📘 Dispense Master-Level: Python per la Cybersecurity

Modulo: Cyber Security & Ethical Hacking - S2L4 **Focus:** Socket Programming, Port Scanning, HTTP Enumeration

---

Introduzione: Python nell'Ethical Hacking

1. 📖 Spiegazione Approfondita

Studiare Python nel contesto della sicurezza informatica non serve solo a "saper programmare", ma è una competenza strategica fondamentale. L'obiettivo principale è duplice:

1. **Analisi Strutturale:** Comprendere come sono scritti i tool di sicurezza (spesso open source) permette di analizzarne il funzionamento intimo, modificarli o fixarli se necessario.
2. **Automazione (Scripting):** Un Ethical Hacker deve spesso automatizzare task ripetitivi o creare exploit personalizzati al volo. Python è il linguaggio standard de facto per queste operazioni grazie alla sua sintassi chiara e alle potenti librerie di rete.

In questo modulo, non ci limitiamo alla teoria, ma costruiamo tre strumenti reali: un gestore di **Socket di rete** (Server TCP), un **Port Scanner** e un **Rilevatore di verbi HTTP**.

3. 🧠 Analisi Tecnica & Memorizzazione

- **Definizione Chiave:** Python agisce come "colla" tra i vari sistemi operativi e protocolli di rete, permettendo la creazione rapida di socket grezzi (raw sockets) per manipolare il traffico TCP/IP.
- **Contesto Reale:** Durante un Penetration Test, potresti dover scrivere uno script veloce per fare *fuzzing* su un servizio sconosciuto o per estrarre dati (data exfiltration) eludendo i controlli standard.

Socket di Rete (Server TCP)

1. 📖 Spiegazione Approfondita

I socket sono i punti terminali (endpoint) di un canale di comunicazione bidirezionale tra due programmi in esecuzione su una rete. Immaginateli come le "prese elettriche" attraverso cui passano i dati. Il codice analizzato implementa un **Server TCP** che rimane in ascolto su un indirizzo IP e una porta specifici. Il flusso logico è rigoroso:

1. **Creazione:** Si istanzia l'oggetto socket.
2. **Binding:** Si associa il socket a un IP e una porta locali (`bind`).
3. **Listening:** Si mette il socket in modalità di ascolto, definendo una coda di attesa (`listen`).
4. **Accepting:** Il server accetta una connessione in ingresso. Questa è una chiamata *bloccante*: il codice si ferma finché un client non si connette. Restituisce un *nuovo* oggetto socket specifico per quella connessione.

5. **Receiving & Decoding:** Si ricevono i dati grezzi (bytes), che devono essere decodificati (es. in UTF-8) per essere leggibili.

2. 🛠 Sintassi & Comandi (Cleaning)

Ecco il codice corretto e commentato (ripulito da errori OCR come `while 11` o `while 27`):

```
import socket

# Configurazione del Server
SRV_ADDR = "192.168.56.102" # IP dell'interfaccia su cui ascoltare
SRV_PORT = 44444            # Porta di ascolto

# 1. Creazione del socket (IPv4, TCP)
# AF_INET = Address Family IPv4
# SOCK_STREAM = Protocollo TCP
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# 2. Binding (Associazione IP:Porta)
s.bind((SRV_ADDR, SRV_PORT))

# 3. Listen (Inizio ascolto)
# Il parametro 1 indica la dimensione della coda (backlog)
s.listen(1)

print("Server started! Waiting for connections...")

# 4. Accept (Accettazione connessione)
# Restituisce: connection (il socket per comunicare), address (IP e porta
del client)
connection, address = s.accept()
print('Client connected with address:', address)

# 5. Ciclo infinito per ricezione dati
while True:
    # Ricezione buffer di 1024 byte
    data = connection.recv(1024)

    # Se non ci sono dati, il client ha chiuso la connessione
    if not data: break

    # Decodifica da bytes a stringa e stampa
    print(data.decode('utf-8'))

    # Opzionale: invio conferma al client
    # connection.sendall(b"Message Received\n")

connection.close()
```

Comando Lato Client (per testare): Per connettersi a questo server, si usa `netcat` (spesso abbreviato `nc`) da un terminale client (es. Kali Linux):

> `nc [IP_SERVER] [PORTA]` *Esempio:* `nc 192.168.56.102 44444`

3. 🧠 Analisi Tecnica & Memorizzazione

- **Socket.AF_INET:** Indica l'uso di indirizzi IPv4.
- **Socket.SOCK_STREAM:** Indica l'uso del protocollo TCP (orientato alla connessione, affidabile).
- **Backlog (in `listen`):** Il numero di connessioni non accettate che il sistema può mettere in coda prima di rifiutarne di nuove.
- **Decode:** I socket trasmettono *byte*, non stringhe. È obbligatorio usare `.decode('utf-8')` in ricezione e `.encode('utf-8')` in invio.

4. 🔗 Cross-Connection Master (Integrazione Extra-Slide)

In un contesto di **Malware Analysis** o creazione di **Backdoor**, questo codice rappresenta la base di una *Bind Shell*. Se al posto di stampare i dati ricevuti, il programma li passasse a un terminale di sistema (es. `/bin/bash`) e restituisse l'output, avremmo creato un controllo remoto completo del computer.

5. ⚠ Focus Esame

- **Domanda:** Qual è la differenza tra il socket `s` e il socket `connection` nel codice sopra?
  - *Risposta:* `s` è il socket di ascolto (master) che attende nuove connessioni. `connection` è il socket effimero creato specificamente per gestire lo scambio dati con *quel* singolo client connesso.
- **Trabocchetto:** Cosa succede se ometti `s.bind()`?
  - *Risposta:* Il sistema operativo non saprà su quale porta ascoltare e, chiamando `listen()`, verrà generata un'eccezione.

Port Scanner

1. 📖 Spiegazione Approfondita

Il Port Scanning è la prima fase attiva del *Footprinting*. Lo script analizzato permette di scansionare un range di porte su un target specifico per identificare servizi attivi. La logica si basa sul "tentativo di connessione":

1. Si chiede all'utente l'IP target e il range di porte (es. "1-100").
2. Si esegue il parsing della stringa per ottenere porta di inizio e fine.
3. Si itera con un ciclo `for` su ogni porta del range.
4. Per ogni porta, si tenta una connessione TCP. Se la connessione ha successo (codice di ritorno 0), la porta è **APERTA**. Se fallisce, è **CHIUSA** (o filtrata).

2. 🛠 Sintassi & Comandi (Cleaning)

```
import socket

target = input("Enter the IP address to scan: ")
portrange = input("Enter the port range to scan (es 5-200): ")

# Parsing dell'input "min-max"
```

```
lowport = int(portrange.split('-')[0])
highport = int(portrange.split('-')[1])

print('Scanning host', target, 'from port', lowport, 'to port', highport)

# Ciclo di scansione (nota: range esclude l'estremo superiore, quindi +1
idealmente)
for port in range(lowport, highport + 1):
    # Creazione socket ad ogni iterazione
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Impostazione timeout per velocità (consigliato, non presente slide
originale ma cruciale)
    s.settimeout(1)

    # connect_ex restituisce un codice di errore invece di lanciare
un'eccezione
    status = s.connect_ex((target, port))

    if status == 0:
        print('*** Port', port, '- OPEN ***')
    else:
        # Opzionale: stampare le porte chiuse crea molto rumore
        print('Port', port, '- CLOSED')

    s.close()
```

3. 🧠 Analisi Tecnica & Memorizzazione

- **`connect_ex()` vs `connect()`:** Questa è la distinzione fondamentale.
  - `connect()`: Se fallisce, lancia un'eccezione Python (`ConnectionRefusedError`), che interrompe il programma se non gestita con try/except.
  - `connect_ex()`: Restituisce un intero. `0` significa successo. Qualsiasi altro numero è un codice di errore C-style (errno). È ideale per gli scanner perché semplifica il flusso logico (if/else).
- **String Parsing:** L'uso di `split('-')` trasforma "80-100" in una lista `['80', '100']`, permettendo l'accesso agli indici `[0]` e `[1]`.

4. 🔗 Cross-Connection Master (Integrazione Extra-Slide)

Questo script è una versione rudimentale di **Nmap** (`nmap -sT -p [range] [target]`). Mentre Nmap è ottimizzato, asincrono e capace di *Service Version Detection*, questo script esegue una "TCP Connect Scan" sequenziale (lenta e rumorosa nei log del firewall).

5. ⚠ Focus Esame

- **Domanda:** Perché creiamo e chiudiamo il socket (`s = socket...`, `s.close()`) *dentro* il ciclo for?
  - *Risposta:* Perché un socket TCP, una volta chiuso o utilizzato per un tentativo fallito, non è riutilizzabile per una nuova connessione verso una porta diversa. Ne serve uno nuovo per ogni tentativo.

Rilevatore Verbi HTTP (HTTP Method Enumeration)

1. 📖 Spiegazione Approfondita

I server web rispondono a diversi "verbi" (metodi) HTTP. Oltre ai classici `GET` (richiedere una pagina) e `POST` (inviare dati), esistono verbi amministrativi come `PUT`, `DELETE`, o `TRACE` che, se abilitati erroneamente, rappresentano gravi vulnerabilità. L'esempio utilizza la libreria standard `http.client` per inviare una richiesta con il verbo **OPTIONS**. Il verbo `OPTIONS` è progettato specificamente per chiedere al server: "Quali metodi supporti su questa risorsa?".

2. 🛠 Sintassi & Comandi (Cleaning)

```
import http.client

host = input("Inserire host/IP del sistema target: ")
port = input("Inserire la porta del sistema target (default:80): ")

if port == "":
    port = 80 # Gestione default

try:
    # 1. Creazione connessione HTTP
    connection = http.client.HTTPConnection(host, port)

    # 2. Invio richiesta OPTIONS sulla root "/"
    connection.request('OPTIONS', '/')

    # 3. Ottenimento risposta
    response = connection.getresponse()

    # 4. Lettura header o status
    # Nota: Spesso le info sono nell'header 'Allow', non solo nello status
    print("Codice risposta:", response.status)
    print("Metodi abilitati (Header Allow):", response.getheader("Allow"))

    connection.close()

except ConnectionRefusedError:
    print("Connessione fallita")
except Exception as e:
    print(f"Errore generico: {e}")
```

3. 🧠 Analisi Tecnica & Memorizzazione

- **HTTP Verbs pericolosi:**
  - `PUT`: Permette di caricare file sul server (potenziale *Remote Code Execution* se si caricano shell).
  - `DELETE`: Permette di cancellare risorse.
  - `TRACE`: Può portare a vulnerabilità XST (Cross-Site Tracing).
- **Oggetto `response`:** Contiene lo stato (es. 200 OK), ma le informazioni cruciali sui metodi sono spesso contenute negli **Header HTTP** della risposta, specificamente nell'header `Allow`.

Esercizi di Programmazione (Analisi Logica)

Di seguito l'analisi logica degli esercizi proposti nelle slide, classificati per difficoltà.

Livello Facile

1. **Somma numeri:** Input utente -> casting a `int` -> operazione aritmetica `+`.
2. **Pari/Dispari:** Uso dell'operatore modulo `%`. Se `numero % 2 == 0` è pari, altrimenti dispari.
3. **Lunghezza Stringa:** Funzione built-in `len(stringa)`.
4. **Anno Bisestile:** Logica booleana complessa. Un anno è bisestile se:
    - Divisibile per 4 AND (NON divisibile per 100 OR divisibile per 400).
    - Codice: `if (anno % 4 == 0 and anno % 100 != 0) or (anno % 400 == 0):`

Livello Medio

1. **Fattoriale:** Moltiplicazione cumulativa di un numero per i suoi antecedenti fino a 1 (es. 5! = 5*4*3*2*1). Si risolve con un ciclo `for` o ricorsione.
2. **Fibonacci:** Sequenza dove ogni numero è la somma dei due precedenti (0, 1, 1, 2, 3, 5...). Richiede due variabili di stato per tracciare `n-1` e `n-2`.
3. **Invertire Stringa:** In Python si usa lo slicing avanzato: `stringa[::-1]`.

Livello Difficile

1. **Conteggio Parole:**
    - Utilizzare `stringa.split()` per ottenere una lista di parole.
    - Utilizzare un dizionario (hash map) per contare le occorrenze: chiave=parola, valore=conteggio.
2. **Statistica (Media, Mediana, Moda):**
    - *Media:* Somma elementi / numero elementi.
    - *Mediana:* Ordinare la lista (`sort`) e prendere l'elemento centrale.
    - *Moda:* L'elemento più frequente (richiede conteggio occorrenze come sopra).

5. ⚠ Focus Esame (Esercizi)

- **Input:** Ricorda sempre che `input()` restituisce una **stringa**. Per fare calcoli matematici devi convertire esplicitamente con `int()` o `float()`.
- **Cicli:** Attenzione ai loop infiniti (`while True`) senza condizioni di uscita (`break`), comuni quando si gestiscono socket server.