturate da strumenti di monitoraggio (netcat o Wireshark).

OBIETTIVO: Lo scopo dell'attività è verificare e documentare il flusso di **datagram UDP** generati: si intende dimostrare come vengono creati i pacchetti, come transitano sulla rete locale e come appaiono agli strumenti di cattura consentendo di raccogliere evidenze utili per analisi di comportamento, approfondimenti diagnostici e valutazioni del carico di rete.

IMPORTAZIONE LIBRERIE

```
import socket          #  per la creazione e l'uso di socket (UDP)
import random          #  per generare dati casuali (payload 1KB)
import sys             #  per uscire in caso di errore

print("--- UDP-Flood: generazione e invio controllato di datagram da 1024 byte ---")
```

In questa prima sezione del codice vengono importate le **librerie** fondamentali necessarie al funzionamento dello script e viene mostrato un messaggio introduttivo che descrive l'operazione in corso.
Le librerie **socket, random e sys** costituiscono la base tecnica dell'intero programma:
Socket serve per la creazione e l'utilizzo dei socket UDP, indispensabili per l'invio dei pacchetti verso il target;
Random viene utilizzata per generare il contenuto casuale dei datagram, come richiesto dalla traccia, garantendo che ogni pacchetto abbia un payload differente di 1024 byte;
Sys consente invece di gestire eventuali errori o terminazioni controllate dello script, interrompendo in modo sicuro l'esecuzione in caso di input non validi.

Il messaggio stampato con il comando **print("--- UDP-Flood: generazione e invio controllato di datagram da 1024 byte ---")** ha la funzione di introdurre l'operazione e rendere chiaro all'utente cosa sta per accadere: il programma sta avviando la simulazione di invio controllato di pacchetti UDP da 1024 byte ciascuno.
Questo messaggio iniziale non ha una funzione tecnica ma serve a rendere l'esecuzione più leggibile e ordinata, facilitando anche la correlazione con i dati catturati durante il monitoraggio.

INPUT UTENTE

```python
# 1) INPUT UTENTE
try:

    target_ip = input("Inserisci l'IP target (es. 127.0.0.1): ").strip() # Chiede all'utente l'indirizzo IP target e rimuove spazi indesiderati
    target_port = int(input("Inserisci la porta target (es. 1234): ").strip()) # Chiede la porta target e converte in intero; può sollevare ValueE
    packet_count = int(input("Quanti pacchetti da 1KB vuoi inviare? ").strip()) # Chiede il numero di pacchetti da inviare e converte in intero

except ValueError:   # Se uno degli input che dovrebbe essere numerico non può essere convertito
    print("\n[ERRORE] Porta e numero pacchetti devono essere interi.") # Messaggio di errore chiaro per l'utente
    sys.exit(1)  # Termina il programma con codice di uscita 1 (errore)

# controllo minimi
if packet_count <= 0:  # Controlla che il numero di pacchetti sia maggiore di zero
    print("\n[ERRORE] Numero pacchetti deve essere > 0.")
    sys.exit(1) # Se non lo è termina il programma

if not (1 <= target_port <= 65535):  # Controlla che la porta sia nel range valido per TCP/UDP
    print("\n[ERRORE] Porta fuori range (1-65535).")
    sys.exit(1) # Se non valida, termina il programma
```

Questa sezione rappresenta la parte logica dedicata alla raccolta e alla validazione dei dati forniti dall'utente.
L'intero blocco è racchiuso all'interno della struttura **try–except**, scelta per potere gestire le eccezioni.
Il blocco **try** serve per racchiudere le istruzioni che potrebbero generare errori durante l'esecuzione: in questo caso, la conversione dei valori forniti dall'utente in numeri interi.
Quando l'utente digita un valore errato (ad esempio una lettera al posto di un numero per la porta o per il numero di pacchetti), il programma genera automaticamente un'eccezione di tipo ValueError; la presenza di questo blocco di intercettare quell'errore e di reagire in modo controllato invece di interrompersi bruscamente.
In questo caso, viene visualizzato un messaggio di errore chiaro e leggibile e il programma termina con il comando **sys.exit(1**), che chiude l'esecuzione restituendo un codice di errore (1) al sistema.
Questo approccio serve per mantenere la stabilità e la prevedibilità del programma: senza un controllo esplicito, un semplice input sbagliato farebbe interrompere lo script con un messaggio d'errore tecnico poco comprensibile.
Dopo il blocco *try–except* vengono eseguiti i **controlli minimi** ovvero verifiche logiche sui dati già convertiti correttamente in numeri.
Queste condizioni, implementate tramite le istruzioni **if** e **if not,** permettono di accertare che i valori rispettino i limiti imposti dal protocollo e dalle regole operative del programma.
La prima *condizione* **if packet_count <= 0**: verifica che il numero di pacchetti da inviare sia positivo: un valore pari a zero o negativo non avrebbe senso in un contesto di invio di datagram, per cui il programma mostra un messaggio d'errore e termina immediatamente.
La seconda *condizione* **if not (1 <= target_port <= 65535):** controlla che la porta inserita rientri nell'intervallo consentito dai protocolli TCP e UDP (da 1 a 65535), se il valore supera questi limiti o è pari a zero, anche in questo caso lo script blocca l'esecuzione ed esce con un messaggio d'errore.
In sintesi, questa parte del codice assicura che il programma lavori solo con dati coerenti e validi, prevenendo in anticipo eventuali malfunzionamenti durante l'invio dei pacchetti.

PREPARAZIONE PACCHETTO

```
# 2) PREPARAZIONE PACCHETTO
def build_packet_1kb(): # Definisce una funzione per generare un pacchetto di 1024 byte
    return bytes(random.getrandbits(8) for _ in range(1024)) # Crea una sequenza di 1024 byte; per ogni posizio
```

In questa sezione viene definita la **funzione build_packet_1kb()** avente il compito di generare il contenuto di un singolo pacchetto da 1024 byte.
La funzione utilizza il **modulo random** per produrre dati casuali, rispettando così la richiesta della traccia che prevede l'invio di pacchetti con contenuto variabile e non ripetitivo.
All'interno di **return** viene usata un'espressione compatta che combina una *list comprehension* (forma compatta di ciclo che permette di generare una sequenza in un'unica riga) con il **costruttore bytes()**.
La porzione random.getrandbits(8) genera un numero casuale di 8 bit (cioè un valore compreso tra 0 e 255), mentre la funzione **range(1024)** specifica che questa operazione deve essere ripetuta esattamente 1024 volte, in modo da costruire una sequenza di un kilobyte.
Il risultato finale è un blocco di dati binari che rappresenta il **payload** del pacchetto UDP, pronto per essere inviato alla destinazione.
Questa funzione garantisce che ogni pacchetto inviato sia diverso dal precedente, permettendo di simulare in modo più realistico un traffico di rete variabile e non statico.

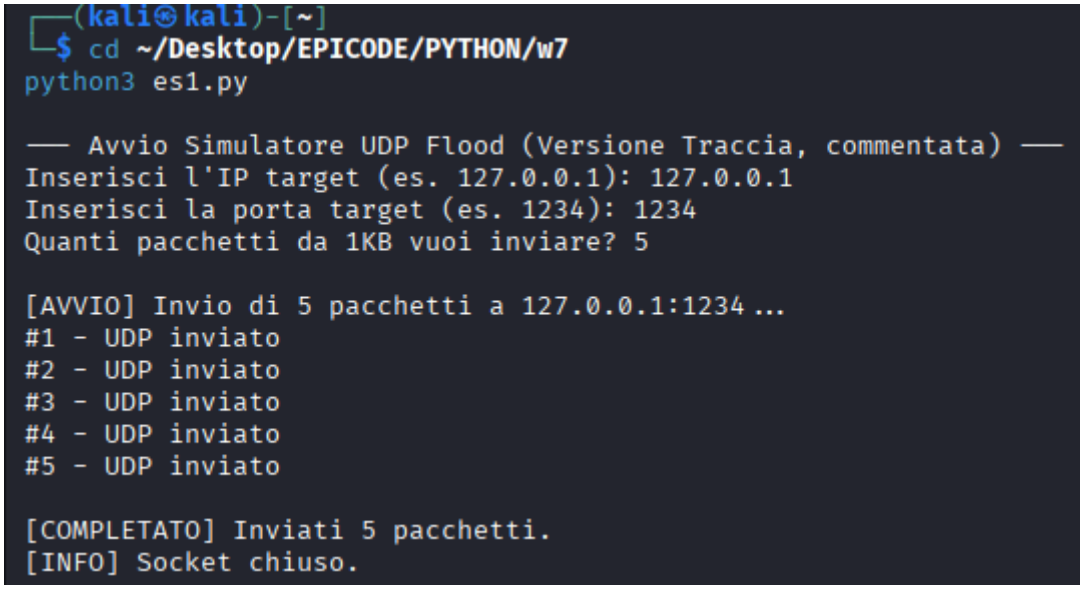
CREZIONE SOCKET ED INVIO PACCHETTI

```python
# 3) CREAZIONE SOCKET ED IVIO PACCHETTI
sent = 0  # contatore dei pacchetti inviati con successo

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # Crea il socket UDP (AF_INET = IPv4, SOCK_DGRAM = UDP)
    print(f"\n[AVVIO] Invio di {packet_count} pacchetti a {target_ip}:{target_port}...") # Stampa un messaggio di avvio che riporta i

    # NOTE: nelle slide il "while True" è mostrato commentato per evitare flood infinito.
    # Ho usato  un ciclo for per inviare un numero definito di pacchetti.
    # while True:  # <- non attivato: se decommentato causa invio continuo (flood)

    for i in range(1, packet_count + 1): # Inizia un ciclo che ripete l'invio per il numero di pacchetti scelto dall'utente
        pkt = build_packet_1kb() # Prepara il pacchetto di 1KB che verrà inviato nel ciclo
        s.sendto(pkt, (target_ip, target_port)) # Invia il pacchetto al target (IP, porta) usando sendto()

        sent += 1 #incrementa il contatore dei pacchetti inviati con successo
        print(f"#{i} - UDP inviato") # Stampa un messaggio di conferma per ogni pacchetto inviato, mostrando il numero progressivo


    print(f"\n[COMPLETATO] Inviati {sent} pacchetti.")  # Alla fine del ciclo riepilogo dei pacchetti inviati

except Exception as e: # Cattura tutte le eccezioni generiche (Exception) e le assegna alla variabile 'e' per visualizzarne il dettag
    print(f"\n[ERRORE] Durante invio: {e}")  # stampa l' errore per il debug

finally: # Blocco eseguito sempre, sia in caso di errore che di esecuzione corretta, per chiudere il socket in modo sicuro

    try: # Inizia il blocco di codice principale che potrebbe generare errori durante la creazione del socket o l'invio dei pacchetti
        s.close() # Chiude il socket per liberare le risorse di rete e terminare correttamente la connessione UDP

    except: # Gestisce eventuali errori durante la chiusura del socket senza interrompere il programma
        pass # Non viene eseguita alcuna azione, serve solo a completare il blocco except

    print("[INFO] Socket chiuso.") # Conferma a schermo che il socket è stato chiuso correttamente# Conferma a schermo che il socket
```

In questa sezione viene realizzata la parte operativa principale dello script: la creazione del **socket UDP**, il ciclo di invio dei pacchetti e la gestione ordinata di eventuali errori.
Per prima cosa viene inizializzato il contatore **sent**, utilizzato per tracciare quanti pacchetti sono stati effettivamente inviati con successo.
All'interno del **blocco try** viene creato il socket tramite **socket.socket(socket.AF_INET, socket.SOCK_DGRAM)** iparametro **AF_INET** indica che verrà utilizzato il protocollo *IPv4* (Internet Protocol versione 4), mentre **SOCK_DGRAM** specifica che il socket opererà in modalità *UDP* (User Datagram Protocol), cioè basato sull'invio di datagrammi senza connessione diretta tra mittente e destinatario.
Subito dopo, il programma mostra un messaggio informativo che riassume quanti pacchetti verranno inviati e verso quale **combinazione** di **indirizzo IP e porta** così da confermare in modo esplicito i parametri operativi prima dell'avvio dell'esecuzione.
L'invio vero e proprio è gestito da un **ciclo for** che itera dal pacchetto 1 fino al valore indicato dall'utente, ad ogni iterazione viene richiamata la **funzione build_packet_1kb()** che genera il payload di 1024 byte; il pacchetto così costruito viene inviato al target tramite il metodo **sendto(pkt, (target_ip, target_port)).**
Dopo ogni invio, il contatore **sent** viene incrementato e viene stampato un messaggio di conferma (**#n - UDP inviato**) che permette di seguire in tempo reale l'avanzamento dell'operazione e di correlare facilmente gli invii con i datagram osservati dagli strumenti di monitoraggio.
Al termine del ciclo, un messaggio riepilogativo conferma il numero totale di pacchetti inviati.
Il **blocco except Exception as e** intercetta eventuali anomalie durante la creazione del socket o durante la fase di invio (ad esempio errori di rete o parametri non raggiungibili).

In caso di eccezione, viene mostrato un messaggio d'errore con il dettaglio dell'eccezione, così da facilitare l'analisi e il debug, evitando interruzioni brusche e poco leggibili.
La struttura si chiude con il **blocco finally**, che viene eseguito in ogni caso, sia in presenza di errore sia in condizioni normali.
All'interno del finally il socket viene chiuso esplicitamente con **s.close(),** racchiuso a sua volta in un piccolo **try/except** per gestire eventuali errori in fase di chiusura senza compromettere l'uscita del programma.
L'ultimo messaggio [INFO] Socket chiuso. conferma che le risorse di rete sono state rilasciate correttamente.
L' organizzazione **try per la fase critica di invio**, **except per la gestione centralizzata degli errori** e **finally per la chiusura garantita del socket** assicura che lo script sia robusto, tracciabile e non lasci risorse aperte, anche in caso di problemi durante l'esecuzione.

ESECUZIONE UDP

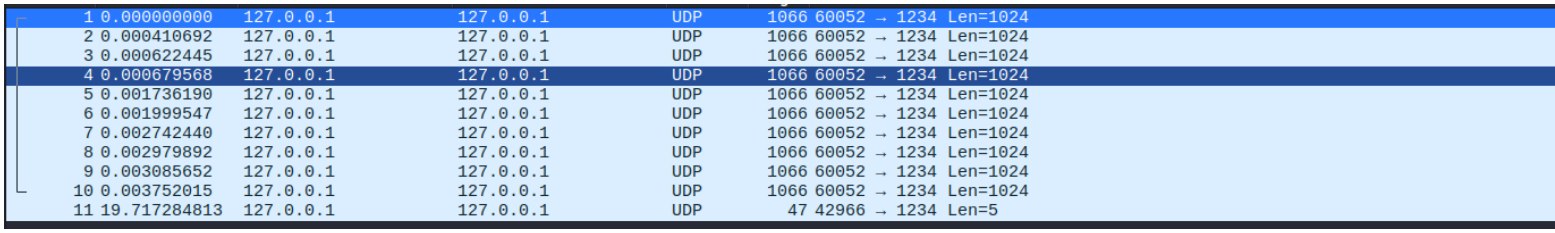


Durante l'esecuzione del programma vengono richiesti all'utente tre parametri fondamentali: **l'indirizzo IP di destinazione**, la porta **UDP** e il **numero di pacchetti da inviare**.
Nel caso mostrato l'indirizzo scelto è il **127.0.0.1** (loopback locale), la porta **1234** e un totale di **5 pacchetti da 1 KB** ciascuno.
Una volta avviato, lo script genera e invia in sequenza i pacchetti al target indicato, fornendo per ognuno un messaggio di conferma numerato (#1 - UDP inviato, #2 - UDP inviato, ecc.).
Al termine dell'invio, viene visualizzato un riepilogo che conferma l'avvenuto completamento dell'operazione con il numero totale di pacchetti trasmessi e un messaggio di chiusura del socket a garanzia della corretta terminazione della connessione e del rilascio delle risorse di rete, inoltre la cattura dei pacchetti è stata verificata contemporaneamente con Wireshark.

VERIFICA RICEZIONI DATI UDP

Durante la fase di test è stato aperto un **listener locale** sulla porta UDP 1234 utilizzando il **comando nc -u -l -p 1234**, in modo da verificare la ricezione effettiva dei pacchetti generati dallo script.
I dati visualizzati a schermo sono costituiti da byte casuali, coerenti con il contenuto del payload di 1024 byte generato tramite la funzione **random.getrandbits(8)**.
La presenza di flussi binari privi di significato testuale conferma il corretto invio e ricevimento dei datagrammi all'interno dell'interfaccia loopback (127.0.0.1), dimostrando la funzionalità del socket e la coerenza del payload previsto.

VERIFICA CON WIRESHARK



1	0.000000000	127.0.0.1	127.0.0.1	UDP	1066 60052 → 1234 Len=1024
2	0.000410692	127.0.0.1	127.0.0.1	UDP	1066 60052 → 1234 Len=1024
3	0.000622445	127.0.0.1	127.0.0.1	UDP	1066 60052 → 1234 Len=1024
4	0.000679568	127.0.0.1	127.0.0.1	UDP	1066 60052 → 1234 Len=1024
5	0.001736190	127.0.0.1	127.0.0.1	UDP	1066 60052 → 1234 Len=1024
6	0.001999547	127.0.0.1	127.0.0.1	UDP	1066 60052 → 1234 Len=1024
7	0.002742440	127.0.0.1	127.0.0.1	UDP	1066 60052 → 1234 Len=1024
8	0.002979892	127.0.0.1	127.0.0.1	UDP	1066 60052 → 1234 Len=1024
9	0.003085652	127.0.0.1	127.0.0.1	UDP	1066 60052 → 1234 Len=1024
10	0.003752015	127.0.0.1	127.0.0.1	UDP	1066 60052 → 1234 Len=1024
11	19.717284813	127.0.0.1	127.0.0.1	UDP	47 42966 → 1234 Len=5

Per confermare il corretto invio dei pacchetti generati dallo script, è stata avviata una cattura in **Wireshark** sull'interfaccia **Loopback (lo)**, applicando il *filtro udp.port == 1234*.
Dalla cattura risultano chiaramente visibili i datagrammi UDP provenienti dall'indirizzo **127.0.0.1** e diretti verso lo stesso indirizzo locale, con **porta sorgente variabile** e **porta di destinazione 1234**.
Ogni pacchetto presenta una **lunghezza complessiva di 1066 byte**, corrispondente a 1024 byte di payload più gli header di protocollo UDP/IP.
Questa evidenza conferma che i pacchetti di 1 KB sono stati effettivamente generati e trasmessi in sequenza, in conformità con le specifiche richieste della traccia.

CONCLUSIONE

L’attività ha dimostrato in modo pratico come sia possibile generare e monitorare il traffico UDP attraverso uno script controllato. Il test ha confermato il corretto funzionamento del programmache ha inviato pacchetti di dimensione predefinita verso il target locale, con risultati verificabili sia dal terminale sia tramite Wireshark.
L’esperimento ha permesso di comprendere in maniera chiara il comportamento del protocollo UDP e la logica che sta alla base dei meccanismi di flood, evidenziando al tempo stesso l’importanza del controllo e del monitoraggio del traffico di rete.
L’attività, pur svolta in ambiente sicuro e circoscritto, ha fornito un esempio concreto di come anche semplici script possano generare un impatto rilevante sulle comunicazioni di rete, sottolineando il valore della prevenzione e dell’analisi nella gestione della sicurezza informatica.

GLOSSARIO

UDP (User Datagram Protocol) – Protocollo di trasporto senza connessione che consente l'invio rapido di pacchetti di dati ("datagrammi") senza richiedere una conferma di ricezione.
È usato in applicazioni dove la velocità è prioritaria rispetto all'affidabilità, come streaming o gaming online.
Datagramma – Unità di dati inviata tramite UDP.
Ogni datagramma viaggia in modo indipendente e può arrivare a destinazione in ordine diverso o non arrivare affatto.
Socket – Interfaccia software che permette la comunicazione tra due programmi attraverso la rete.
In questo esercizio è stato utilizzato un socket UDP per inviare pacchetti da una macchina all'altra.
Loopback (127.0.0.1) – Indirizzo IP riservato al test locale.
Tutto il traffico diretto a 127.0.0.1 rimane all'interno del sistema stesso, senza uscire sulla rete.
Porta (Port) – Numero che identifica un processo o servizio di rete in ascolto su un sistema.
Nell'esercizio, la porta 1234 è stata utilizzata come destinazione dei pacchetti UDP.
Payload – Contenuto effettivo di un pacchetto, ovvero i dati trasmessi all'interno del datagramma, esclusi gli header di protocollo.
Wireshark – Strumento di analisi di rete che consente di catturare e visualizzare i pacchetti in transito, utile per verificare e diagnosticare le comunicazioni tra dispositivi o applicazioni.
Netcat (nc) – Utility da riga di comando che consente di creare connessioni TCP o UDP per testare e simulare la comunicazione tra host. In questo contesto è stata utilizzata per ricevere i pacchetti inviati dallo script.
Flood (UDP Flood) – Tecnica di invio massivo di pacchetti UDP verso un target, utilizzata in ambito didattico per simulare situazioni di sovraccarico o denial of service.
127.0.0.1:1234 – Rappresentazione dell'indirizzo IP (127.0.0.1) e della porta (1234) che identificano univocamente la destinazione dei pacchetti all'interno della macchina locale.