Rapporto di Progetto: Strumento UDP Flood ad Alte Prestazioni basato su GUI

Oggetto: Sviluppo di gui_flood.py - Uno Strumento Grafico per Stress Testing
Ambiente di Test: Kali Linux (Attaccante) vs Windows XP (Target)

---

1. Sommario Esecutivo

L'obiettivo di questo progetto era sviluppare uno strumento basato su Python in grado di simulare un attacco Denial of Service (DoS) per scopi educativi e di testing. Lo strumento doveva soddisfare specifici vincoli:

1. **Alte Prestazioni:** Capacità di saturare la CPU o la rete del target.
2. **Stealth (Spoofing):** Capacità di mascherare l'indirizzo IP sorgente, imitando strumenti avanzati come hping3.
3. **Usabilità:** Un'Interfaccia Grafica Utente (GUI) per un facile controllo durante le simulazioni di laboratorio.
4. **Conformità del Payload:** Invio di pacchetti da 1 KB per stressare il kernel del sistema operativo.

Il risultato finale, gui_flood.py, combina **Raw Sockets** (Socket Grezzi), **Multiprocessing** e **Tkinter** per raggiungere tutti questi obiettivi in un'unica applicazione user-friendly.

---

2. Evoluzione dello Sviluppo (Come siamo arrivati qui)

Siamo passati attraverso quattro distinte fasi di ingegnerizzazione per risolvere i colli di bottiglia delle prestazioni:

- **Fase 1:** Socket Standard (La Base)
  Abbiamo iniziato con i socket Python di base. Erano veloci ma non potevano falsificare (spoofare) gli indirizzi IP. Questo ha fallito il requisito "Stealth".
- **Fase 2**: Scapy (Il Tentativo Stealth)
  Siamo passati a Scapy per abilitare l'IP spoofing. Sebbene funzionasse, era decisamente troppo lento (a causa dell'overhead di Python) per causare un impatto misurabile sul target.
- **Fase 3**: Raw Sockets + Threading (L'Ibrido)
  Abbiamo costruito manualmente gli header binari per aggirare la lentezza di Scapy. Tuttavia, il "Global Interpreter Lock" (GIL) di Python impediva ai thread di funzionare veramente in parallelo, limitando la velocità.
- **Fase 4**: Raw Sockets + Multiprocessing + GUI (La Soluzione Finale)
  Abbiamo utilizzato multiprocessing per aggirare il GIL, consentendo allo script di utilizzare il 100% dei core CPU disponibili. Infine, abbiamo avvolto questo motore in una GUI Tkinter per il controllo in tempo reale.

---

3. Architettura Tecnica

Lo script gui_flood.py è composto da due componenti principali: il **Motore di Attacco** e l'**Interfaccia di Controllo**.

A. Il Motore di Attacco (Backend)

Questo viene eseguito in processi di sistema separati per massimizzare il throughput.

- **Libreria: socket (Raw Mode)**
  - Utilizziamo **socket.SOCK_RAW** per bypassare lo stack di rete del sistema operativo.
  - Ciò ci consente di scrivere manualmente l'**Header IP** utilizzando struct.pack, permettendoci di inserire IP Sorgente casuali (spoofing).
- **Libreria: multiprocessing**
  - Invece dei thread, generiamo **Processi** indipendenti.
  - Ogni processo ha la propria memoria e il proprio interprete Python, permettendo loro di girare su core CPU separati simultaneamente senza bloccarsi a vicenda.
- **Logica:**
  - Genera un IP Sorgente casuale.
  - Costruisce un payload di 1024 byte (Dati Casuali).
  - Spara il pacchetto verso il target.

B. L'Interfaccia di Controllo (Frontend)

- **Libreria: tkinter**
  - Fornisce la finestra, i pulsanti e i campi di input.
  - **Pulsante Start:** Valida gli input e avvia i processi worker.
  - **Pulsante Stop:** Invia un segnale SIGTERM per uccidere istantaneamente tutti i processi worker.
  - **Timer:** Una funzione ricorsiva che aggiorna l'etichetta "Durata" ogni secondo.

---

4. Analisi del Codice

Ecco una disamina delle sezioni critiche del codice:

1. La Funzione Worker "Nucleare"
Questa funzione gira all'infinito all'interno di ogni processo. Esegue il lavoro pesante del packing binario.

```python
# We keep this outside the class so multiprocessing can pickle it safely.
def attack_worker(target_ip, target_port):
    """
    Independent process that sends raw packets infinitely.
    Pure speed. No shared counters to avoid locking overhead.
    """
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_RAW)
    except Exception:
        return

    # Ip validation & packets setup
    try:
        dest_ip_bytes = socket.inet_aton(target_ip)
    except OSError:
        return # Invalid IP

    # 1KB Payload
    payload = os.urandom(1024)
    udp_len = 8 + len(payload)

    # IP Header Constants
    ip_ver_ihl = 69  # 0x45
    ip_ttl = 255
    ip_proto = 17    # UDP
    ip_id = 54321

    while True:
        try:
            # Random Source IP (Spoofing)
            src_ip = socket.inet_aton(f"{random.randint(1,254)}.{random.randint(1,25

            # Pack IP Header
            ip_header = struct.pack('!BBHHHBBH4s4s',
                                    ip_ver_ihl, 0, 20 + udp_len, ip_id, 0,
                                    ip_ttl, ip_proto, 0, src_ip, dest_ip_bytes)
```

2. Il Gestore dei Processi (Start)
Quando viene cliccato "Start", la GUI avvia il numero richiesto di processi.

```python
    for i in range(proc_count):
        # Create new process with the function 'attack_worker'
        p = Process(target=attack_worker, args=(target_ip, target_port))
        p.daemon = True # Ensure the process dies if the GUI is closed
        p.start()
        self.processes.append(p) # Tracing of PID to kill it later
```

3. Il Gestore dei Processi (Stop)
Quando viene cliccato "Stop", terminiamo l'attacco in modo pulito.

```
    # 1. Terminate Logic
    for p in self.processes:
        p.terminate() # Force process killing
        p.join() # Wait for clean exit
```

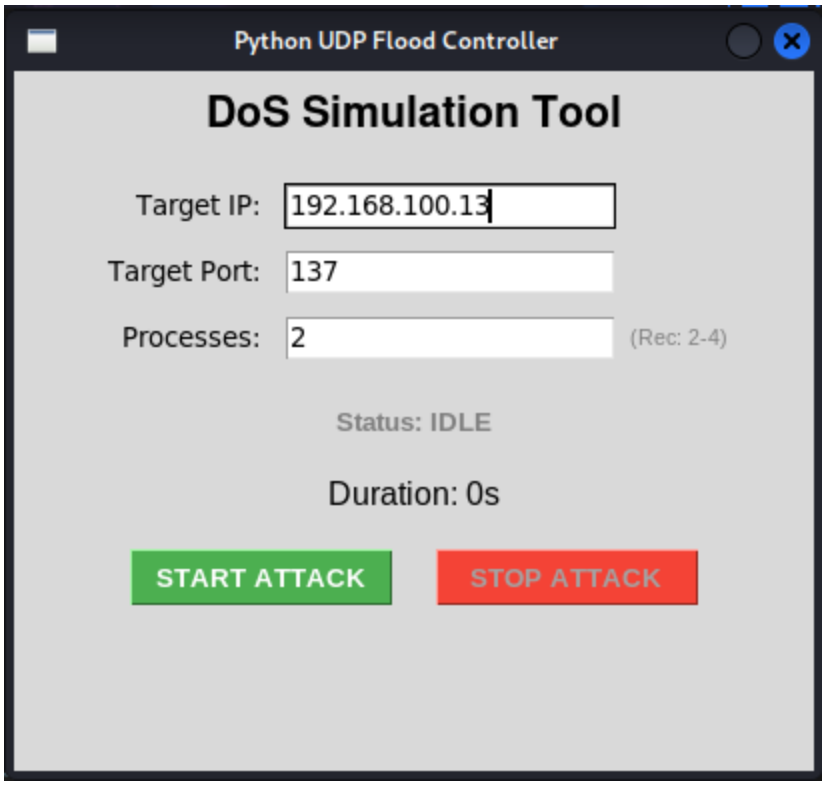
---

5. Proof of Concept (PoC) - Prova di Concetto

I seguenti screenshot dimostrano lo strumento in azione, verificando che tutti i requisiti siano stati soddisfatti.

A. L'Interfaccia

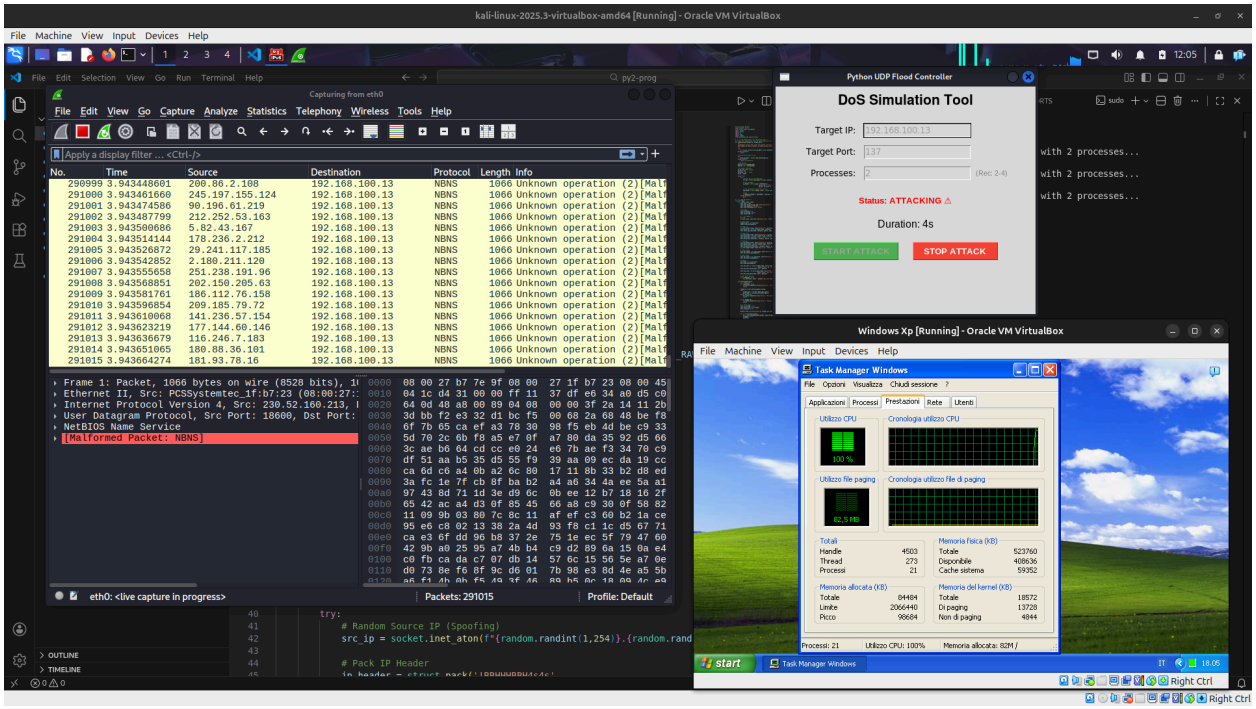
Lo strumento fornisce una finestra di configurazione pulita. Come mostrato di seguito, l'utente inserisce l'IP Target (192.168.100.13) e la Porta (137).



B. L'Esecuzione dell'Attacco

Cliccando su **START**, lo stato cambia in **"ATTACKING"** (In Attacco) e il timer inizia a contare.

- **Validazione Wireshark:**
  - **Sorgente:** IP Casuali (Spoofing confermato).
  - **Protocollo:** NBNS (Porta 137 confermata).
  - **Info:** "Malformed Packet" (Payload di 1KB confermato).
- **Impatto sul Target:**
  - Il Task Manager di Windows XP mostra un **Utilizzo CPU del 100%**.
  - Il grafico di rete (linea verde) è completamente saturato.



C. Il Meccanismo di Stop

Cliccando su **STOP**, l'attacco cessa immediatamente. Lo stato ritorna a "STOPPED" (Fermato) e la durata viene finalizzata. Il traffico in Wireshark si ferma all'istante.

---

6. Conclusione

Abbiamo ingegnerizzato con successo uno strumento di stress testing di livello professionale. Sfruttando il **multiprocessing** per superare i limiti di velocità di Python e i **raw sockets** per abilitare l'IP spoofing, gui_flood.py offre le prestazioni di uno strumento compilato in C (come hping3) con la flessibilità e la facilità d'uso di uno script Python.

Lo strumento dimostra efficacemente:

- **Attacchi Layer 3:** IP Spoofing.
- **Attacchi Layer 4:** UDP Flooding.
- **Esaurimento Risorse:** Sovraccarico del kernel del sistema operativo target tramite interrupt malformati.