

A Level Computer Science Non-Examined Assessment (NEA)

Sam Leonard

Contents

1	Analysis	2
1.1	Identification and Background to the Problem	2
1.2	Analysis of problem	12
1.3	Success Criteria	18
1.4	Description of current system or existing solutions	18
1.5	Prospective Users	24
1.6	Data Dictionary	24
1.7	Data Flow Diagram	25
1.8	Description of Solution Details	27
1.9	Acceptable Limitations	37
1.10	Test Strategy	37
2	Design	38
2.1	Overall System Design (High Level Overview)	38
2.2	Design of User Interfaces HCI	38
2.3	System Algorithms	40
2.4	Input data Validation	45
2.5	Proposed Algorithms for complex structures (flow charts or Pseudo Code)	45
3	Technical Solution	47
3.1	Overview to direct the examiner to areas of complexity and explain design evidence	47
4	Testing	47
4.1	Test Plan	47
4.2	Test Table / Testing Evidence	47
4.2.1	Printing a usage message when run without parameters	47
4.2.2	Printing a help message when passed -h	47
4.2.3	Printing a help message when passed -help	48
4.2.4	Translating a CIDR specified subnet into a list of IP addresses	49
4.2.5	Scanning a subnet with ICMP ECHO REQUEST messages	50
4.2.6	Scanning without first checking whether hosts are up.	51
4.2.7	Detecting whether a TCP port is open	53
4.2.8	Detecting whether a TCP port is closed	54
4.2.9	Detecting whether a TCP port is filtered	55
4.2.10	Detecting whether a UDP port is open	56
4.2.11	Detecting whether a UDP port is closed	57
4.2.12	Detecting whether a UDP port is filtered	57
4.2.13	Detecting the operating system of another machine	58
4.2.14	Detecting the service and its version running behind a port	59

5	Evaluation	60
5.1	Reflection on final outcome	60
5.2	Evaluation against objectives, end user feedback	60
5.3	Potential improvements	60
6	Appendices	60
6.1	icmp_ping	60
6.2	ping_scanner	62
6.3	subnet_to_addresses	64
6.4	tcp_scan	65
6.4.1	connect_scan	65
6.4.2	syn_scan	66
6.5	udp_scan	69
6.6	version_detection	75
6.7	modules	80
6.8	examples	115
6.9	netscan	116
6.10	tests	121

1 Analysis

1.1 Identification and Background to the Problem

The problem I am trying to solve with my project is how to look at devices on a network from a “black box” perspective and gain information about what services are running etc. Services are programs which their entire purpose is to provide a *service* to other programs, for example a server hosting a website would be running a service whose purpose is to send the webpage to people who try to connect to the website.

There are many steps in-between a device turning on to interacting with the internet.

1. load networking drivers
2. Starting Dynamic Host Configuration Protocol (DHCP) daemon
3. Broadcasting DHCP request for an IP address
4. Get assigned an IP address

There are many more steps than I have listed above but these are the most important ones. Starting from a linux computer being switched on the first step is that the kernel needs to load the networking drivers. The kernel is the basis for the operating system, it is what interacts with the hardware in the most fundamental way. drivers are small bits of code which the kernel can load in order to interact with certain hardware modules such as the Network Interface Card (NIC) which is essential for interfacing with the network, hence the name.

Next once the kernel has loaded the required drivers and the system has booted the networking ‘daemons’ must be started. In linux a daemon is a program that runs all the time in the background to serve a specific purpose or utility. For example when I start my laptop the following daemons start upowerd (power management), systemd (manages the creation of all processes), dbus-daemon (manages inter-process communication), iwd (manages my WiFi connections) and finally Dynamic Host Configuration Protocol Client Daemon (DHCPD) which manages all interactions with the network around DHCP.

Once the daemons are all started the DHCP client can now take issue commands to the daemon for it to carry out. The DHCP client is simply a daemon that runs in the background to carry out any interactions between the current machine and the DHCP server. The DHCP server is normally the WiFi router or network switch for the local network and it manages a list of which computer has which IP address and negotiates with new computers trying to join a network to get them a free IP address. The DHCP client starts the DHCP address negotiation with the server by sending a discover message with the address 255.255.255.255 which is the IP limited broadcast address which means that whatever is listening at the other end will forward this packet on to everyone on the subnet. When the DHCP server (normally the router, sometimes a separate machine) on the subnet receives this message it reserves a free IP address for that client and then responds with a DHCP offer which contains the address the server is offering, the length of time the address is valid for and the subnet mask of the network. The client must then respond with a DHCP request message to request the offered address, this is in case of multiple DHCP servers offering addresses. Finally the DHCP server responds with a DHCP acknowledge message showing that it has received the request. Figure 2 shows a packet capture from my laptop where I turned WiFi off, started wireshark listening and plugged in an Ethernet cable, I have it showing only the DHCP packets so that it is clear to see the entire DHCP negotiation including the 255.255.255.255 limited broadcast destination address and the 0.0.0.0 unassigned address in the source column. I mention using wireshark to do packet capturing above without explaining what either packet capturing or wireshark are so I will do that here. Packets I define below and wireshark is simply a tool which intercepts all the network communications on a single computer and records them to a file as well as displaying them to the user as well as performing some analysis and dissecting each of the protocols used. This means that I can record the DHCP negotiation shown below and show it to you using wireshark to get all the information out of the packets being sent over the wire.

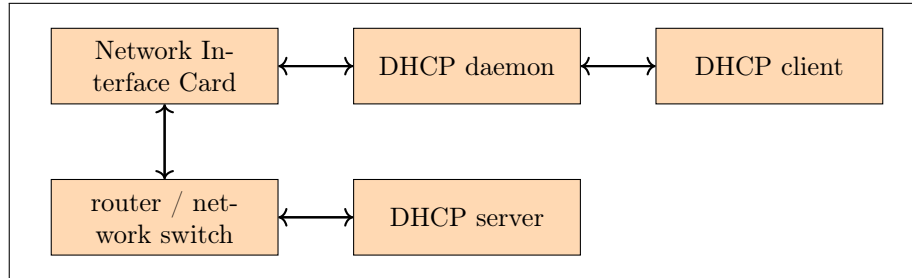


Figure 1: A block diagram showing the relationship between different elements of a DHCP negotiation.

No.	Time	Source	Destination	Protocol	Info
6	0.983737378	0.0.0.0	255.255.255.255	DHCP	DHCP Discover
32	4.239092378	192.168.1.1	192.168.1.47	DHCP	DHCP Offer
34	4.239420587	0.0.0.0	255.255.255.255	DHCP	DHCP Request
36	4.241743101	192.168.1.1	192.168.1.47	DHCP	DHCP ACK

Figure 2: DHCP address negotiation

All computer networking is encapsulated in the Open Systems Interconnection model (OSI model) which has 7 layers:

7. Application: Applications Programming Interface (API)s, Hypertext transfer Protocol (HTTP), File Transfer Protocol (FTP) among others.
6. Presentation: encryption/decryption, encoding/decoding, decompression etc...
5. Session: Managing sessions, PHP Hypertext Processor (PHP) session IDs etc...
4. Transport: TCP and UDP among others.
3. Network: ICMP and IP among others.
2. Data Link: MAC addressing, Ethernet protocol etc...
1. Physical: The physical Ethernet cabling/NIC.

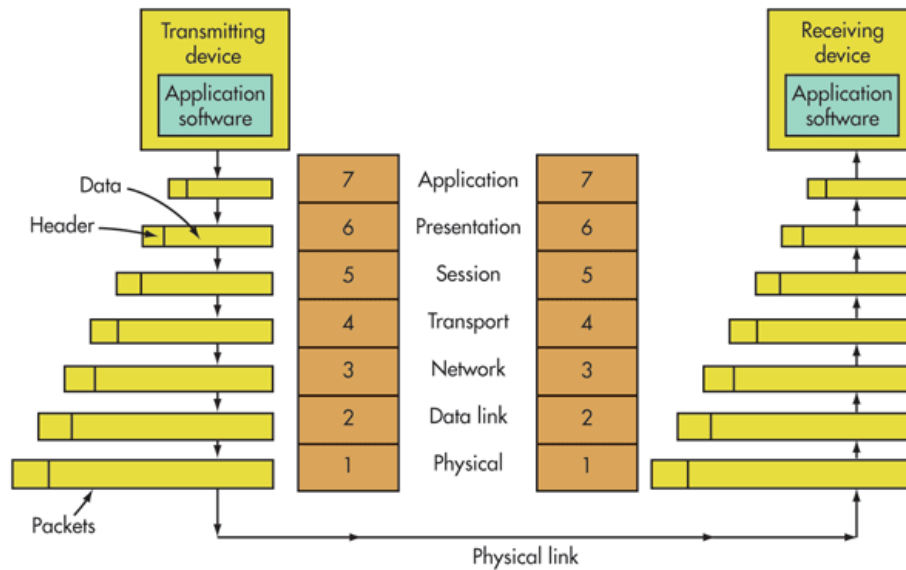


Figure 3: OSI model diagram, source: <https://www.electronicdesign.com>

Each of these layers is essential to the running of the internet but a single communication might not include all of the layers. These communications are all based on the most fundamental part of the internet: the packet. Packets are sequences of ones and zeros sent between computers which are used to transfer data as well as to control how networks function. They consist of different layers of information each specifying where the packet where should go next at a different level along with fundamentally the data/instructions contained in the innermost layer. When packets are sent between computers a certain number of layers are stripped off by each computer so that it knows where to send the packet next at which point it will add all the layers back again, this time with the instructions needed to go from the current computer to the next one on its route. Each of these layers actually consists of a number of fields at the start called a header some layers also append a footer to the end of the packet. The actual data being transferred in the packet can be quite literally anything, HTTP transfers websites so Hypertext Markup Language (HTML) files and images etc. . . . In particular there are two pieces of information stored in headers which together define the final destination of the packet: the IP address and the port number. The IP address defines the destination machine and the port number defines which “port” on the remote machine the packet should be sent to. Ports are essential entrances to a computer, for example if a computer was a hotel the IP address would be the address and location of the hotel and the port number would be the room inside the hotel. There are 65535 ports and 0 is a special reserved port. Both Transmission Control Protocol

(TCP) and User Datagram Protocol (UDP) use ports, TCP ports are mainly used for transferring data where reliability is a concern, as TCP has built in checks for packet loss whereas UDP does not and as such is used for purposes where speed is more important and missing some data is inconsequential, such as video streaming and playing games.

I'm going to use the example of getting a very simple static HTML page with an image inside. The code for the page is shown in listing 1. In figure 4 you can see how the page renders. However far more interestingly is how the browser retrieved the page, in figure 5 you can see the full sequence of packets that were exchanged for the browser to get the resources it needed to render the page. I am hosting the page using Python3's `http.server` module which is super convenient and just makes the current directory open on port 8000 from there I can just navigate to `/example.html` and it will render the page. Breaking figure 5 down packet one shows the browser receiving the request from the user to display `http://192.168.1.47:8000/example.html` and attempting to connect to 192.168.1.47 on port 8000. Packets two and three show the negotiation of this request through to the full connection being made. The browser now makes an HTTP GET request for the page `example.html` over the established TCP connection as shown in packet 4. The server then acknowledges the request and sends a packet with the PSH flag set as shown in packets 6 and 7. The PSH flag is a request to the browser to say that it is OK to received the buffered data, i.e. `example.html`. The browser then sends back an acknowledgement and the server sends the page as shown in packets 7 and 8. Finally the browser sends a final acknowledgement of having received the page before initiating a graceful session teardown by sending a FIN ACK packet which indicates the end of a session. Once the server responds to the FIN ACK with it's own the browser sends a final acknowledgement. This then repeats itself when the browser parses the HTML and realises theres an image which it needs to get from the server as well, except the image is a larger file and so takes a few more PSH packets. In figures 6 and 7 you can see a set of ladder diagrams which show the entire transaction symbolically. I have also colour coded figure 7 with green arrow heads to the initial handshakes, blue for the HTTP protocol transactions and red for the TCP connection teardown packets.

This shows clearly the interaction between each of the different layers in the OSI model, the browser at level 7: Application rendering the webpage. Level 6: Presentation is skipped as we have no files which need to be served compressed because they are so large. Level 5: Session is shown by the TCP session negotiation and graceful teardown of the TCP session. Level 4: Transport is shown when the image and webpage are transferred from the server to the browser. Level 3/2/1 are shown in figure 8 where you can see the IP layer information along with Ethernet II and finally frame 4 which is the bytes that went down the wire.

This is a really big heading

wow para

graphs a

re amazi

ng

No.	Time	Source	Destination	Protocol	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	56196 → 12345 [SYN] Seq=0 Win=43690 Len=
2	0.000009524	127.0.0.1	127.0.0.1	TCP	12345 → 56196 [RST, ACK] Seq=1 Ack=1 Win=
3	6.808420598	127.0.0.1	127.0.0.1	TCP	56198 → 12345 [SYN] Seq=0 Win=43690 Len=
4	7.830566490	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
5	9.842573743	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
6	13.942571238	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
7	22.130575535	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
8	38.258578004	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]

[Toggle image](#)

Figure 4: A basic static HTML webpage.

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help					
Apply a display filter... <Ctrl-/> Expression... +					
No.	Time	Source	Destination	Protocol	Info
1	0.000000000	192.168.1...	192.168.1...	TCP	57790 → 8000 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 S
2	0.000622552	192.168.1...	192.168.1...	TCP	8000 → 57790 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0
3	0.000646626	192.168.1...	192.168.1...	TCP	57790 → 8000 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSva
4	0.000806427	192.168.1...	192.168.1...	HTTP	GET /example.html HTTP/1.1
5	0.001032018	192.168.1...	192.168.1...	TCP	8000 → 57790 [ACK] Seq=1 Ack=363 Win=64896 Len=0 TS
6	0.002978389	192.168.1...	192.168.1...	TCP	8000 → 57790 [PSH, ACK] Seq=1 Ack=363 Win=64896 Len=0
7	0.002991460	192.168.1...	192.168.1...	TCP	57790 → 8000 [ACK] Seq=363 Ack=186 Win=30336 Len=0
8	0.003141019	192.168.1...	192.168.1...	HTTP	HTTP/1.0 200 OK (text/html)
9	0.003152622	192.168.1...	192.168.1...	TCP	57790 → 8000 [ACK] Seq=363 Ack=779 Win=31488 Len=0
10	0.003952333	192.168.1...	192.168.1...	TCP	57790 → 8000 [FIN, ACK] Seq=363 Ack=779 Win=31488 L
11	0.004220421	192.168.1...	192.168.1...	TCP	8000 → 57790 [ACK] Seq=779 Ack=364 Win=64896 Len=0
12	0.026948474	192.168.1...	192.168.1...	TCP	57792 → 8000 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 S
13	0.027523772	192.168.1...	192.168.1...	TCP	8000 → 57792 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0
14	0.027544820	192.168.1...	192.168.1...	TCP	57792 → 8000 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSva
15	0.027678073	192.168.1...	192.168.1...	HTTP	GET /document/screenshots/packet_drop.png HTTP/1.1
16	0.027932568	192.168.1...	192.168.1...	TCP	8000 → 57792 [ACK] Seq=1 Ack=432 Win=64768 Len=0 TS
17	0.030230298	192.168.1...	192.168.1...	TCP	8000 → 57792 [PSH, ACK] Seq=1 Ack=432 Win=64768 Len=0
18	0.030238964	192.168.1...	192.168.1...	TCP	57792 → 8000 [ACK] Seq=432 Ack=188 Win=30336 Len=0
19	0.030330743	192.168.1...	192.168.1...	TCP	8000 → 57792 [ACK] Seq=188 Ack=432 Win=64768 Len=43
20	0.030337416	192.168.1...	192.168.1...	TCP	57792 → 8000 [ACK] Seq=432 Ack=4532 Win=39040 Len=0
21	0.030381844	192.168.1...	192.168.1...	TCP	8000 → 57792 [ACK] Seq=4532 Ack=432 Win=64768 Len=5
22	0.030388177	192.168.1...	192.168.1...	TCP	57792 → 8000 [ACK] Seq=432 Ack=10324 Win=50560 Len=
23	0.030429506	192.168.1...	192.168.1...	TCP	8000 → 57792 [ACK] Seq=10324 Ack=432 Win=64768 Len=
24	0.030434304	192.168.1...	192.168.1...	TCP	57792 → 8000 [ACK] Seq=432 Ack=13220 Win=56448 Len=
25	0.030479143	192.168.1...	192.168.1...	TCP	8000 → 57792 [ACK] Seq=13220 Ack=432 Win=64768 Len=
26	0.030484516	192.168.1...	192.168.1...	TCP	57792 → 8000 [ACK] Seq=432 Ack=16116 Win=62208 Len=
27	0.030603768	192.168.1...	192.168.1...	TCP	8000 → 57792 [ACK] Seq=16116 Ack=432 Win=64768 Len=
28	0.030612973	192.168.1...	192.168.1...	TCP	57792 → 8000 [ACK] Seq=432 Ack=21908 Win=73728 Len=
29	0.030643425	192.168.1...	192.168.1...	TCP	8000 → 57792 [ACK] Seq=21908 Ack=432 Win=64768 Len=
30	0.030655076	192.168.1...	192.168.1...	TCP	57792 → 8000 [ACK] Seq=432 Ack=26252 Win=82432 Len=
31	0.030695063	192.168.1...	192.168.1...	TCP	8000 → 57792 [ACK] Seq=26252 Ack=432 Win=64768 Len=
32	0.030700281	192.168.1...	192.168.1...	TCP	57792 → 8000 [ACK] Seq=432 Ack=32044 Win=94080 Len=
33	0.030745441	192.168.1...	192.168.1...	TCP	8000 → 57792 [ACK] Seq=32044 Ack=432 Win=64768 Len=
34	0.030750695	192.168.1...	192.168.1...	TCP	57792 → 8000 [ACK] Seq=432 Ack=37836 Win=105600 Len=
35	0.030793610	192.168.1...	192.168.1...	HTTP	HTTP/1.0 200 OK (PNG)
36	0.030799924	192.168.1...	192.168.1...	TCP	57792 → 8000 [ACK] Seq=432 Ack=42612 Win=115200 Len=
37	0.030883862	192.168.1...	192.168.1...	TCP	57792 → 8000 [FIN, ACK] Seq=432 Ack=42612 Win=11520
38	0.031107867	192.168.1...	192.168.1...	TCP	8000 → 57792 [ACK] Seq=42612 Ack=433 Win=64768 Len=
get_webpage_perfect.pcapng Packets: 38 · Displayed: 38 (100.0%) Profile: Default					

Figure 5: A full chain of packets that shows retrieving a basic webpage from the server.



Figure 6: Ladder diagram of figure 5.

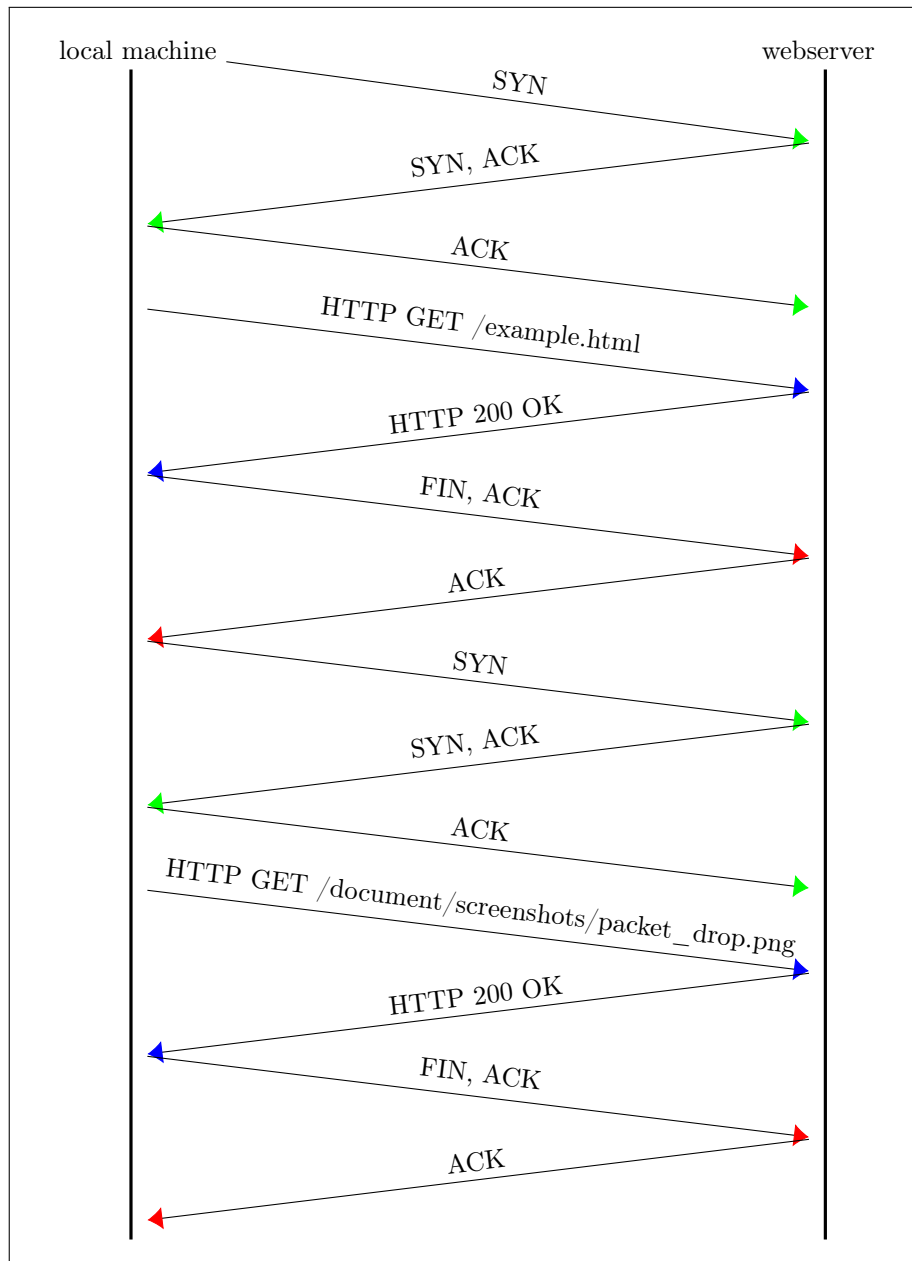


Figure 7: A simplified ladder diagram showing the transaction in figure 5

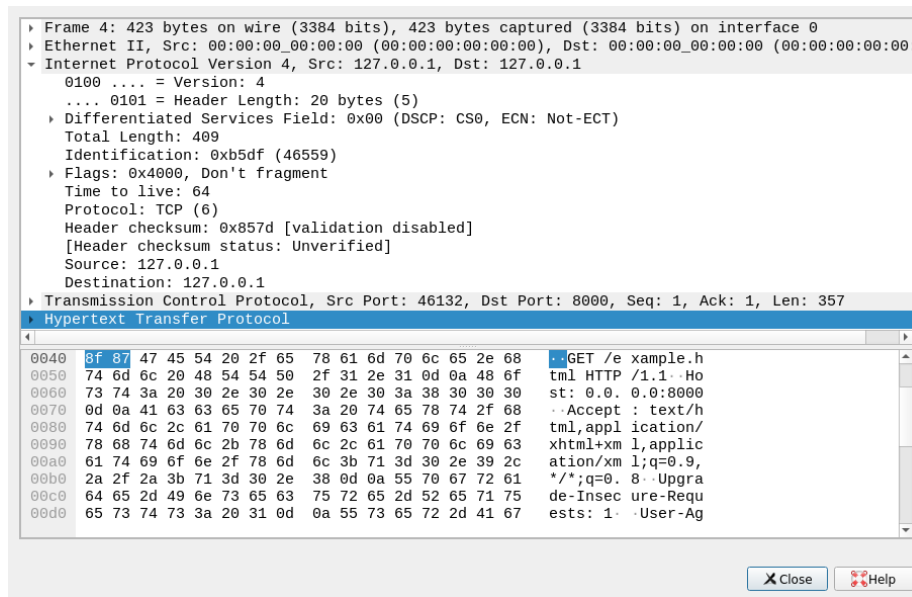


Figure 8: A look inside a TCP packet.

Listing 1: example.html

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>Wow I can add titles</title>
5  </head>
6  <body>
7
8  <h1>This is a really big heading</h1>
9  <p>wow para</p>
10 <p>graphs a</p>
11 <p>re amazi</p>
12 <p>ng</p>
13 <script type="text/javascript">
14   function imgtog() {
15     if (document.getElementById("img").style.display == "none") {
16       document.getElementById("img").style = "block"
17     } else {
18       document.getElementById("img").style.display = "none"
19     }
20   }
21
22 </script>
23

```

```
24 
25
26 <button onclick="imgtog()">Toggle image</button>
27
28
29 </body>
30 </html>
```

1.2 Analysis of problem

The problem with looking at a network from the outside is that the purpose of the network is to allow communication inside of the network, thus very little is exposed externally. This presents a challenge as we want to know what is on the network as well as what each of them is running which is not always possible due to the limited information that services will reveal about themselves. Firewalls also play large part in making scanning networks difficult as sometimes they simply drop packets instead of sending a TCP RST packet (reset connection packet). When firewalls drop packets it becomes exponentially more difficult as you don't know whether your packet was corrupted or lost in transit or if it was just dropped.

To demonstrate this I will show three things:

1. A successful connection over TCP.
2. An attempted connection to a closed port.
3. An attempted connection with a firewall rule to drop packets.

Firstly A successful TCP connection. For a TCP connection to be established there is a three way handshake between the communicating machines. Firstly the machine trying to establish the connection sends a TCP SYN packet to the other machine, this packet holds a dual purpose, to ask for a connection and if it is accepted to SYNchronise the sequence numbers being used to detect whether packets have been lost in transport. The receiving machine then replies with a TCP SYN ACK which confirms the starting sequence number with the SYN part and ACKnowledges the connection request. The sending machine then acknowledges this by sending a final TCP ACK packet back. This connection initialisation is shown in figure 9 by packets one, two and three. Data transfer can then commence by sending a TCP packet with the PSH and ACK flags set along with the data in the data portion of the packet, this is shown in figure 12 where wireshark allows us to take a look inside the packet to see the data being sent in the packet along with the PSH and ACK flags being set. The code I used to generate these is shown in figures 10 and 11. Breaking the code down in figure 11 you can see me initialising a socket object then I bind it to localhost (127.0.0.1) port 12345 localhost is just an address which allows connections between programs running on the same computer as connections are

looped back onto the current machine, hence its alternative name: the loopback address. I then tell it to listen for incoming connections, the one just means how many connections to keep as a backlog. I then accept the connection from the program in figure 10, line 3. I then tell the program to listen for up to 1024 bytes in the data part of any TCP packets sent. The program in figure 10 then sends some data which we then see printed to the screen in figure 11, both programs then close the connection.

No.	Time	Source	Destination	Protocol	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [SYN] Seq=0
2	0.000019294	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [SYN, ACK]
3	0.000033431	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [ACK] Seq=1
4	53.378941809	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [PSH, ACK]
5	53.378958066	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [ACK] Seq=1
6	65.928944995	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [FIN, ACK]
7	65.936113471	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [ACK] Seq=3
8	85.536923935	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [FIN, ACK]
9	85.536940026	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [ACK] Seq=2

Figure 9: Packets starting a TCP session, transferring some data then ending it.

```

In [1]: import socket

In [2]: sender = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

In [3]: sender.connect(("127.0.0.1", 12345))

In [4]: sender.send(b"hi I'm data what's your name? "*10)
Out[4]: 300

In [5]: sender.close()

```

Figure 10: Transferring some basic text data over a TCP connection.

```
In [1]: import socket
In [2]: receiver = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
In [3]: receiver.bind(("127.0.0.1", 12345))
In [4]: receiver.listen(1)
In [5]: connection, address = receiver.accept()
In [6]: connection.recv(1024)
Out[6]: b'hi I'm data what's your name? hi I'm data what's your name? hi I'm data what's your name? hi I'm
data what's your name? hi I'm data what's your name? hi I'm data what's your name? hi I'm data what's you
r name? hi I'm data what's your name? hi I'm data what's your name? hi I'm data what's your name? "
In [7]: connection.close()
```

Figure 11: Receiving some basic text data over a TCP connection.

No.	Time	Source	Destination	Protocol	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [SYN] Seq=0
2	0.000019294	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [SYN, ACK]
3	0.000033431	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [ACK] Seq=1
4	53.378941809	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [PSH, ACK]
5	53.378958066	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [ACK] Seq=1
6	65.928944995	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [FIN, ACK]
7	65.936113471	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [ACK] Seq=3
8	85.536923935	127.0.0.1	127.0.0.1	TCP	47710 → 12345 [FIN, ACK]
9	85.536940026	127.0.0.1	127.0.0.1	TCP	12345 → 47710 [ACK] Seq=2

▶	Frame 4: 366 bytes on wire (2928 bits), 366 bytes captured (2928 bits) on
▶	Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00
▶	Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▶	Transmission Control Protocol, Src Port: 47710, Dst Port: 12345, Seq: 1,
▶	Data (300 bytes)

0000	00 00 00 00 00 00 00 00	00 00 00 00 08 00 45 00E.
0010	01 60 70 81 40 00 40 06	cb 14 7f 00 00 01 7f 00	.`p@.@.
0020	00 01 ba 5e 30 39 09 d1	70 b2 e9 c6 d7 ad 80 18	...^09..p.....
0030	01 56 ff 54 00 00 01 01	08 0a 1a 7c 9a 84 1a 7b	.V.T.... ...{
0040	ca 01 68 69 20 49 27 6d	20 64 61 74 61 20 77 68	..hi I'm data wh
0050	61 74 27 73 20 79 6f 75	72 20 6e 61 6d 65 3f 20	at's you r name?
0060	68 69 20 49 27 6d 20 64	61 74 61 20 77 68 61 74	hi I'm d ata what
0070	27 73 20 79 6f 75 72 20	6e 61 6d 65 3f 20 68 69	's your name? hi
0080	20 49 27 6d 20 64 61 74	61 20 77 68 61 74 27 73	I'm dat a what's
0090	20 79 6f 75 72 20 6e 61	6d 65 3f 20 68 69 20 49	your na me? hi I
00a0	27 6d 20 64 61 74 61 20	77 68 61 74 27 73 20 79	'm data what's y
00b0	6f 75 72 20 6e 61 6d 65	3f 20 68 69 20 49 27 6d	our name ? hi I'm
00c0	20 64 61 74 61 20 77 68	61 74 27 73 20 79 6f 75	data wh at's you
00d0	72 20 6e 61 6d 65 3f 20	68 69 20 49 27 6d 20 64	r name? hi I'm d
00e0	61 74 61 20 77 68 61 74	27 73 20 79 6f 75 72 20	ata what 's your
00f0	6e 61 6d 65 3f 20 68 69	20 49 27 6d 20 64 61 74	name? hi I'm dat
0100	61 20 77 68 61 74 27 73	20 79 6f 75 72 20 6e 61	a what's your na
0110	6d 65 3f 20 68 69 20 49	27 6d 20 64 61 74 61 20	me? hi I 'm data
0120	77 68 61 74 27 73 20 79	6f 75 72 20 6e 61 6d 65	what's y our name
0130	3f 20 68 69 20 49 27 6d	20 64 61 74 61 20 77 68	? hi I'm data wh
0140	61 74 27 73 20 79 6f 75	72 20 6e 61 6d 65 3f 20	at's you r name?
0150	68 69 20 49 27 6d 20 64	61 74 61 20 77 68 61 74	hi I'm d ata what
0160	27 73 20 79 6f 75 72 20	6e 61 6d 65 3f 20	's your name?

Figure 12: Highlighted packet carrying the data being transferred in figure 10.

Next an attempted connection to a closed port. In figure 13 packet one you can see the same TCP SYN packet as we saw in the attempted connection to an open port, as you would expect. The difference comes in the next packet with the TCP RST flag being sent back. This flag means to reset the connection, or if the connection is not yet established as in this case it means that the port is closed, hence why the packet is highlighted red in figure 13. The code used to generate this is shown in figure 14 line two shows the initialisation of a socket object. In line 3 the program tries to connect to port 12345 on localhost again, except this time we get a connection refused error back this shows us that the remote host sent a TCP RST packet back, which is reflected in figure 13.

Finally I will show a connection where the firewall is configured to drop the packet. However first I will explain a bit about firewalls and how they work.

Firewalls are essentially the gatekeepers of the internet they decide whether a packet gets to pass or whether they shall not pass. Firewalls work by a set of rules which decide what happens to it. A rule might be that it is coming from a certain IP address or has a certain destination port. The actions taken after the packet has had it's fate decided by the rules can be one of the following three (on iptables on linux): ACCEPT, DROP and RETURN, accept does exactly what you think it would an lets the packet through, drop quite literally just drops the packet and sends no reply whatsoever, return is more complicated and has no effect on how port scanning is done and as such we will ignore it. A common set of rules for something like a webserver would be to DROP all incoming packets and then allow exceptions for certain ports i.e. port 80 for HTTP or 443 for Hypertext transfer Protocol Secure (HTTPS). I will be using a linux utility called iptables for implementing all firewall rules on my system for demonstration purposes. Packet number three in figure 13 shows the connection request from line 4 of 14 except that I have enabled a firewall rule to drop all packets from the address 127.0.0.1, using the iptables command as so: `iptables -I INPUT -s 127.0.0.1 -j DROP`. This command reads as for all packets arriving (-I INPUT) with source address 127.0.0.1 (-s 127.0.0.1) drop them sending no response (-j DROP). With this firewall rule in place you can see in figure 13 packet 3 receives no response and as such Python assumes that the packet just got lost and as such tries to send the packet again repeatedly, this continued for more than 30 seconds before a stopped it as shown by the time column in figure 13 and the final KeyboardInterrupt in figure 14. The amount of time that a system will wait still trying to reconnect depends on the OS and a other factors but the minimum time is 100 seconds as specified by RFC 1122, on most systems it will be between 13 and 30 minutes according the linux manual page on TCP.

man 7 tcp:

tcp_retries2 (integer; default: 15; since Linux 2.2)

The maximum number of times a TCP packet is retransmitted in established state before giving up. The default value is 15, which corresponds to a duration of approximately between 13 to 30 minutes, depending on the retransmission timeout. The RFC 1122 specified minimum limit of 100 seconds is typically deemed too short.

No.	Time	Source	Destination	Protocol	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	56196 → 12345 [SYN] Seq=0 Win=43690 Len=
2	0.000009524	127.0.0.1	127.0.0.1	TCP	12345 → 56196 [RST, ACK] Seq=1 Ack=1 Win=
3	6.808420598	127.0.0.1	127.0.0.1	TCP	56198 → 12345 [SYN] Seq=0 Win=43690 Len=
4	7.830566490	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
5	9.842573743	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
6	13.942571238	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
7	22.130575535	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]
8	38.258578004	127.0.0.1	127.0.0.1	TCP	[TCP Retransmission] 56198 → 12345 [SYN]

Figure 13: Attempted connection to a closed port with and without firewall rule to drop packets.

```

In [1]: import socket

In [2]: a = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

In [3]: a.connect(("127.0.0.1", 12345))
-----
ConnectionRefusedError                                Traceback (most recent call last)
<ipython-input-3-fbc96d60b5f2> in <module>
----> 1 a.connect(("127.0.0.1", 12345))

ConnectionRefusedError: [Errno 111] Connection refused

In [4]: a.connect(("127.0.0.1", 12345))
^C-----
KeyboardInterrupt                                    Traceback (most recent call last)

```

Figure 14: The code used to produce firewall packet dropping example in figure 13

Having explained firewalls, how they affect port scanning and other things above I will now explain what I am actually trying to achieve with my project and how I am going to do it. I am trying to make a tool similar to nmap which will be able to detect the state (as in whether the port is open/closed or filtered etc) of ports on remote machines, detect which hosts are up on a subnet and finally I want to be able to try to detect what services are listening behind any of the ports. I am going to be writing in Python version 3.7.2 as it is the latest stable release of Python 3 and has many features which are not in even fairly recent versions such as 3.5, the biggest one of these being fstrings which are where I can put a single a 'f' before a string and then any formatting options I put inside using curly braces are expanded and formatted accordingly. This allows for a clear and consistent string formatting syntax which I will use extensively. I will be using Python in particular as a language because it is very readable and has extensive low level bindings to C networking functions with the socket module allowing me to write code quickly which is easily understandable and has a clear purpose and at the same time be able to use low level networking functions and even changing the behaviour at this low level with `socket.setsockopt`. As well as this the socket module allows me to open sockets that communicate using many different protocols such as TCP, UDP and Internet Control Message Protocol (ICMP) just to name a few. These features combine to make Python a great language for writing networking software with a high level of abstraction. In regards to the OSI model my code will sit with the user interface at level 7 specifying what to do at a high level then the actual scanning takes place at levels 3, 4 and 5 with host detection being at level 3. Port scanning will be taking place At level 4 for TCP SYN scanning and UDP scanning. Whereas `connect()` scanning and version detection will sit at level 5. Finally I will look at what is actually handling all of the networking on my machine. My machine runs linux and as such all networking is handled by system calls to the linux kernel. For example the `socket.connect` method is just

a call to the underlying linux kernel's connect syscall but presenting a kinder call signature to the user as the Python socket library does some processing before the syscall is made.

1.3 Success Criteria

1. Probe another computer's networking from a black box perspective.
2. To help the user with usage/help messages when prompted.
3. Send ICMP ECHO requests to determine whether a machine is active or not.
4. Translate Classless Inter-Domain Routing (CIDR) specified subnets into a list of domains.
5. Perform any scan type without first checking whether the host is up.
6. Detect whether a TCP port is open (can be connected to).
7. Detect whether a TCP port is closed (will refuse connections).
8. Detect whether a TCP port is filtered (a firewall is preventing or monitoring access).
9. Detect whether a UDP port is open (can be connected to).
10. Detect whether a UDP port is closed (will refuse connections).
11. Detect whether a UDP port is filtered (a firewall is preventing or monitoring access).
12. Detect the operating system of another machine on the network solely from sending packets to the machine and interpreting the responses.
13. Detect what service is listening behind a port.
14. Detect the version of the service running behind a port.

1.4 Description of current system or existing solutions

Nmap is currently the most popular tool for doing port scanning and host enumeration. It supports the scanning types for determining information about remote hosts.

- TCP: SYN
- TCP: `Connect()`
- TCP: ACK
- TCP: Window

- TCP: Maimon
- TCP: Null
- TCP: FIN
- TCP: Xmas
- UDP
- Zombie host/idle
- Stream Control Transmission Protocol (SCTP): INIT
- SCTP: COOKIE-ECHO
- IP protocol scan
- FTP: bounce scan

As well as supporting a vast array of scanning types it also can do service version detection and operating system detection via custom probes. Nmap also has script scanning which allows the user to write a script specifying exactly how they want to scan e.g. to circumvent port knocking (where packets must be sent to a sequence of ports in order before access to the final port is allowed). It also supports a plethora of options to avoid firewalls or Intrusion Detection System (IDS) such as sending packets with spoofed checksums/source addresses and sending decoy probes. Nmap can do many more things than I have listed above as is illustrated quite clearly by the fact there is an entire working on using nmap (<https://nmap.org/book/>). The following is an example nmap scan which I did on my home network: `nmap -sC -sV -oA networkscan 192.168.1.0/24`. Breaking it down this means to enable script scanning `-sc`, enable version detection `-sV` and then output all results in all the common formats: XML, nmap and greppable, using the base name `networkscan` which produces three files: `networkscan.(nmap,gmap,xml)`. Before I go into what each file contains I will explain some terminology, greppable is anything which can be easily searched with the linux `grep` which stands for Globally search a Regular Expression and Print, which basically means look in files for lines that contain a certain word or pattern, for example finding all lines with the word “hi” in them in the file “document” `grep hi document`. Onto the files: `networkscan.nmap` contains what would usually be printed by nmap while the scan is being run, it looks like this:

```
# Nmap 7.70 scan initiated Wed Apr 10 19:36:18 2019 as:
nmap -sC -sV -oA /home/tritoke/thing 192.168.1.0/24
Nmap scan report for router.asus.com (192.168.1.1)
Host is up (1.0s latency).
Not shown: 995 closed ports
PORT      STATE SERVICE      VERSION
```

```

53/tcp open domain      (generic dns response: NOTIMP)
| fingerprint-strings:
|   DNSVersionBindReqTCP:
|     version
|_   bind
80/tcp open http        ASUS WRT http admin
|_http-server-header: httpd/2.0
|_http-title: Site doesn't have a title (text/html).
515/tcp open printer
8443/tcp open ssl/http  ASUS WRT http admin
|_http-server-header: httpd/2.0
|_http-title: Site doesn't have a title (text/html).
| ssl-cert: Subject: commonName=192.168.1.1/countryName=US
| Not valid before: 2018-05-05T05:05:17
|_Not valid after: 2028-05-05T05:05:17
9100/tcp open jetdirect?
1 service unrecognized despite returning data. If you know the service/version,
please submit the following fingerprint at
https://nmap.org/cgi-bin/submit.cgi?new-service :
SF-Port53-TCP:V=7.70%I=7%D=4/10%Time=5CAE3DC5%P=x86_64-pc-linux-gnu%r(DNSV
SF:ersionBindReqTCP,20,"\0\x1e\0\x06\x85\x85\0\x01\0\0\0\0\0\0\0\x07version\
SF:x04bind\0\0\x10\0\x03")%r(DNSStatusRequestTCP,E,"\0\x0c\0\0\x90\x04\0\0
SF:\0\0\0\0\0\0");
Service Info: CPE: cpe:/o:asus:wrt_firmware

```

Above is just the report for one such device in the report as the full thing is over 200 lines long. In it you can see information such as which ports are open and what services are running behind them as this is my router you can see port 8443 which nmap has recognised to be hosting the ASUS web admin from which you can configure the route. Then after that some other associated information extracted from the server. Most of this extra information is from the `-sC` flag which is script scanning and allows advanced interaction with running services specifically to gain more information by providing specialised probing per protocol. We can also see at the end an unrecognised service which nmap shows us the data it returned and asks us to submit a new service report at a given URL if we recognise the service. This system of submitting fingerprints of services is how nmap is so good at recognising services: it has a lot of data to look at and learn from in regards to service fingerprinting.

Next `networkscan.gnmap`:

```

# Nmap 7.70 scan initiated Wed Apr 10 19:36:18 2019 as:
  nmap -sC -sV -oA /home/tritoke/networkscan 192.168.1.0/24
Host: 192.168.1.1 (router.asus.com) Status: Up
Host: 192.168.1.1 (router.asus.com) Ports: 53/open/tcp//domain//
(generic dns response: NOTIMP)/, 80/open/tcp//http//ASUS WRT http admin/,
515/open/tcp//printer///, 8443/open/tcp//ssl|http//ASUS WRT http admin/,
9100/open/tcp//jetdirect?/// Ignored State: closed (995)

```

```
Host: 192.168.1.8 (android-25a97e36c2e74456) Status: Up
Host: 192.168.1.8 (android-25a97e36c2e74456) Ports: 5060/filtered/tcp//sip///
Ignored State: closed (999)
```

Again this is not all of the file as it is very large. As you can see above all of the information is on a single line for each type of scan, this is useful if you want to scan a large number of hosts and just want to know which hosts are up you can do `grep 'Status: Up' networkscan.gnmap` which outputs this:

```
$ grep 'Status: Up' networkscan.gnmap
Host: 192.168.1.1 (router.asus.com) Status: Up
Host: 192.168.1.8 (android-25a97e36c2e74456) Status: Up
Host: 192.168.1.10 (diskstation) Status: Up
Host: 192.168.1.88 () Status: Up
Host: 192.168.1.88 () Status: Up
Host: 192.168.1.117 () Status: Up
Host: 192.168.1.159 (groot) Status: Up
Host: 192.168.1.159 (groot) Status: Up
Host: 192.168.1.176 (ET0021B7C01F2E) Status: Up
```

Showing you clearly the hosts which are online and then their host names. Other ways to use this output format would be to find out which ports are open on only one machine, or which hosts have a webserver running on them or a vulnerable version of a mail server etc. In general it is useful for when you want to filter results.

Finally we have eXtensible Markup Language (XML) format:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE nmaprun>
3 <?xml-stylesheet href="file:///usr/bin/./share/nmap/nmap.xsl"
  type="text/xsl"?>
4 <!-- Nmap 7.70 scan initiated Wed Apr 10 19:36:18 2019 as: nmap -sC -sV
  -oA /home/tritoke/thing 192.168.1.0/24 -->
5 <nmaprun scanner="nmap" args="nmap -sC -sV -oA /home/tritoke/thing
  192.168.1.0/24" start="1554921378" startstr="Wed Apr 10 19:36:18
  2019" version="7.70" xmloutputversion="1.04">
6 <verbose level="0"/>
7 <debugging level="0"/>
8 <host starttime="1554921379" endtime="1554923187"><status state="up"
  reason="syn-ack" reason_ttl="0"/>
9 <address addr="192.168.1.1" addrtype="ipv4"/>
10 <hostnames>
11 <hostname name="router.asus.com" type="PTR"/>
12 </hostnames>
13 <ports><extraports state="closed" count="995">
14 <extrareasons reason="conn-refused" count="995"/>
15 </extraports>
16 <port protocol="tcp" portid="53"><state state="open" reason="syn-ack"
  reason_ttl="0"/><service name="domain" extrainfo="generic dns
```

```

        response: NOTIMP"
        servicefp="SF-Port53-TCP:V=7.70%I=7%D=4/10%Time=5CAE3DC5%P=x86_64
17 -pc-linux-gnu%r(DNSVersionBindReqTCP,20,&quot;\0\x1e\0\x06\x85\x85\0
18 \x01\0\0\0\0\0\0\x07version\x04bind\0\0\0\x10\0\x03&quot;)%r
19 (DNSStatusRequestTCP,E,&quot;\0\x0c\0\0\x90\x04\0\0\0\0\0\0\0&quot;);"
        method="probed" conf="10"/><script id="fingerprint-strings"
        output="&#xa; DNSVersionBindReqTCP: &#xa; version&#xa; bind"><elem
        key="DNSVersionBindReqTCP">&#xa; version&#xa; bind</elem>
20 </script></port>

```

It is verbose in the extreme contains the reason why each port has the state it does as well as a vast amount of other data that the other scans didn't include as well as this it is not very human readable meaning that this format is more likely available because it is easier for other programs to parse than the other formats. As well as this the verbosity can be good if you really need to dive into why a port was marked as closed etc or the exact bytes that a service replied with.

In terms of where nmap lives in the software stack is that it is an application at level 7 when the user interacts with it but it uses several libraries which interact at level 2 which it uses to get the raw headers of the packets being sent and thus gain information from them.

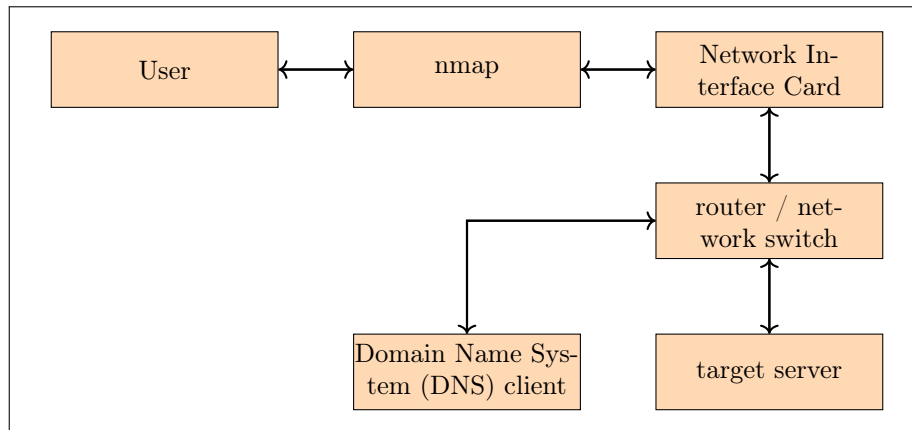


Figure 15: A block diagram showing how nmap sits in the software stack.

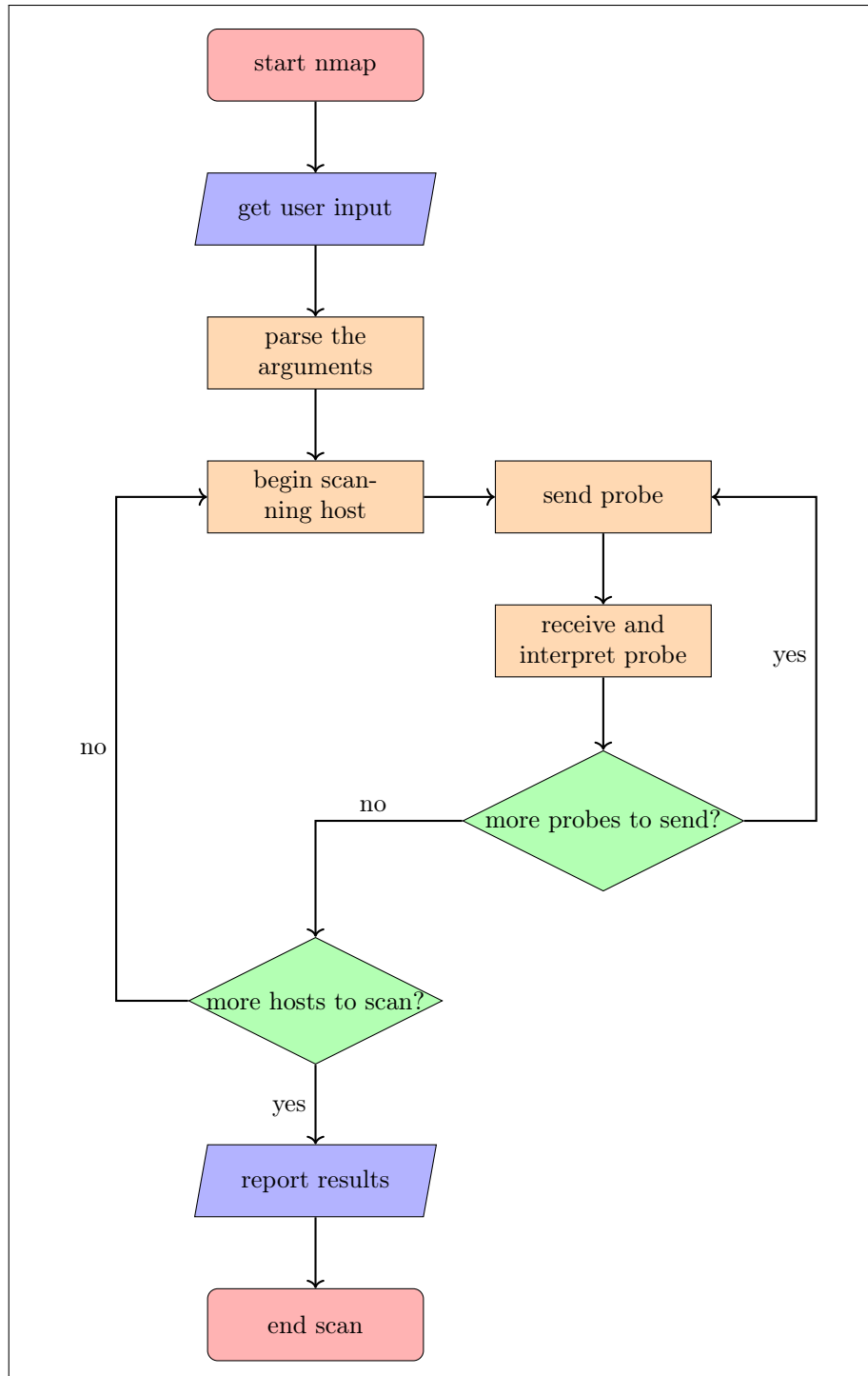


Figure 16: A flow chart showing how nmap does scanning.

1.5 Prospective Users

The prospective users of this system would be system administrators, penetration testers or network engineers. In my case my prospective users would be my school's system administrators and it would allow them to see an outsiders perspective on for example the server running the school's website page or to see if any of the programs on the servers were leaking information through banners etc. (most services send a banner with information like what protocol version they use and other information)

1.6 Data Dictionary

So while my program is running it will need to store many different things in memory:

- the list of hosts to scan
- the list of ports to scan on each host
- the state of each port we are scanning on each host
- the packet received by the listening socket (temporarily before processing)
- various counters and positional indicators are almost inevitable
- the probes to be used for version detection

So I am going to try to estimate the amount of RAM my program will use based on scanning a CIDR specified subnet of 192.168.1.0/24, and the most common ports 1000 ports of each machine I will not consider version detection as I am unsure of how I will implement it currently. To measure the size of object in python we can use the `getsizeof` function provided by the `sys` module, I also have a file called 'hosts' which contains the addresses specified by 192.168.1.0/24 and a file 'ping_bytes' which contains 4 captured packets from the ping command which I captured during an early exploratory testing phase.

Listing 2: some testing I did on the size of python objects

```
1 >>> with open("hosts", "r") as f:
2 ...     hosts = f.read().splitlines()
3 ...
4 >>> import sys
5 >>> sys.getsizeof(hosts)
6 2216
7 >>> ports = list(range(1000))
8 >>> sys.getsizeof(ports)
9 9112
10 >>> len(hosts)*sys.getsizeof(ports) / 2**10 # 2*10 is one kibibyte
11 2278.0
12 >>> sys.getsizeof(True)
13 28
```

```

14 >>> len(hosts)*(sys.getsizeof(True)) / 2**10
15 7.0
16 >>> pings[0]
17 '45 00 00 54 0f 82 40 00 40 01 2d 25 7f 00 00 01 7f 00 00 01 08 00 41 c5
    02 4f 00 01 cd ef 0f 5c de 9b 0d 00 08 09 0a 0b 0c 0d 0e 0f 10 11
    12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 26 27
    28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35 36 37'
18 >>> from binascii import unhexlify
19 >>> ping = unhexlify(pings[0].replace(" ", "")) # turn the string of
    numbers into a bytes object
20 >>> sys.getsizeof(ping)
21 117
22 >>> len(hosts)*sys.getsizeof(ping) / 2**10
23 29.25
24 >>> 2278.0 + 7.0 + 29.25 + 2.22
25 2316.47

```

As shown in Listing 2 we can see that by far the most space intensive item stored by our program will be the port numbers for each host, making up just less than ninety six percent of the total space used by the mock data I created. However overall 2.3 mebibytes is not a huge amount of data by any means.

Holding	Data type	Space used /Kib	Percentage of total
ports	List[int]	2278	98.34
hosts	List[str]	2.22	0.1
port state	List[bool]	7	0.3
packets	List[bytes]	29.25	1.26

1.7 Data Flow Diagram

In my application there will be three way information flow:

1. sending packets (data) out from my application
2. receiving packets back from the targets
3. how my program sends data around between functions

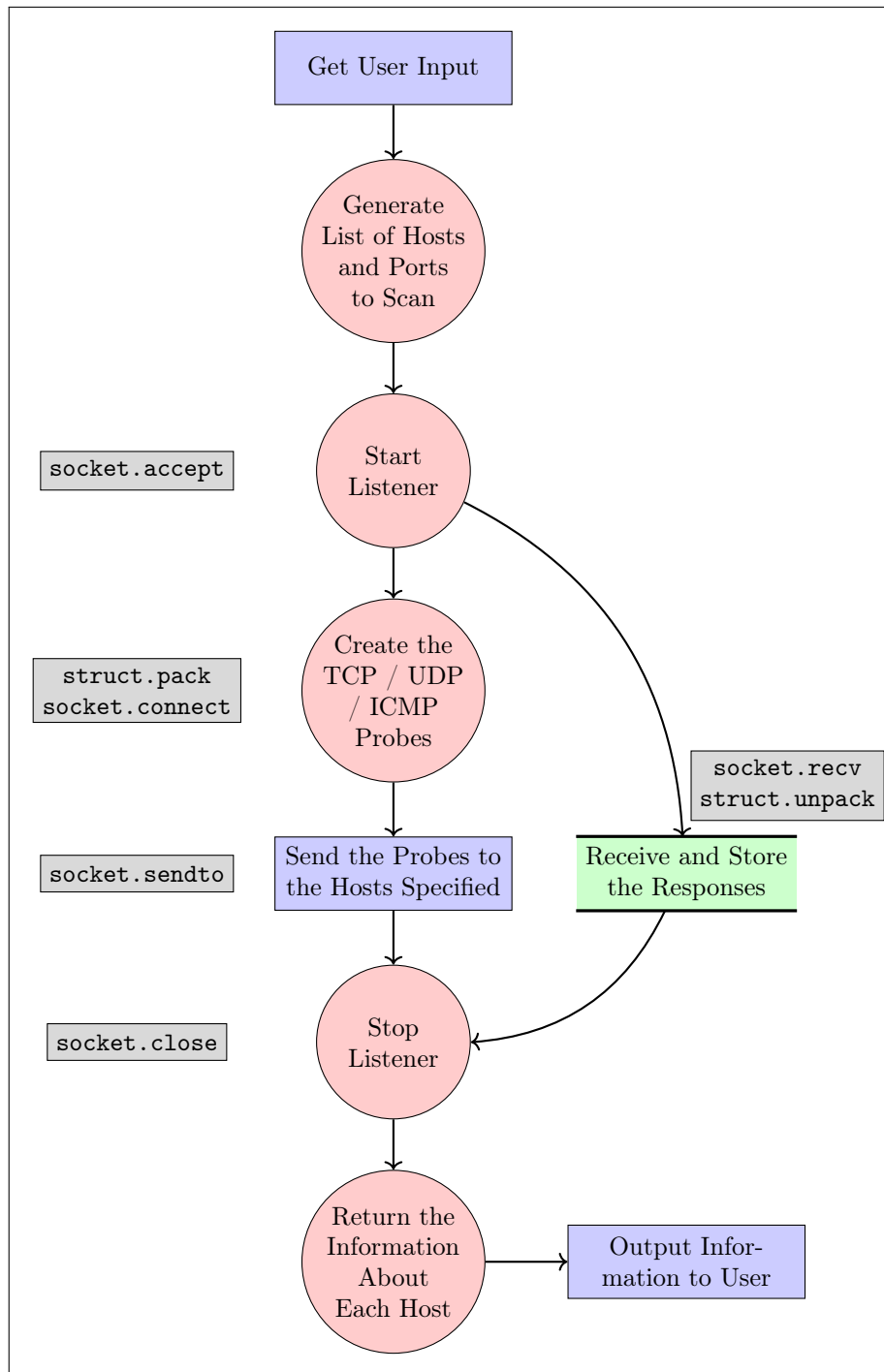


Figure 17: A data flow diagram for information in my application.

1.8 Description of Solution Details

I will be using Python version 3.7.2 for my project because I am already familiar with Python's syntax and its socket library has a very nice high level API for making system calls to the kernel's low level networking functions. This makes it very nice for a networking project like mine as it allows me to easily prototype and explore many ideas about how I could implement my solution without wasting vast amounts of time.

The first point of the success criteria that I wanted to get a feel for was receiving and sending ICMP ECHO requests aka pings. ICMP as a protocol sits at layer 3 of the OSI model this means it is a layer below what you are normally give access to in the socket module. This means instead of getting a bytes object with just the data from the header you instead get a bytes object which contains the entire packet and you have to dissect it yourself to get the information out of it, this can be quite difficult if it weren't for the struct module. The struct module provides a convenient API for converting between packed values i.e. packets in network endianness to unpacked values i.e. a double representing the current time in local endianness. Interactions with the socket module are mainly through the pack and unpack functions. For each of these functions you provide a format specifier defining how to unpack/pack the bytes/values. In Listing 3 you can see an example of me using the struct.pack function to pack the values which comprise an ICMP ECHO REQUEST into a packet and sending it the localhost address (127.0.0.1). This program is effectively the complement to the program listed in listing 4 which uses struct.unpack to unpack value from the received ICMP packet before printing the fields out to the terminal. Listing 3 makes use of the IP checksum function which I wrote. In figure 18 you can see the output when I run the command `ping 127.0.0.1` which the code in figure 4 is listening for packets.

Listing 3: A prototype for sending ICMP ECHO REQUEST packets.

```
1  #!/usr/bin/python3.7
2  import socket
3  import struct
4  import os
5  import time
6  import array
7
8  from os import getcwd, getpid
9  import sys
10 sys.path.append("../modules/")
11
12 import ip_utils
13
14
15 ICMP_ECHO_REQUEST = 8
16
17 # opens a raw socket for the ICMP protocol
```

```

18 ping_sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
    socket.IPPROTO_ICMP)
19 # allows manual IP header creation
20 # ping_sock.setsockopt(socket.SOL_IP, socket.IP_HDRINCL, 1)
21
22 ID = os.getpid() & 0xFFFF
23
24 # the two zeros are the code and the dummy checksum, the one is the
    sequence number
25 dummy_header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, 0, ID, 1)
26
27 data = struct.pack("d", time.time()) + bytes((192 -
    struct.calcsize("d")) * "A", "ascii")
28
29 checksum = ip_utils.ip_checksum(dummy_header+data)
30
31 header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, checksum, ID, 1)
32
33 packet = header + data
34
35 ping_sock.sendto(packet, ("127.0.0.1", 1))

```

Listing 4: A prototype for receiving ICMP ECHO REQUEST packets.

```

1  #!/usr/bin/python3.7
2
3  import socket
4  import struct
5  import time
6  from typing import List
7
8  # socket object using an IPV4 address, using only raw socket access, set
    ICMP protocol
9  ping_sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
    socket.IPPROTO_ICMP)
10
11 packets: List[bytes] = []
12
13 while len(packets) < 1:
14     recPacket, addr = ping_sock.recvfrom(1024)
15     ip_header = recPacket[:20]
16     icmp_header = recPacket[20:28]
17
18     ip_hp_ip_v, ip_dscp_ip_ecn, ip_len, ip_id, ip_flg_ip_off, ip_ttl,
        ip_p, ip_sum, ip_src, ip_dst = struct.unpack('!BBHHHBBHII',
        ip_header)
19
20     hl_v = f"{ip_hp_ip_v:08b}"
21     ip_v = int(hl_v[:4], 2)

```

```

22 ip_hl = int(hl_v[4:], 2)
23 dscp_ecn = f"{ip_dscp_ip_ecn:08b}"
24 ip_dscp = int(dscp_ecn[:6], 2)
25 ip_ecn = int(dscp_ecn[6:], 2)
26 flgs_off = f"{ip_flgs_ip_off:016b}"
27 ip_flgs = int(flgs_off[:3], 2)
28 ip_off = int(flgs_off[3:], 2)
29 src_addr = socket.inet_ntoa(struct.pack('!I', ip_src))
30 dst_addr = socket.inet_ntoa(struct.pack('!I', ip_dst))
31
32 print("IP header:")
33 print(f"Version: [{ip_v}]\nInternet Header Length:
      [{ip_hl}]\nDifferentiated Services Point Code:
      [{ip_dscp}]\nExplicit Congestion Notification: [{ip_ecn}]\nTotal
      Length: [{ip_len}]\nIdentification: [{ip_id:04x}]\nFlags:
      [{ip_flgs:03b}]\nFragment Offset: [{ip_off}]\nTime To Live:
      [{ip_ttl}]\nProtocol: [{ip_p}]\nHeader Checksum:
      [{ip_sum:04x}]\nSource Address: [{src_addr}]\nDestination
      Address: [{dst_addr}]\n")
34
35 msg_type, code, checksum, p_id, sequence = struct.unpack('!bbHHh',
      icmp_header)
36 print("ICMP header:")
37 print(f"Type: [{msg_type}]\nCode: [{code}]\nChecksum:
      [{checksum:04x}]\nProcess ID: [{p_id:04x}]\nSequence:
      [{sequence}]"
38 packets.append(recPacket)
39 open("current_packet", "w").write("\n".join(" ".join(map(lambda x:
      "{x:02x}", map(int, i))) for i in packets))

```

Listing 5: A function for calculating the IP checksum for a set of bytes.

```

1 def ip_checksum(packet: bytes) -> int:
2     """
3     ip_checksum function takes in a packet
4     and returns the checksum.
5     """
6     if len(packet) % 2 == 1:
7         # if the length of the packet is even, add a NULL byte
8         # to the end as padding
9         packet += b"\0"
10
11     total = 0
12     for first, second in (
13         packet[i:i+2]
14         for i in range(0, len(packet), 2)
15     ):
16         total += (first << 8) + second
17
18     # calculate the number of times a

```

```
19     # carry bit was added and add it back on
20     carried = (total - (total & 0xFFFF)) >> 16
21     total &= 0xFFFF
22     total += carried
23
24     if total > 0xFFFF:
25         # adding the carries generated a carry
26         total &= 0xFFFF
27
28     # invert the checksum and take the last 16 bits
29     return (~total & 0xFFFF)
```

```

flags: [0]
fragment offset: [0]
ttl: [64]
prot: [1]
checksum: [28457]
source address: [127.0.0.1]
destination address: [127.0.0.1]

type: [0]
code: [0]
checksum: [9703]
p_id: [39682]
sequence: [256]

version: [4]
header length: [5]
dscp: [0]
ecn: [0]
total length: [21504]
identification: [21075]
flags: [0]
fragment offset: [64]
ttl: [64]
prot: [1]
checksum: [21737]
source address: [127.0.0.1]
destination address: [127.0.0.1]

type: [8]
code: [0]
checksum: [7566]
p_id: [39682]
sequence: [512]

version: [4]
header length: [5]
dscp: [0]
ecn: [0]
total length: [21504]
identification: [21331]
flags: [0]
fragment offset: [0]
ttl: [64]
prot: [1]
checksum: [21545]
source address: [127.0.0.1]
destination address: [127.0.0.1]

type: [0]
code: [0]
checksum: [7574]
p_id: [39682]
sequence: [512]

```

Figure 18: Dissecting an ICMP ECHO REQUEST packet.

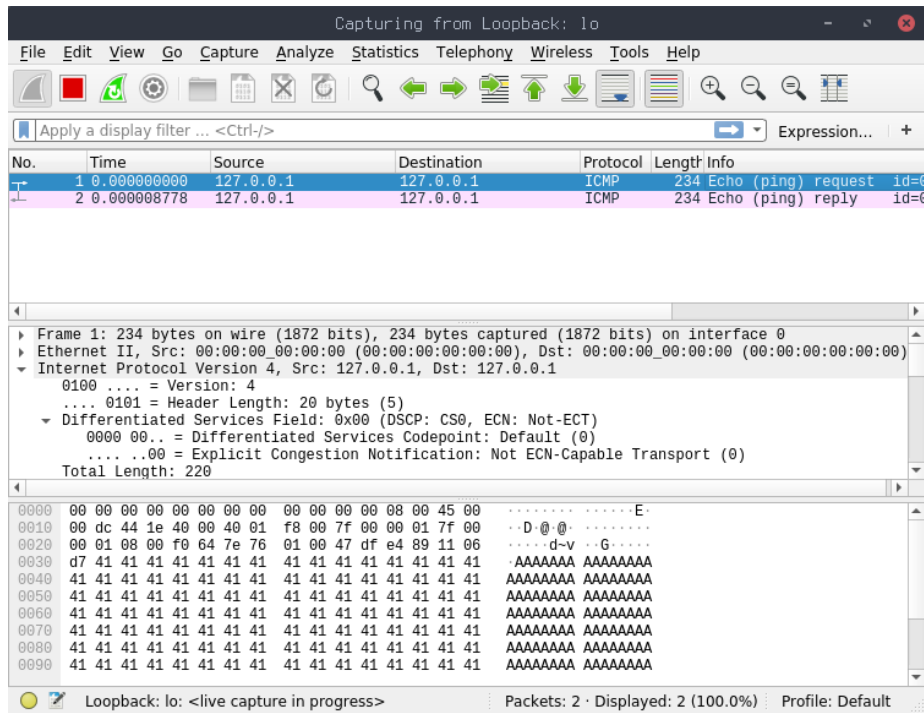


Figure 19: Screenshot of wireshark showing a successful send of an ICMP ECHO REQUEST packet.

```

1 #!/usr/bin/python
2
3 import socket
4 import struct
5 # socket object using an IPV4 address, using only raw socket access, set ICMP
6 ping_sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP
7 )
8 # this line sets the IP_HDRINCL attribute in SOL_IP to 1 allowing us to manual
9 ly create IP headers,
10 ping_sock.setsockopt(socket.SOL_IP, socket.IP_HDRINCL, 1)
11
12 while 1:
13     recPacket, addr = ping_sock.recvfrom(1024)
14     icmp_header = recPacket[20:28]
15     msg_type, code, checksum, p_id, sequence = struct.unpack('bbHh', icmp_header)
16     print("type: [" + str(msg_type) + "] code: [" + str(code) + "] checksum: ["
17           + str(checksum) + "] p_id: [" + str(p_id) + "] sequence: [" + str(sequence)
18           + "]")
19     print(" ".join(":%02h" % i for i in recPacket))
20
21 icmp-echo-send.py 16, 34 - 37 All
22 icmp-echo-send.py 16L, 775C written
23
24 tritoke@thinkpadX40:~/networkScanner/Code
25
26 PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data:
27 64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.076 ms
28 64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.093 ms
29 64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.098 ms
30 ^C
31 --- 127.0.0.1 ping statistics ---
32 3 packets transmitted, 3 received, 0% packet loss, time 25ms
33 rtt min/avg/max/mdev = 0.076/0.089/0.098/0.009 ms
34 tritoke@thinkpadX40:~/networkScanner/Code$ ping 127.0.0.1
35 PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data:
36 64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.075 ms
37 64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.093 ms
38 ^C
39 --- 127.0.0.1 ping statistics ---
40 2 packets transmitted, 2 received, 0% packet loss, time 17ms
41 rtt min/avg/max/mdev = 0.075/0.084/0.093/0.009 ms
42 tritoke@thinkpadX40:~/networkScanner/Code$ ping 127.0.0.1
43 PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data:
44 64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.084 ms
45 64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.104 ms
46 64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.098 ms
47 ^C
48 --- 127.0.0.1 ping statistics ---
49 3 packets transmitted, 3 received, 0% packet loss, time 43ms
50 rtt min/avg/max/mdev = 0.084/0.095/0.104/0.011 ms
51 tritoke@thinkpadX40:~/networkScanner/Code$

```

Figure 20: Screenshot showing me first successfully dissecting an ICMP ECHO REQUEST packet.

Having done these prototypes I have identified that it would probably be best to abstract the code for dissecting all the headers i.e. ICMP, TCP and Internet Protocol (IP) into classes where I can just pass the received packet into the class and have it dissect it for me and then I will also get access to some of the benefits of classes such as the `__repr__` method which is called when you print classes out and allows me to control what is printed out. Before I started to write the final piece I wanted to make a prototype ping scanner, as this would allow me to get a feel for making a scanner as well as further exploring low level protocol interactions.

Listing 6: An attempt at making a ping scanner.

```

1 #!/usr/bin/python3.7
2 from os import getcwd, getpid
3 import sys
4 sys.path.append("../modules/")
5
6 import ip_utils
7
8 import socket

```

```

9  from functools import partial
10 from itertools import repeat
11 from multiprocessing import Pool
12 from contextlib import closing
13 from math import log10, floor
14 from typing import List, Tuple
15 import struct
16 import time
17
18
19 def round_significant_figures(x: float, n: int) -> float:
20     """
21     rounds x to n significant figures.
22     round_significant_figures(1234, 2) = 1200.0
23     """
24     return round(x, n-(1+int(floor(log10(abs(x))))))
25
26
27 def recieved_ping_from_addresses(ID: int, timeout: float) ->
28     List[Tuple[str, float, int]]:
29     """
30     Takes in a process id and a timeout and returns the list of
31     addresses which sent
32     ICMP ECHO REPLY packets with the packed id matching ID in the time
33     given by timeout.
34     """
35     ping_sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
36                               socket.IPPROTO_ICMP)
37     time_remaining = timeout
38     addresses = []
39     while True:
40         time_waiting = ip_utils.wait_for_socket(ping_sock,
41                                                  time_remaining)
42         if time_waiting == -1:
43             break
44         time_recieved = time.time()
45         recPacket, addr = ping_sock.recvfrom(1024)
46         ip_header = recPacket[:20]
47         ip_hp_ip_v, ip_dscp_ip_ecn, ip_len, ip_id, ip_flg_ip_off,
48             ip_ttl, ip_p, ip_sum, ip_src, ip_dst =
49             struct.unpack('!BBHHHBBHII', ip_header)
50         icmp_header = recPacket[20:28]
51         msg_type, code, checksum, p_id, sequence =
52             struct.unpack('bbHHh', icmp_header)
53         time_remaining -= time_waiting
54         time_sent = struct.unpack("d",
55                                   recPacket[28:28+struct.calcsize("d")])[0]
56         time_taken = time_recieved - time_sent
57         if p_id == ID:
58             addresses.append((str(addr[0]), float(time_taken),

```

```

        int(ip_ttl)))
50     elif time_remaining <= 0:
51         break
52     else:
53         continue
54     return addresses
55
56
57 with closing(socket.socket(socket.AF_INET, socket.SOCK_RAW,
    socket.IPPROTO_ICMP)) as ping_sock:
58     addresses = ip_utils.ip_range("192.168.1.0/24")
59     local_ip = ip_utils.get_local_ip()
60     if addresses is not None:
61         addresses_to_scan = filter(lambda x: x!=local_ip, addresses)
62     else:
63         print("error with ip range specification")
64         exit()
65     p = Pool(1)
66     ID = getpid() & 0xFFFF
67     replied = p.apply_async(recieved_ping_from_addresses, (ID, 2))
68     for address in zip(addresses_to_scan, repeat(1)):
69         try:
70             packet = ip_utils.make_icmp_packet(ID)
71             ping_sock.sendto(packet, address)
72         except PermissionError:
73             pass
74     p.close()
75     p.join()
76     hosts_up = replied.get()
77     print("\n".join(map(lambda x: f"host: [{x[0]}}\tresponded to an ICMP
        ECHO REQUEST in {round_significant_figures(x[1], 2):<10}
        seconds, ttl: [{x[2]}}", hosts_up)))

```

for responses for a set amount of time and then start sending the packets in a different process before waiting for the listening process to get all the responses back and collecting the results from that process.

1.9 Acceptable Limitations

Originally I had planned to include dedicated operating system detection as an option however I ran out of time having implemented version detection. However it still does Operating system detection partially as some services are linux only and while doing service and version detection especially the Common Platform Enumeration (CPE) parts of the matched service/version will contain operating system information, such as microsoft ActiveSync would indicate that the system being scanned was a windows system which is reflected in the match directive and attached CPE information:

```
match activesync m|^.\0\x01\0[^\0]\0[^\0]\0[^\0]\0[^\0]\0[^\0]\0.*  
\0\0\0$|s p/Microsoft ActiveSync/ o/Windows/ cpe:/a:microsoft:acti  
vesync/ cpe:/o:microsoft:windows/a
```

1.10 Test Strategy

I am going to use two different methods to test my program:

1. Unit testing
2. Wireshark

I am using two separate testing strategies because they are both good at different things, both of which I need to show that my project works. Firstly I am using unit testing to test some general purpose functions which are pure functions (are independent of the current state of the machine) such as `ip_range()` and other functions which I can just check the returned value against what it should be.

Wireshark is useful for the other half of the program which uses impure functions and the low level networking e.g. `make_tcp_packet()`. Wireshark makes this easy by allowing capture of all the packets going over the wire, as well as this it has a vast array of packet decoders (2231 in my install) which it can use to dissect almost any packet that would be on the network. The main benefit of wireshark is that I can see my scanners sending packets and then check whether the parsers that I have written for the different protocols are working. I can also check that the checksums in each of the various protocols is valid as wireshark does checksum verification for various protocols.

2 Design

2.1 Overall System Design (High Level Overview)

There are two types of scanning implemented for different scan types in my program.

- `Connect()`
- version
- listener / sender

`Connect()` scanning is the simplest in that it takes in a list of ports and simply calls the `socket.connect()` method on it and sees whether it can connect or not and the ports are marked accordingly as open or closed.

Version scanning is very similar to `Connect()` scanning in that it takes in a list of ports and connects to them, except it then sends a probe to the target to elicit a response and gain some information about the service running behind the port.

Listener / sender scanning does exactly what it says on the tin: it sets up a “listener” in another process to listen for responses from the host which the “sender” is sending packets to. It can then differentiate between open, open|filtered, filtered and closed ports based on whether it receives a packet back and what flags (part of TCP packets are a one byte long section which store “flags” where each bit in the byte represents a different flag) are set in the received packet.

2.2 Design of User Interfaces HCI

I have designed my system to have a similar interface to the most common tool currently used: nmap this is because I believe that having a familiar interface will not only make it easier for someone who is familiar with nmap to use my tool it also makes it so that anything learnt using either tool is applicable to both which benefits everyone.

Based on this perception I have used the same option flags as nmap as well as similar help messages and an identical call signature (how the program is used on the command line). Running `./netscan.py <options> <target_spec>` is identical to `nmap <options> <target_spec>` in terms of which scan types will be run, which hosts will be scanned and which ports are scanned. Below you can see the help message generated by `./netscan.py --help`.

```
usage: netscan.py [-h] [-Pn] [-sL] [-sn] [-sS] [-sT] [-sU] [-sV] [-p PORTS]
                  [--exclude_ports EXCLUDE_PORTS]
                  target_spec
```

positional arguments:

target_spec specify what to scan, i.e. 192.168.1.0/24

optional arguments:

-h, --help	show this help message and exit
-Pn	assume hosts are up
-sL	list targets
-sn	disable port scanning
-sS	TCP SYN scan
-sT	TCP connect scan
-sU	UDP scan
-sV	version scan
-p PORTS, --ports PORTS	scan specified ports
--exclude_ports EXCLUDE_PORTS	ports to exclude from the scan

It shows clearly which are required arguments and which are optional ones, as well as what each argument actually does. It also allows some arguments to be called with either a short format e.g. -p and with a most verbose format --ports this allows the user to be clearer if they are using the tool as part of an automated script to perform scanning as it is more immediately obvious what the more verbose flags do.

2.3 System Algorithms

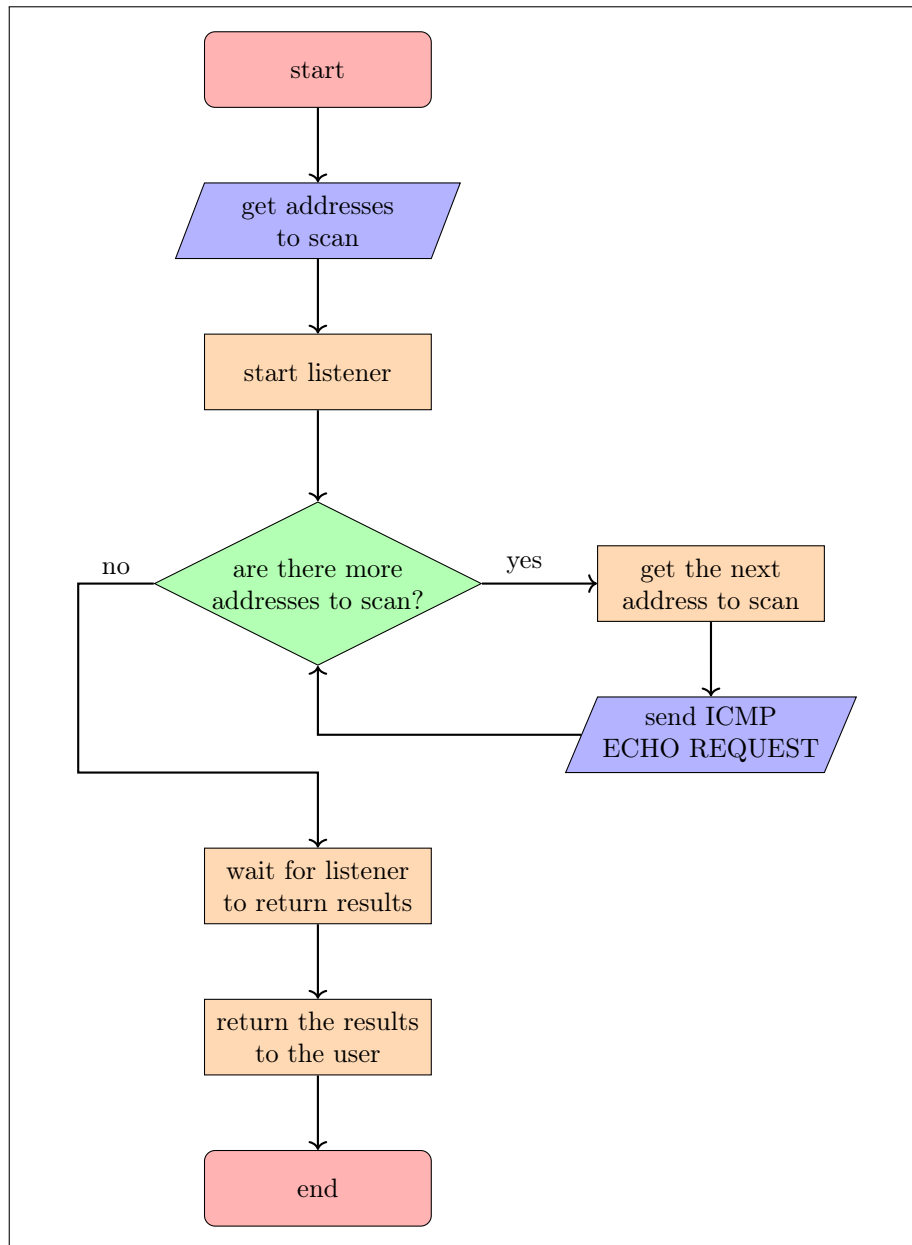


Figure 22: The logic for how I will do Ping Scanning.

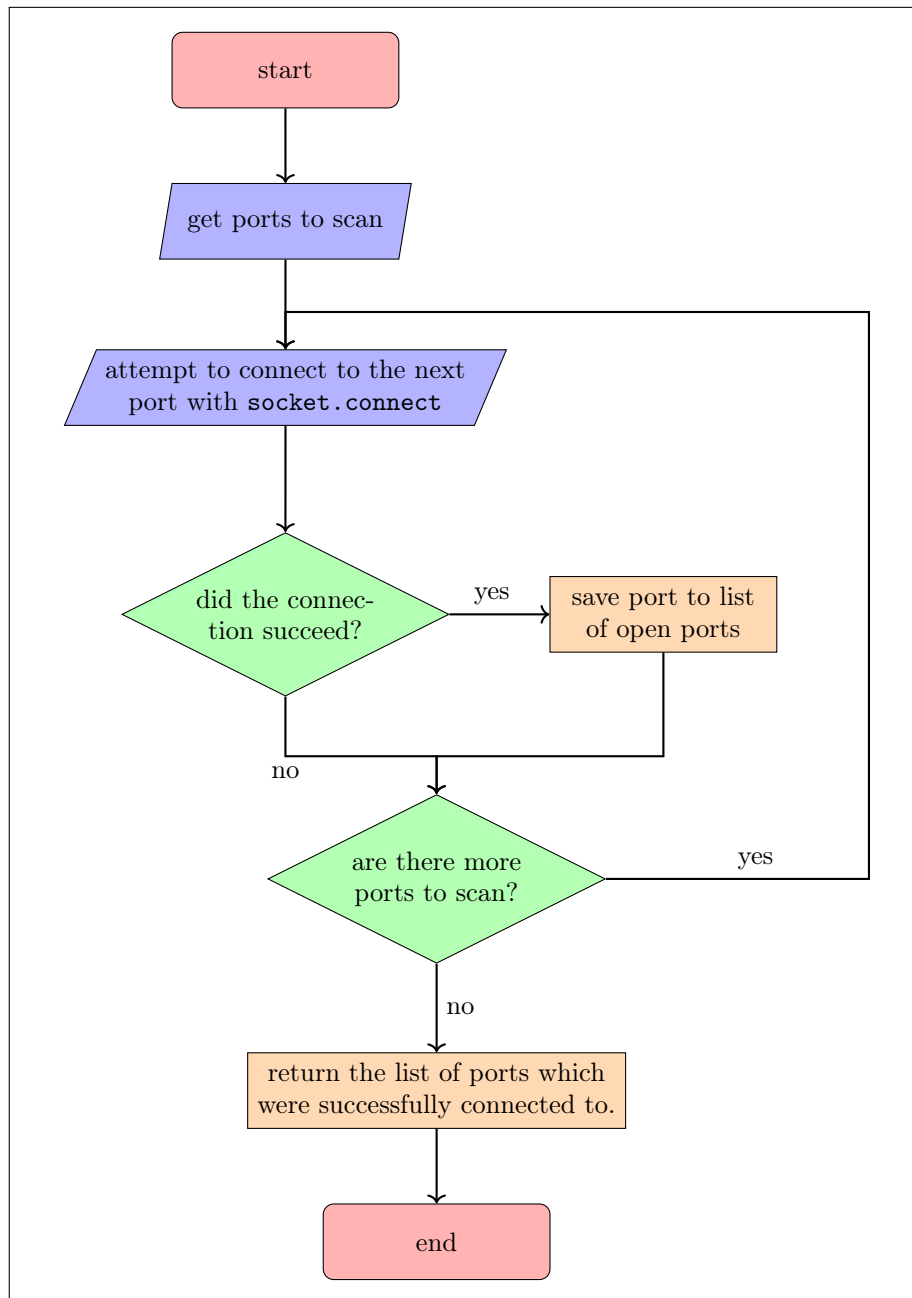


Figure 23: The logic for how I will do TCP connect Scanning.

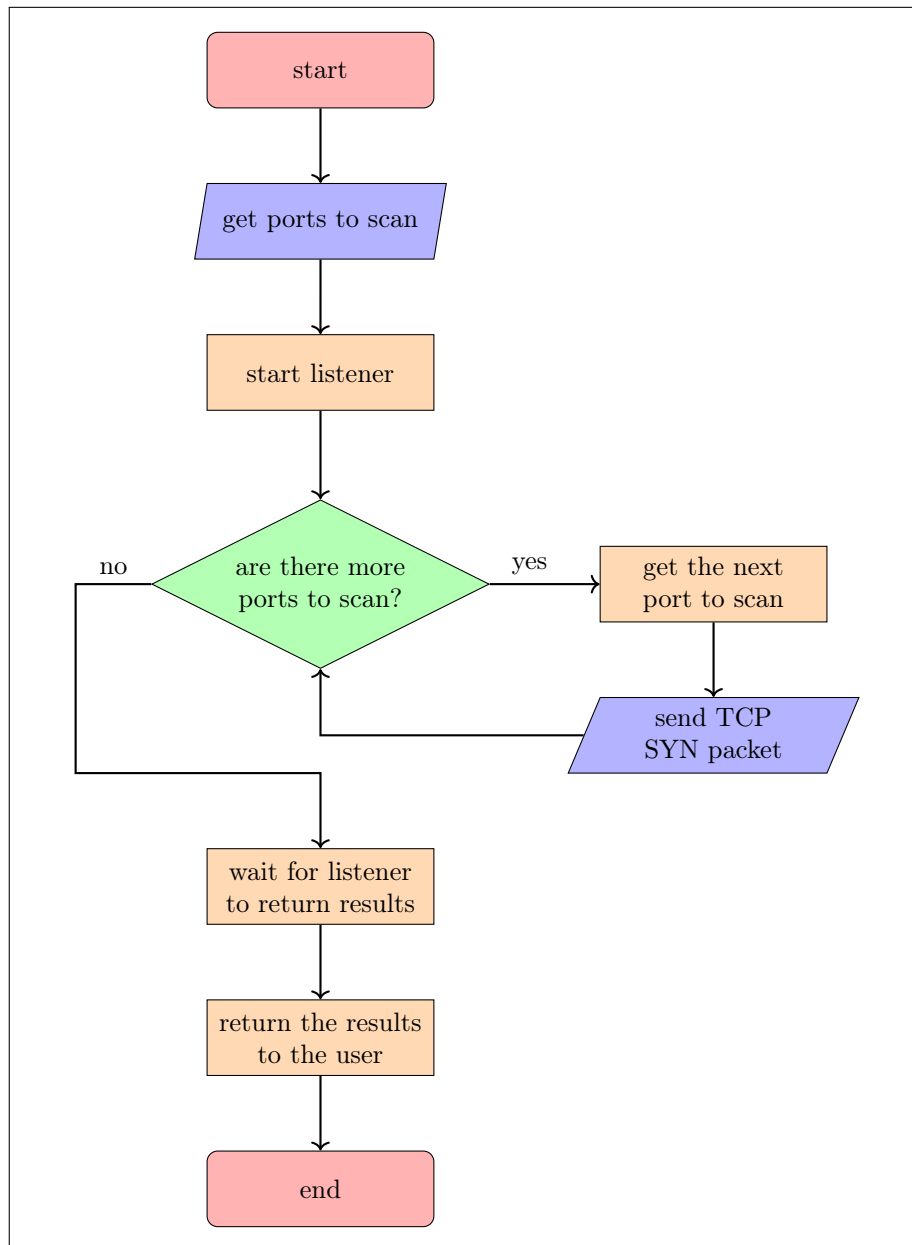


Figure 24: The logic for how I will do TCP SYN scanning.

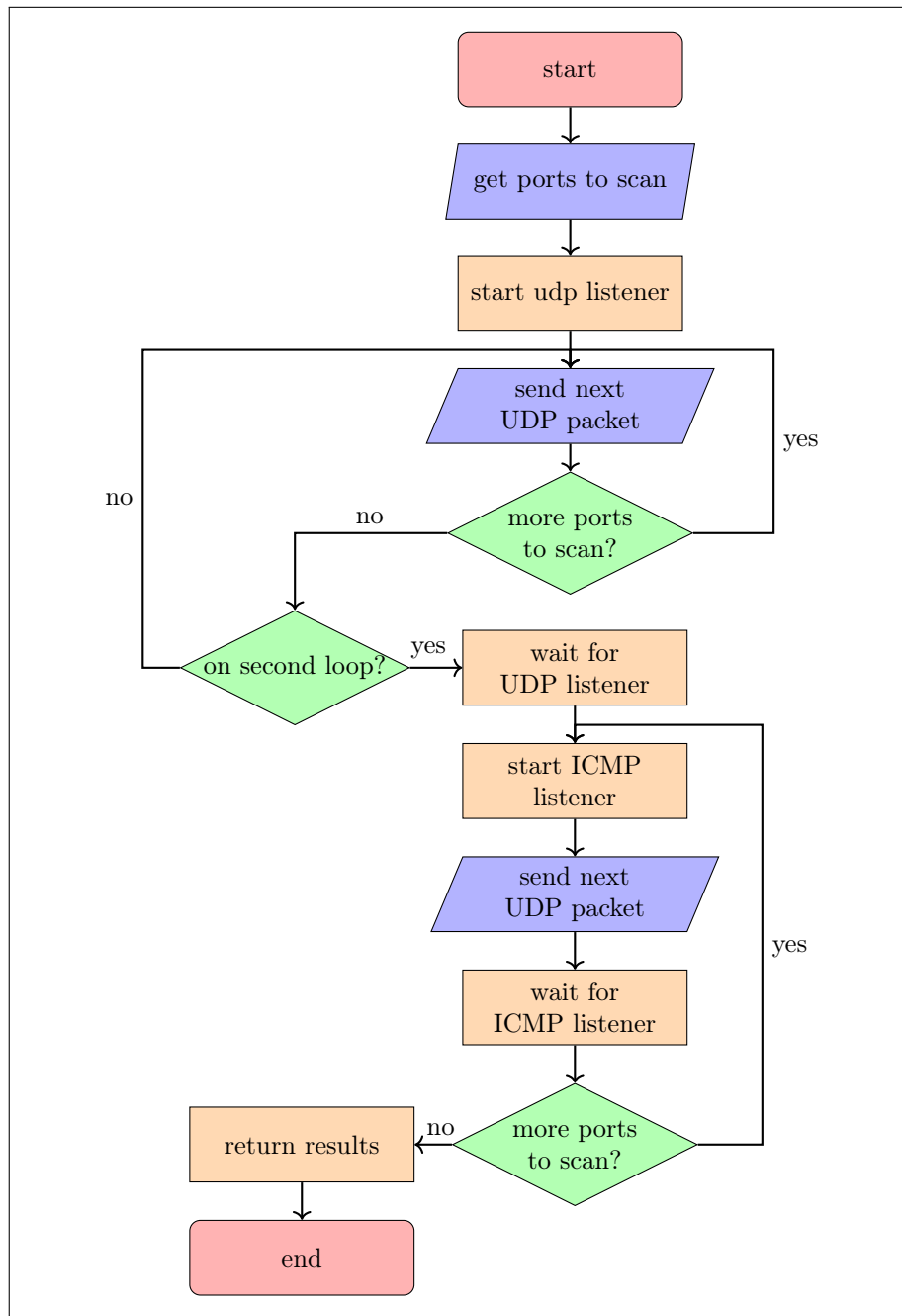


Figure 25: The logic behind how UDP scanning works.

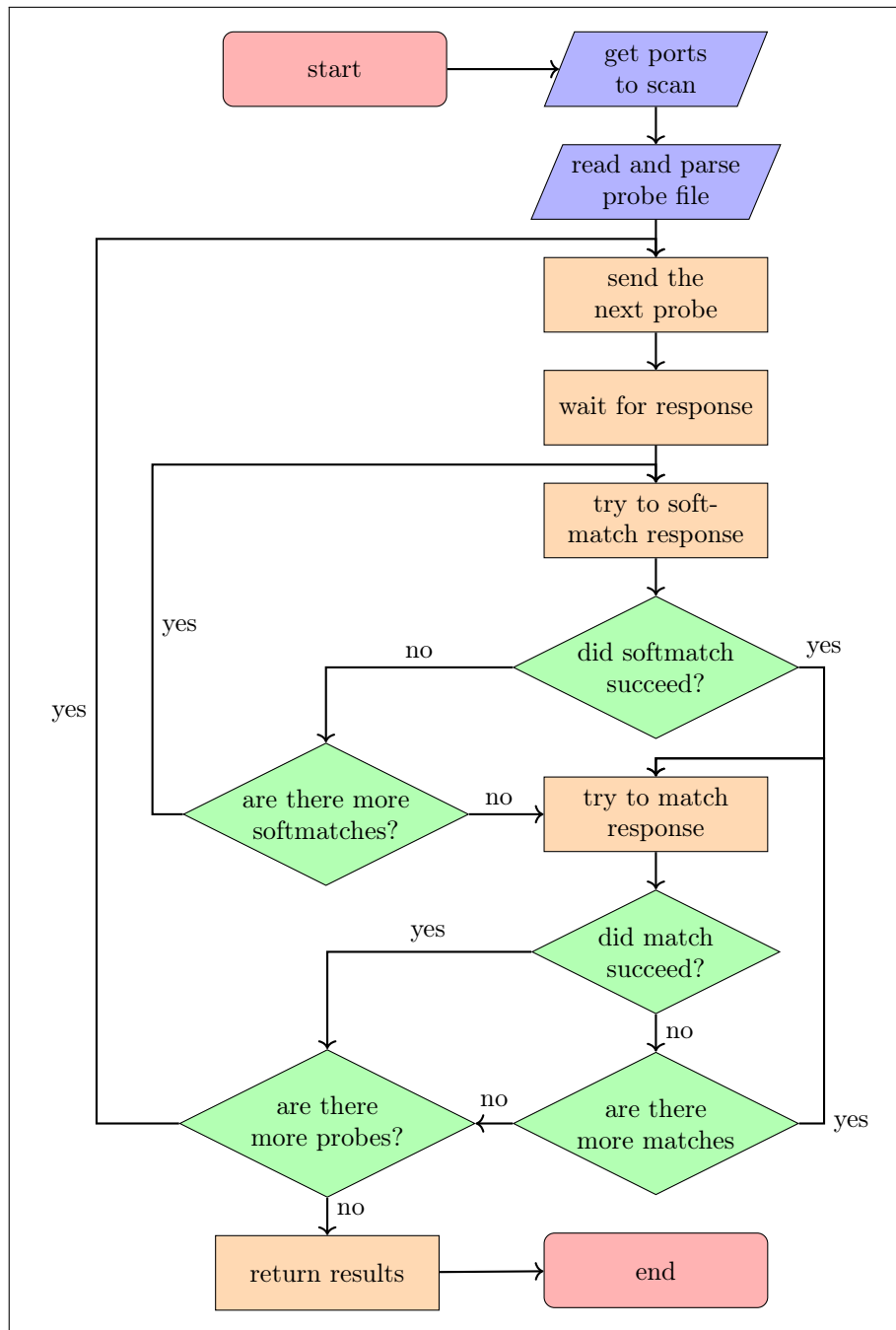


Figure 26: The logic behind how version detection works.

2.4 Input data Validation

My program takes very little input from the user which means that there is a very low chance of the program crashing due to user input error as the errors are detected. All data which is entered is either parsed using a regular expression with the case of the ports directive (-p) or is run through checking functions like `ip_utils.is_valid_ip`. As well as using these checking functions whenever an IP address is converted between “long form” and “dot form” which is used in every type of scanning.

2.5 Proposed Algorithms for complex structures (flow charts or Pseudo Code)

Algorithm 1 My algorithm for turning a CIDR specified subnet into a list of actual IP addresses

```
1: procedure IP_RANGE
2:   network_bits  $\leftarrow$  number of network bits specified
3:   ip  $\leftarrow$  base IP address
4:   mask  $\leftarrow$  0
5:   for maskbit  $\leftarrow$  (32 - network_bits), 31 do
6:     mask  $\leftarrow$  mask +  $2^{\text{maskbit}}$ 
7:   lower_bound  $\leftarrow$  ip AND mask  $\triangleright$  zero the last 32-network_bits
8:   upper_bound  $\leftarrow$  ip OR (mask XOR 0xFFFFFFFF)  $\triangleright$  turn the last
   32-network_bits to ones
9:   addresses  $\leftarrow$  empty list
10:  for address  $\leftarrow$  lower_bound, upper_bound do
11:    append CONVERT_TO_DOT(address) to addresses
  return addresses
```

Algorithm 2 My algorithm for pretty-printing a dictionary of lists of portnumbers such that ranges are specified as start-end instead of start,start+1,...,end

```

1: procedure COLLAPSE
2:   port_dictionary  $\leftarrow$  dictionary of lists of portnumbers
3:   key_results  $\leftarrow$  empty list  $\triangleright$  stores the formatted result for each key
4:   for key in port_dictionary do
5:     ports  $\leftarrow$  port_dict[key]
6:     result  $\leftarrow$  key + "{"
7:     if ports is empty then
8:       new_sequence  $\leftarrow$  FALSE
9:       for index  $\leftarrow$  1, (length of ports) - 1 do
10:        port = ports[index]
11:        if index = 0 then
12:          result  $\leftarrow$  result + ports[0]  $\triangleright$  append the first element
13:          if ports[index+1] = port + 1 then
14:            result  $\leftarrow$  result + "-"  $\triangleright$  begin a new sequence
15:          else
16:            result  $\leftarrow$  result + ","  $\triangleright$  not a sequence
17:          else if port + 1  $\neq$  ports[index+1] then  $\triangleright$  break in sequence
18:            result  $\leftarrow$  result + port + ","
19:            new_sequence  $\leftarrow$  TRUE
20:          else if port + 1 = ports[index+1] & new_sequence then
21:            result  $\leftarrow$  result + "-"
22:            new_sequence  $\leftarrow$  FALSE
23:          result  $\leftarrow$  result + ports[(length of ports)-1] + "}"
24:          append result to key_results
25:   return "{" + (key_results separated by ", ") + "}"

```

3 Technical Solution

3.1 Overview to direct the examiner to areas of complexity and explain design evidence

need to do this.

4 Testing

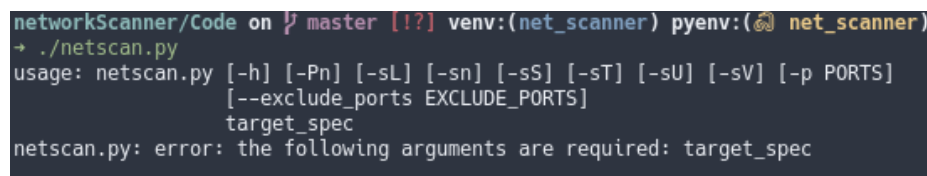
4.1 Test Plan

I will be testing my application using a combination of unit tests and Wireshark where applicable. Unit tests are more suitable to doing tests on specific functions to make sure that regressions don't occur while developing the application. A regression is when a feature or change that was implemented into the program is by accident and would cause the application to break. Wireshark I will use to show the scanning portion of my code and where external connections are made/custom packets created.

4.2 Test Table / Testing Evidence

4.2.1 Printing a usage message when run without parameters

To show this I will run my program passing it no parameters. This should print out a message of the form: `USAGE: ./<program> <required> <parameters>` where everything in angle brackets should be replaced by what is necessary for my program. In figure 27 you can see me run `./netscan.py` with no parameters and it prints out the required usage message telling me that I am missing the `target_spec` parameter, this shows that it passed this test. This shows success criteria 2.



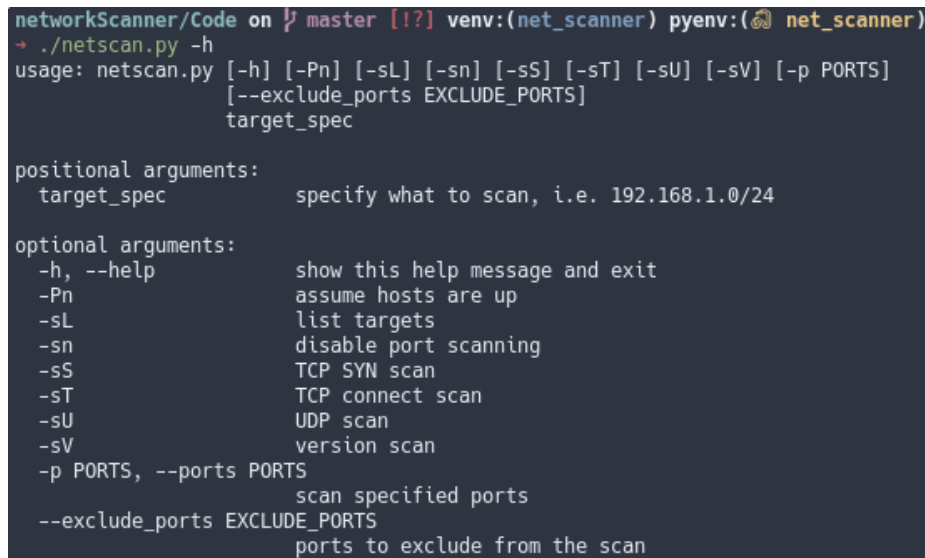
```
networkScanner/Code on master [!?] venv:(net_scanner) pyenv:(net_scanner)
➔ ./netscan.py
usage: netscan.py [-h] [-Pn] [-sL] [-sn] [-sS] [-sT] [-sU] [-sV] [-p PORTS]
                  [--exclude_ports EXCLUDE_PORTS]
                  target_spec
netscan.py: error: the following arguments are required: target_spec
```

Figure 27: Screenshot showing my program being run without parameters.

4.2.2 Printing a help message when passed -h

To show this I will run my program with the `-h` flag. This should print out a message showing each of the options as well as what each of them do. It should also print out whether they are positional arguments or optional arguments and if an argument can have two forms then it should print out both forms of the

flag, i.e. `-p --ports`. In figure 28 you can see me run my program with the `-h` flag and it proceeds to print of a help message with messages with what each option is for as well as short and long form of arguments, this shows my program passed this test. This shows success criteria 2.



```
networkScanner/Code on master [!?] venv:(net_scanner) pyenv:(net_scanner)
→ ./netscan.py -h
usage: netscan.py [-h] [-Pn] [-sL] [-sn] [-sS] [-sT] [-sU] [-sV] [-p PORTS]
                  [--exclude_ports EXCLUDE_PORTS]
                  target_spec

positional arguments:
  target_spec            specify what to scan, i.e. 192.168.1.0/24

optional arguments:
  -h, --help            show this help message and exit
  -Pn                  assume hosts are up
  -sL                  list targets
  -sn                  disable port scanning
  -sS                  TCP SYN scan
  -sT                  TCP connect scan
  -sU                  UDP scan
  -sV                  version scan
  -p PORTS, --ports PORTS
                        scan specified ports
  --exclude_ports EXCLUDE_PORTS
                        ports to exclude from the scan
```

Figure 28: Screenshot showing my program being run with the `-h` flag.

4.2.3 Printing a help message when passed `-help`

To show this I will run my program with the `--help` flag. This should produce the exact same output as with `-h`. This shows the exact same message as in the test of `-h`. To prove this if I take the shasum of the output for both flags we can see that the hashes are identical and therefore the originals were also identical, this is shown in figure 30. This shows success criteria 2.

```
networkScanner/Code on 🐧 master [!?] venv:(net_scanner) pyenv:(🐍 net_scanner)
→ ./netscan.py --help
usage: netscan.py [-h] [-Pn] [-sL] [-sn] [-sS] [-sT] [-sU] [-sV] [-p PORTS]
                  [--exclude_ports EXCLUDE_PORTS]
                  target_spec

positional arguments:
  target_spec            specify what to scan, i.e. 192.168.1.0/24

optional arguments:
  -h, --help            show this help message and exit
  -Pn                  assume hosts are up
  -sL                  list targets
  -sn                  disable port scanning
  -sS                  TCP SYN scan
  -sT                  TCP connect scan
  -sU                  UDP scan
  -sV                  version scan
  -p PORTS, --ports PORTS
                        scan specified ports
  --exclude_ports EXCLUDE_PORTS
                        ports to exclude from the scan
```

Figure 29: Screenshot showing my program being run with the help flag.

```
networkScanner/Code on 🐧 master [!?] venv:(net_scanner) pyenv:(🐍 net_scanner)
→ ./netscan.py --help | shasum
47b52bb86cbe927e2c1a9521e4c55c2369f541a5 -

networkScanner/Code on 🐧 master [!?] venv:(net_scanner) pyenv:(🐍 net_scanner)
→ ./netscan.py -h | shasum
47b52bb86cbe927e2c1a9521e4c55c2369f541a5 -
```

Figure 30: Screenshot showing the hashes of the two help messages.

4.2.4 Translating a CIDR specified subnet into a list of IP addresses

To show this I will run my program with the `-sL` flag and I will specify a small subnet of `192.168.1.0/28` (I have chosen such a small subnet such that it will fit on my terminal and therefore in a screenshot). I expect the list of addresses to be `192.168.1.1 - 192.168.1.14`. To prove that my program works I will screenshot the output when run with the stated parameters and I will use a website to translate the same subnet and show that it displays the same addresses as my program. In figure 31 you can see that the output from my program matches the expected list of IP addresses from `192.168.1.1` to `192.168.1.14` which is also shown by the screen shot of the same subnet translated by the `ipcalc` utility on linux. This proves my program works and covers success criteria 4.

```

networkScanner/Code on master [x1?] venv:(net_scanner) pyenv:(net_scanner)
→ ./netscan.py -sL 192.168.1.0/28
Targets:
192.168.1.1
192.168.1.2
192.168.1.3
192.168.1.4
192.168.1.5
192.168.1.6
192.168.1.7
192.168.1.8
192.168.1.9
192.168.1.10
192.168.1.11
192.168.1.12
192.168.1.13
192.168.1.14

```

Figure 31: Screenshot showing the output of my program when asked to translate the subnet 192.168.1.0/28.

```

networkScanner/Code on master [!?] venv:(net_scanner) pyenv:(net_scanner)
→ ipcalc 192.168.1.0/28
Address: 192.168.1.0      11000000.10101000.00000001.0000 0000
Netmask: 255.255.255.240 = 28 11111111.11111111.11111111.1111 0000
Wildcard: 0.0.0.15      00000000.00000000.00000000.0000 1111
=>
Network: 192.168.1.0/28  11000000.10101000.00000001.0000 0000
HostMin: 192.168.1.1    11000000.10101000.00000001.0000 0001
HostMax: 192.168.1.14   11000000.10101000.00000001.0000 1110
Broadcast: 192.168.1.15 11000000.10101000.00000001.0000 1111
Hosts/Net: 14           Class C, Private Internet

```

Figure 32: Screenshot showing the range displayed by the ipcalc utility when asked to calculate the same subnet.

4.2.5 Scanning a subnet with ICMP ECHO REQUEST messages

To show this I will run my program with the `-sn` flag and specify the subnet of my local network 192.168.178.0/24. This should produce a list of all the hosts which are up on the network. In figure 33 you can see you can see my program's output showing that the hosts:

- 192.168.178.60
- 192.168.178.56
- 192.168.178.30
- 192.168.178.1

all responded with ICMP ECHO REPLY messages, this is reflected in a packet capture I took while performing the scan. A section of this scan is shown in figure 34 where you can see some of ICMP ECHO REQUEST messages my program sent, along with some of the requests to hosts that don't exist, note the different addresses in the source and destination fields and the Echo (ping) request vs reply in the info column. This successfully shows success criteria 1 and 3.

```
networkScanner/Code on master [!?] venv:(net_scanner) pyenv:(net_scanner) took 13s
→ sudo ./netscan.py 192.168.178.0/24 -sn
host: [192.168.178.60] responded to an ICMP ECHO REQUEST in 0.00011s  ttl: [64]
host: [192.168.178.56] responded to an ICMP ECHO REQUEST in 0.28s  ttl: [64]
host: [192.168.178.30] responded to an ICMP ECHO REQUEST in 0.027s  ttl: [64]
host: [192.168.178.1] responded to an ICMP ECHO REQUEST in 0.031s  ttl: [64]
```

Figure 33: Screenshot showing the output of a scan of my local network.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	192.168.178.60	192.168.178.30	ICMP	234	Echo (ping) request
2	0.000749915	192.168.178.60	192.168.178.56	ICMP	234	Echo (ping) request
3	0.004504662	192.168.178.60	192.168.178.20	ICMP	234	Echo (ping) request
4	0.004830456	192.168.178.60	192.168.178.48	ICMP	234	Echo (ping) request
5	0.005289695	192.168.178.60	192.168.178.1	ICMP	234	Echo (ping) request
6	0.026946346	192.168.178.30	192.168.178.60	ICMP	234	Echo (ping) reply
7	0.036125893	192.168.178.1	192.168.178.60	ICMP	234	Echo (ping) reply
8	0.281829344	192.168.178.56	192.168.178.60	ICMP	234	Echo (ping) reply
9	0.282171289	192.168.178.60	192.168.178.51	ICMP	234	Echo (ping) request
10	2.329937472	192.168.178.60	192.168.178.21	ICMP	234	Echo (ping) request
11	2.330018351	192.168.178.60	192.168.178.35	ICMP	234	Echo (ping) request

Figure 34: Screenshot showing a selection of the packets being sent by this scan.

4.2.6 Scanning without first checking whether hosts are up.

To show this I will perform a TCP scan on a small subnet where I know there are no hosts and show that the scan continues despite there actually being no host on the other end. To do this I will pass the `-Pn` flag and I will specify the subnet `192.168.43.0/28` which I know has no hosts on it. I will also specify `-p 12345` to only scan port 12345 so that there are fewer requests in the packet capture. Finally I will specify `-sS` to do TCP SYN SCANNING. I expect to see a multiple of 14 Address Resolution Protocol (ARP) messages. This is because I don't know how many times my NIC will retry at getting the destination Media Access Control (MAC) address. It needs to destination MAC address to send the packet to its destination as we are scanning a private IP range of my router. In figure 35 you can see the output of my program when run with the specified flags, you can see that as expected it showed that there were no open ports on those machines as they don't exist. In figure 36 you can see the packet capture of the packets my code sent, however there are only ARP

messages, this is because we are scanning in the private IP range of my router which was the only way I could guarantee that there was no machine at the other end. However this is as expected, as well as this we can see 42 ARP requests, which is 3×14 ARP requests, which would indicate each scan made three ARP requests before giving up. This shows my program can perform scans without first checking if the host is up, showing success criteria 5.

```
networkScanner/Code on master [X1?] venv:(net_scanner) pyenv:(net_scanner)
→ sudo ./netscan.py -Pn -p 12345 192.168.43.0/28 -sS

Scan report for: 192.168.43.11
Open ports:

Scan report for: 192.168.43.5
Open ports:

Scan report for: 192.168.43.6
Open ports:

Scan report for: 192.168.43.7
Open ports:

Scan report for: 192.168.43.13
Open ports:

Scan report for: 192.168.43.8
Open ports:

Scan report for: 192.168.43.9
Open ports:

Scan report for: 192.168.43.2
Open ports:

Scan report for: 192.168.43.14
Open ports:

Scan report for: 192.168.43.3
Open ports:

Scan report for: 192.168.43.4
Open ports:

Scan report for: 192.168.43.12
Open ports:

Scan report for: 192.168.43.10
Open ports:

Scan report for: 192.168.43.1
Open ports:
```

Figure 35: Screenshot showing the output from my code when asked to port scan a subnet with no machines behind the addresses.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.11? Tell 192.168.43.182
2	1.011109141	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.11? Tell 192.168.43.182
3	2.024200112	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.11? Tell 192.168.43.182
4	5.041957747	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.5? Tell 192.168.43.182
5	6.051083685	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.5? Tell 192.168.43.182
6	7.064357935	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.5? Tell 192.168.43.182
7	10.084811460	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.6? Tell 192.168.43.182
8	11.090830088	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.6? Tell 192.168.43.182
9	12.104434950	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.6? Tell 192.168.43.182
10	15.127316464	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.7? Tell 192.168.43.182
11	16.134440557	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.7? Tell 192.168.43.182
12	17.144156881	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.7? Tell 192.168.43.182
13	20.185685090	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.13? Tell 192.168.43.182
14	21.197765175	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.13? Tell 192.168.43.182
15	22.211087805	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.13? Tell 192.168.43.182
16	25.231530175	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.8? Tell 192.168.43.182
17	26.237740239	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.8? Tell 192.168.43.182
18	27.251103712	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.8? Tell 192.168.43.182
19	30.261899876	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.9? Tell 192.168.43.182
20	31.277469168	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.9? Tell 192.168.43.182
21	32.290783603	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.9? Tell 192.168.43.182
22	35.291040729	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.2? Tell 192.168.43.182
23	36.317480038	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.2? Tell 192.168.43.182
24	37.330771296	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.2? Tell 192.168.43.182
25	40.307612623	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.14? Tell 192.168.43.182
26	41.330762593	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.14? Tell 192.168.43.182
27	42.344096055	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.14? Tell 192.168.43.182
28	45.339384199	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.3? Tell 192.168.43.182
29	46.344416562	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.3? Tell 192.168.43.182
30	47.357528471	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.3? Tell 192.168.43.182
31	50.399259067	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.4? Tell 192.168.43.182
32	51.410810223	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.4? Tell 192.168.43.182
33	52.424096052	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.4? Tell 192.168.43.182
34	55.449381914	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.12? Tell 192.168.43.182
35	56.450760889	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.12? Tell 192.168.43.182
36	57.464250695	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.12? Tell 192.168.43.182
37	60.471503134	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.10? Tell 192.168.43.182
38	61.490761449	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.10? Tell 192.168.43.182
39	62.504143757	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.10? Tell 192.168.43.182
40	65.501665262	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.1? Tell 192.168.43.182
41	66.504417252	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.1? Tell 192.168.43.182
42	67.517717037	IntelCor_9e:29:dd		ARP	44	Who has 192.168.43.1? Tell 192.168.43.182

Figure 36: Screenshot showing the ARP requests my NIC sent to attempt to determine where to send the attempted connection packets.

4.2.7 Detecting whether a TCP port is open

To show this I will perform a TCP Connect() scan on my local machine while running a script which will listen on port 12345 for any connections and send back a message. To do this I will pass my program the flags `-sT` and `-p 12345` as well as specifying localhost to scan (127.0.0.1). I expect to see a TCP SYN-ACK handshake between my program and the script and then my program to output that the port is open. In figure 39 you can see the expected TCP SYN-ACK handshake performed by my program and the script in figure 37. You can see the output of my program in figure 38, as expected it outputs that port 12345 is open. This shows success criteria 1 and 6.

```

In [1]: import socket

In [2]: target = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

In [3]: target.bind(("127.0.0.1", 12345))

In [4]: target.listen()

In [5]: conn, addr = target.accept()

In [6]: addr
Out[6]: ('127.0.0.1', 53808)

```

Figure 37: Screenshot showing the script I ran to accept a connection on local-host port 12345.

```

networkScanner/Code on master [x1?] venv:(net_scanner) pyenv:(net_scanner) took 9s
→ sudo ./netscan.py 127.0.0.1 -p 12345 -sT

Scan report for: 127.0.0.1
Open ports:
12345 service: netbus?

```

Figure 38: Screenshot showing the output of my script when run with the specified flags and while the script in figure 37 was running.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	53848 → 12345 [SYN] Seq=
2	0.000055204	127.0.0.1	127.0.0.1	TCP	74	12345 → 53848 [SYN, ACK]
3	0.000091877	127.0.0.1	127.0.0.1	TCP	66	53848 → 12345 [ACK] Seq=
4	0.000128597	127.0.0.1	127.0.0.1	TCP	66	53848 → 12345 [FIN, ACK]
5	0.016769292	127.0.0.1	127.0.0.1	TCP	66	12345 → 53848 [ACK] Seq=

Figure 39: Screenshot showing the packet capture of the TCP SYN-ACK handshake performed by the scan in figure 38 with the script in 37.

4.2.8 Detecting whether a TCP port is closed

To show this I will perform a TCP `Connect()` scan on my local machine except instead of running a script to catch the request I will just let it try to connect to the closed port. I expect to see a TCP SYN packet sent to the port and then a RST, ACK packet sent back, my program should output no open ports. To do this I will pass my program the same options as in the test for a TCP open port. In figure 41 you can see the attempted connection to 127.0.0.1 port 12345 along with the RST, ACK packet afterwards indicating the port is closed. This is reflected in figure 40 with no open ports showing success criteria 1 and 7.

```

networkScanner/Code on master [x!?] venv:(net_scanner) pyenv:(net_scanner) took 9s
→ sudo ./netscan.py 127.0.0.1 -p 12345 -sT
[sudo] password for tritoke:

Scan report for: 127.0.0.1
Open ports:

```

Figure 40: Screenshot showing the output of my program when run with the specified options.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	74	53892 → 12345 [SYN] Seq=
2	0.0000006554	127.0.0.1	127.0.0.1	TCP	54	12345 → 53892 [RST, ACK]

Figure 41: Screenshot showing the packet capture of the TCP SYN-RST closed port indication caused by the scan in figure 40.

4.2.9 Detecting whether a TCP port is filtered

To show this I will perform a TCP SYN scan on localhost port 12345 except I will also introduce a firewall rule to drop all requests to localhost. I expect this to produce no response to the initial SYN packet sent by my program and my program to output that port as filtered. To test this I will run my program with the flags `-sS, -p 12345, -Pn` this will cause it to not check whether the host is up, to perform a TCP SYN scan and only scan port 12345. I will also introduce a firewall rule using the linux iptables utility to drop all requests to localhost as so: `iptables -I INPUT -s 127.0.0.1 -j DROP`. The output of my program is shown in figure 42 you can see that port 12345 is displayed as filtered and in the packet capture shown in figure 43 you can see that there is no response to our initial packet which corresponds to what I thought would happen with an iptables rule in place to drop packets. This shows success criteria 1 and 8.

```

networkScanner/Code on master [x!?] venv:(net_scanner) pyenv:(net_scanner) took 4s
→ sudo ./netscan.py 127.0.0.1 -p 12345 -sS -Pn

Scan report for: 127.0.0.1
Open ports:
Filtered ports:
12345 service: netbus?

```

Figure 42: Screenshot showing the output of my program when run with the specified options and a firewall in place to drop all packets to 127.0.0.1.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	TCP	58	38337 → 12345 [SYN]

Figure 43: Screenshot showing the packet capture of the scan in figure 42

4.2.10 Detecting whether a UDP port is open

To show this I will perform a UDP scan on a script I have already written while developing UDP scanning which can be seen in listing 8. I expect to see my program output port 12345 as open and in the packet capture I expect to see two UDP packets followed by two response UDP packets from my listener program. I will test this using the following flags: `-Pn, -p 12345, -sU` these translate to scanning port 12345 over UDP and not checking the host is up beforehand. In figure 44 you can see the output of my program when run as specified and you can see that it correctly detects port 12345 as being open. In figure 45 you can see the packet capture of my program being run however this is not as I expected, I didn't foresee the ICMP destination unreachable messages, these are sent by the kernel in response to the UDP probe which it doesn't know what to do with, however apart from those the capture shows everything as expected. This shows success criteria 1 and 9.

Listing 8: Script to open port 12345 to UDP.

```

1 import socket
2 from contextlib import closing
3
4 with closing(
5     socket.socket(
6         socket.AF_INET,
7         socket.SOCK_DGRAM
8     )
9 ) as s:
10     s.bind(("127.0.0.1", 12345))
11     print("opened port 12345 on localhost")
12     while True:
13         data, addr = s.recvfrom(1024)
14         s.sendto(bytes("Well hello there good sir.", "utf-8"), addr)

```

```

networkScanner/Code on master [X1?] venv:(net_scanner) pyenv:(net_scanner)
→ sudo ./netscan.py 127.0.0.1 -p 12345 -sU -Pn

Scan report for: 127.0.0.1
Open ports:
12345 service: italk?
Filtered ports:

```

Figure 44: Screenshot showing the output of my program when run with the options specified above, and the script in listing 8 is running.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	92	58233 → 12345 Len=50
2	0.000018274	127.0.0.1	127.0.0.1	UDP	92	58233 → 12345 Len=50
3	0.000101924	127.0.0.1	127.0.0.1	UDP	68	12345 → 58233 Len=26 [UDP CHECKSUM INCORRECT]
4	0.000109606	127.0.0.1	127.0.0.1	ICMP	96	Destination unreachable (Port unreachable)
5	0.000121998	127.0.0.1	127.0.0.1	UDP	68	12345 → 58233 Len=26 [UDP CHECKSUM INCORRECT]
6	0.000124894	127.0.0.1	127.0.0.1	ICMP	96	Destination unreachable (Port unreachable)

Figure 45: screenshot showing the packet capture of the scan in figure 44

4.2.11 Detecting whether a UDP port is closed

To show this I will perform a UDP scan on a port which has no service listening behind it. I expect my program to print out no filtered ports and no open ports showing that the port was closed. In the packet capture I expect to see three UDP packets and three response ICMP packets. To test this I will use my program with the following flags: `-p 12345, -Pn, -sU` which perform a UDP port scan without first checking if the host is up. In figure 46 you can see the output of my program when run with the options specified above, you can see that there are no ports displayed as either open or filtered, this shows the my program successfully marked the port as closed. This shows success criteria 1 and 10.

```
networkScanner/Code on master [x!?] venv:(net_scanner) pyenv:(net_scanner) took 8s
→ sudo ./netscan.py 127.0.0.1 -p 12345 -sU -Pn

Scan report for: 127.0.0.1
Open ports:
Filtered ports:
```

Figure 46: screenshot showing the output of my program when scanning with the options specified above.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	92	50615 → 12345 Len=50
2	0.000014482	127.0.0.1	127.0.0.1	ICMP	120	Destination unreachable (Port unreachable)
3	0.000024645	127.0.0.1	127.0.0.1	UDP	92	50615 → 12345 Len=50
4	0.000027543	127.0.0.1	127.0.0.1	ICMP	120	Destination unreachable (Port unreachable)
5	4.028510366	127.0.0.1	127.0.0.1	UDP	92	50615 → 12345 Len=50
6	4.028548735	127.0.0.1	127.0.0.1	ICMP	120	Destination unreachable (Port unreachable)

Figure 47: screenshot showing the packet capture of the scan in figure 46

4.2.12 Detecting whether a UDP port is filtered

To show this I will use my program to perform a UDP scan on my local machine with a firewall rule to drop any ports sent to the localhost address. I expect to see my program to output the port as filtered and in the packet capture I expect to see three UDP packets with no response to any of them. In figure 48 you can see my program correctly identifies the port as being filtered and in figure 49

you can see the packet capture of the scan which also as expected shows the three UDP packets with no reply packets. This shows success criteria 1 and 11.

```
networkScanner/Code on master [x!?] venv:(net_scanner) pyenv:(net_scanner) took 3s
→ sudo ./netscan.py 127.0.0.1 -p 12345 -sU -Pn

Scan report for: 127.0.0.1
Open ports:
Filtered ports:
12345 service: italk?
```


Figure 48: screenshot showing the output of my program when scanning with the options specified above.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	92	41279 → 12345 Len=50
2	0.000008961	127.0.0.1	127.0.0.1	UDP	92	41279 → 12345 Len=50
3	4.026639713	127.0.0.1	127.0.0.1	UDP	92	41279 → 12345 Len=50

Figure 49: screenshot showing the packet capture of the scan in figure 48

4.2.13 Detecting the operating system of another machine

I haven't directly added this as a feature to my project partly because I didn't have time and also because it is partially achieved by version scanning in that if a particular service is detected and that service is OS dependent then you can be fairly certain that machine is running that OS. For example if a machine is open on TCP port 22 and SSH is detected to be running behind that port then they are likely to be running a linux machine. Even more likely if the scan reveals some further information such as the CPE. In figure 50 you can see a scan of my machine where I have Secure SHell (SSH) running, my program reveals that the version is 7.9 and the vendor is openshd which is a unix like operating system, this shows that my ssh version is unix based and therefore I am likely to be running on linux, which is the case. So although it is not directly a feature in a round a bout way. This partially completes success criteria 12.

```
networkScanner/Code on  master [x!?] venv:(net_scanner) pyenv:(🐍 net_scanner)
→ sudo ./netscan.py 127.0.0.1 -sV

Scan report for: 127.0.0.1
Open ports:
22/TCP      ssh
vendorproductname: OpenSSH
version: 7.9
info: protocol 2.0

CPE:
applications
vendor: openbsd
product: openssh
version: 7.9
update:
edition:
language:

Filtered ports:
```

Figure 50: screenshot showing a version scan of my local machine.

4.2.14 Detecting the service and its version running behind a port

To show this I will use my program to perform a version detection scan on my local machine while I am running SSH. I expect to see my program identify that SSH is running on TCP port 22 and that it detects it as OpenSSH version 7.9. To test this I will run my program with the `-sV` flag to indicate version detection and I will run it against the localhost address. In figure 14 you can see that my program successfully identified SSH as running on TCP port 22 as well as the expected identification of OpenSSH version 7.9 operating on protocol version 2. It also identified some CPE information such as OpenSSH coming from the openbsd distribution. This shows success criteria 1 and 14.

```
networkScanner/Code on master [x!?] venv:(net_scanner) pyenv:(net_scanner)
→ sudo ./netscan.py 127.0.0.1 -sV

Scan report for: 127.0.0.1
Open ports:
22/TCP      ssh
vendorproductname: OpenSSH
version: 7.9
info: protocol 2.0

CPE:
applications
vendor: openbsd
product: openssh
version: 7.9
update:
edition:
language:

Filtered ports:
```

Figure 51: screenshot showing a version scan of my local machine running ssh.

5 Evaluation

5.1 Reflection on final outcome

5.2 Evaluation against objectives, end user feedback

5.3 Potential improvements

6 Appendices

6.1 icmp_ping

Listing 9: A prototype program for sending ICMP ECHO REQUEST packets

```
1  #!/usr/bin/env python
2  import socket
3  import struct
4  import os
5  import time
6  from modules.ip_utils import ip_checksum
7
8
9  def main() -> None:
10     ICMP_ECHO_REQUEST = 8
11
12     # opens a raw socket for the ICMP protocol
13     ping_sock = socket.socket(
```

```

14         socket.AF_INET,
15         socket.SOCK_RAW,
16         socket.IPPROTO_ICMP
17     )
18     # allows manual IP header creation
19     # ping_sock.setsockopt(socket.SOL_IP, socket.IP_HDRINCL, 1)
20
21     ID = os.getpid() & 0xFFFF
22
23     # the two zeros are the code and the dummy checksum, the one is the
24     # sequence number
25     dummy_header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, 0, ID, 1)
26
27     data = struct.pack(
28         "d", time.time()
29     ) + bytes(
30         (192 - struct.calcsize("d")) * "A",
31         "ascii"
32     )
33     # the data to send in the packet
34     checksum = socket.htons(ip_checksum(dummy_header + data))
35     # calculates the checksum for the packet and psuedo header
36     header = struct.pack("bbHHh", ICMP_ECHO_REQUEST, 0, checksum, ID, 1)
37     # packs the packet header
38     packet = header + data
39     # concatonates the header and the data to form the final packet.
40     ping_sock.sendto(packet, ("127.0.0.1", 1))
41     # sends the packet to localhost

```

Listing 10: A prototype program for receiving ICMP ECHO REQUEST packets

```

1  #!/usr/bin/env python
2  from modules import headers
3  import socket
4  from typing import List
5
6
7  def main() -> None:
8      # socket object using an IPV4 address, using only raw socket access,
9      # ICMP protocol
10     ping_sock = socket.socket(
11         socket.AF_INET,
12         socket.SOCK_RAW,
13         socket.IPPROTO_ICMP
14     )
15
16     packets: List[bytes] = []
17
18     while len(packets) < 1:

```

```

19         recPacket, addr = ping_sock.recvfrom(1024)
20         ip = headers.ip(recPacket[:20])
21         icmp = headers.icmp(recPacket[20:28])
22
23         print(ip)
24         print()
25         print(icmp)
26         print("\n")
27
28     packets.append(recPacket)

```

6.2 ping_scanner

Listing 11: A prototype program for performing ‘ping’ scans

```

1  #!/usr/bin/env python
2  from modules import headers
3  from modules import ip_utils
4  import socket
5  import struct
6  import time
7  from contextlib import closing
8  from itertools import repeat
9  from math import log10, floor
10 from multiprocessing import Pool
11 from os import getpid
12 from typing import Set, Tuple
13
14
15 def sig_figs(x: float, n: int) -> float:
16     """
17     rounds x to n significant figures.
18     sig_figs(1234, 2) = 1200.0
19     """
20     return round(x, n - (1 + int(floor(log10(abs(x))))))
21
22
23 def ping_listener(
24     ID: int,
25     timeout: float
26 ) -> Set[Tuple[str, float, headers.ip]]:
27     """
28     Takes in a process id and a timeout and returns
29     a list of addresses which sent ICMP ECHO REPLY
30     packets with the packed id matching ID in the time given by timeout.
31     """
32     ping_sock = socket.socket(
33         socket.AF_INET,
34         socket.SOCK_RAW,

```

```

35         socket.IPPROTO_ICMP
36     )
37     # opens a raw socket for sending ICMP protocol packets
38     time_remaining = timeout
39     addresses = set()
40     while True:
41         time_waiting = ip_utils.wait_for_socket(ping_sock,
42                                                 time_remaining)
43         # time_waiting stores the time the socket took to become readable
44         # or returns minus one if it ran out of time
45
46         if time_waiting == -1:
47             break
48         time_recieved = time.time()
49         # store the time the packet was recieved
50         recPacket, addr = ping_sock.recvfrom(1024)
51         # recieve the packet
52         ip = headers.ip(recPacket[:20])
53         # unpack the IP header into its respective components
54         icmp = headers.icmp(recPacket[20:28])
55         # unpack the time from the packet.
56         time_sent = struct.unpack(
57             "d",
58             recPacket[28:28 + struct.calcsize("d")]
59         )[0]
60         # unpack the value for when the packet was sent
61         time_taken: float = time_recieved - time_sent
62         # calculate the round trip time taken for the packet
63         if icmp.id == ID:
64             # if the ping was sent from this machine then add it to the
65             # list of
66             # responses
67             ip_address, port = addr
68             addresses.add((ip_address, time_taken, ip))
69         elif time_remaining <= 0:
70             break
71         else:
72             continue
73     # return a list of all the addresses that replied to our ICMP echo
74     # request.
75     return addresses
76
77 def main() -> None:
78     with closing(
79         socket.socket(
80             socket.AF_INET,
81             socket.SOCK_RAW,
82             socket.IPPROTO_ICMP
83         )

```



```

82     ) as ping_sock:
83         ip_addresses = ["127.0.0.1"] # ip_utils.ip_range("192.168.43.0",
84                                     24)
85         # generate the range of IP addresses to scan.
86         # get the local ip address
87         addresses = [
88             ip
89             for ip in ip_addresses
90             if (
91                 not ip.endswith(".0")
92                 and not ip.endswith(".255")
93             )
94         ]
95
96         # initialise a process pool
97         p = Pool(1)
98         # get the local process id for use in creating packets.
99         ID = getpid() & 0xFFFF
100        # run the listeners.ping function asynchronously
101        replied = p.apply_async(ping_listener, (ID, 5))
102        time.sleep(0.01)
103        for address in zip(addresses, repeat(1)):
104            try:
105                packet = ip_utils.make_icmp_packet(ID)
106                ping_sock.sendto(packet, address)
107            except PermissionError:
108                ip_utils.eprint("raw sockets require root priveleges,
109                                exiting")
110                exit()
111        p.close()
112        p.join()
113        # close and join the process pool to so that all the values
114        # have been returned and the pool closed
115        hosts_up = replied.get()
116        # get the list of addresses that replied to the echo request
117        # from the
118        # listener function
119        print("\n".join(
120            f"host: [{host}]\t" +
121            "responded to an ICMP ECHO REQUEST in " +
122            f"{str(sig_figs(taken, 2))+ 's':<10s} " +
123            f"ttl: [{ip_head.time_to_live}]"
124            for host, taken, ip_head in hosts_up
125        ))

```

6.3 subnet_to_addresses

Listing 12: A program which translates a CIDR specified subnet into a list of addresses and prints them out in sorted order

```
1  #!/usr/bin/env python
2  import re
3  from modules.ip_utils import ip_range, dot_to_long
4
5
6  if __name__ == '__main__':
7      from argparse import ArgumentParser
8      parser = ArgumentParser()
9      parser.add_argument(
10         "ip_subnet",
11         help="The CIDR form ip/subnet that you wish to print" +
12             "the IP addresses specified by."
13     )
14     args = parser.parse_args()
15     CIDR_regex = re.compile(r"(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}/\d+)")
16     search = CIDR_regex.search(args.ip_subnet)
17     if search:
18         ip, network_bits = search.group(1).split("/")
19         print("\\n".join(
20             sorted(
21                 ip_range(ip, int(network_bits)),
22                 key=dot_to_long
23             )
24         ))
```

6.4 tcp_scan

6.4.1 connect_scan

Listing 13: prototype TCP Connect() scanner only attempting to detect the state of port 22

```
1  #!/usr/bin/python3
2  from contextlib import closing
3  import socket
4  LOCAL_IP = "192.168.1.159"
5  PORT = 22
6
7  address = ("127.0.0.1", 22)
8
9  with closing(
10     socket.socket(
11         socket.AF_INET,
12         socket.SOCK_STREAM
13     )
14  ) as s:
```

```

15     try:
16         s.connect(address)
17         print(f"connection on port {PORT} succeeded")
18     except ConnectionRefusedError:
19         print(f"port {PORT} is closed")

```

Listing 14: A program that performs TCP Connect() scanning

```

1  #!/usr/bin/python3
2
3  from typing import List, Set
4
5
6  def connect_scan(address: str, ports: Set[int]) -> List[int]:
7      import socket
8      from contextlib import closing
9      open_ports: List[int] = []
10     for port in ports:
11         # loop through each port in the list of ports to scan
12         try:
13             with closing(
14                 socket.socket(
15                     socket.AF_INET,
16                     socket.SOCK_STREAM
17                 )
18             ) as s:
19                 # open an IPV4 TCP socket
20                 s.connect((address, port))
21                 # attempt to connect the newly created socket to the
22                 # target
23                 # address and port
24                 open_ports.append(port)
25                 # if the connection was successful then add the port to
26                 # the
27                 # list of open ports
28             except ConnectionRefusedError:
29                 pass
30     return open_ports
31
32 def main() -> None:
33     open_ports = connect_scan("192.168.43.225", set(range(65535)))
34     print("\n".join(map(lambda x: f"port: [{x}]\tis open", open_ports)))

```

6.4.2 syn_scan

Listing 15: A prototype program that tries to detect the state of port 22 via TCP SYN scanning (aka half open scanning)

```

1  #!/usr/bin/python3.7
2  from contextlib import closing
3  import socket
4  import ip_utils
5
6  dest_port = 22
7  src_port = ip_utils.get_free_port()
8  local_ip = ip_utils.get_local_ip()
9  dest_ip = "192.168.1.159"
10 local_ip = dest_ip = "127.0.0.1"
11 loc_long = ip_utils.dot_to_long(local_ip)
12
13 SYN = 2
14 RST = 4
15
16
17
18 with closing(
19     socket.socket(
20         socket.AF_INET,
21         socket.SOCK_RAW,
22         socket.IPPROTO_TCP
23     )
24 ) as s:
25     tcp_packet = ip_utils.make_tcp_packet(
26         src_port,
27         dest_port,
28         local_ip,
29         dest_ip,
30         SYN
31     )
32     if tcp_packet is not None:
33         s.sendto(tcp_packet, (dest_ip, dest_port))
34     else:
35         print(f"Couldn't make TCP packet with supplied arguments:",
36             f"source port: [{src_port}]",
37             f"destination port: [{dest_port}]",
38             f"local ip: [{local_ip}]",
39             f"destination ip: [{dest_ip}]",
40             f"SYN flag: [{SYN}]",
41             sep="\n")

```

Listing 16: A program that performs TCP SYN scanning (aka half open scanning)

```

1  #!/usr/bin/python3.7
2  from modules import headers
3  from modules import ip_utils
4  import socket
5  from contextlib import closing

```

```

6 from multiprocessing import Pool
7 from typing import List, Set, Tuple
8
9
10 def syn_listener(address: Tuple[str, int], timeout: float) -> List[int]:
11     """
12     This function is run asynchronously and listens for
13     TCP ACK responses to the sent TCP SYN msg.
14     """
15     print(f"address: [{address}]\ntimeout: [{timeout}]")
16     open_ports: List[int] = []
17     with closing(
18         socket.socket(
19             socket.AF_INET,
20             socket.SOCK_RAW,
21             socket.IPPROTO_TCP
22         ) as s:
23         s.bind(address)
24         # bind the raw socket to the listening address
25         time_remaining = timeout
26         print("started listening")
27         while True:
28             time_taken = ip_utils.wait_for_socket(s, time_remaining)
29             # wait for the socket to become readable
30             if time_taken == -1:
31                 break
32             else:
33                 time_remaining -= time_taken
34             packet = s.recv(1024)
35             # recieve the packet data
36             tcp = headers.tcp(packet[20:40])
37             if tcp.flags == 0b00010010: # syn ack
38                 print(tcp)
39                 open_ports.append(tcp.source)
40                 # check that the header contained the TCP ACK flag and if
41                 # it
42                 # did append it
43             else:
44                 continue
45             print("finished listening")
46     return open_ports
47
48 def syn_scan(dest_ip: str, portlist: Set[int]) -> List[int]:
49     src_port = ip_utils.get_free_port()
50     # request a local port to connect from
51     local_ip = ip_utils.get_local_ip()
52     p = Pool(1)
53     listener = p.apply_async(syn_listener, ((local_ip, src_port), 5))
54     # start the TCP ACK listener in the background

```

```

55     print("starting scan")
56     for port in portlist:
57         packet = ip_utils.make_tcp_packet(src_port, port, local_ip,
58                                           dest_ip, 2)
59         # create a TCP packet with the syn flag
60         with closing(
61             socket.socket(
62                 socket.AF_INET,
63                 socket.SOCK_RAW,
64                 socket.IPPROTO_TCP
65             ) as s:
66             s.sendto(packet, (dest_ip, port))
67             # send the packet to its destination
68
69     print("finished scan")
70     p.close()
71     p.join()
72     open_ports = listener.get()
73     # collect the list of ports that responded to the TCP SYN message
74     print(open_ports)
75     return open_ports
76
77
78 def main() -> None:
79     dest_ip = "127.0.0.1"
80     syn_scan(dest_ip, set(range(2**16)))

```

6.5 udp_scan

Listing 17: A prototype program to detect whether UDP port 53 is open on a target machine

```

1  #!/usr/bin/ python
2  from contextlib import closing
3  import ip_utils
4  import socket
5
6  dest_ip = "192.168.1.1"
7  dest_port = 68
8  local_ip = ip_utils.get_local_ip()
9  local_port = ip_utils.get_free_port()
10
11 local_ip = dest_ip = "127.0.0.1"
12
13 address = (dest_ip, dest_port)
14
15 with closing(
16     socket.socket(

```

```

17         socket.AF_INET,
18         socket.SOCK_RAW,
19         socket.IPPROTO_UDP
20     )) as s:
21     try:
22         pkt = ip_utils.make_udp_packet(
23             local_port,
24             dest_port,
25             local_ip,
26             dest_ip
27         )
28         if pkt is not None:
29             packet = bytes(pkt)
30             s.sendto(packet, address)
31         else:
32             print(
33                 "Error making packet.",
34                 f"local port: [{local_port}]",
35                 f"destination port: [{dest_port}]",
36                 f"local ip: [{local_ip}]",
37                 f"destination ip: [{dest_ip}]",
38                 sep="\n"
39             )
40     except socket.error:
41         raise

```

Listing 18: A program for performing scans on UDP ports.

```

1  #!/usr/bin/env python
2  from modules import headers
3  from modules import ip_utils
4  import socket
5  import time
6  from collections import defaultdict
7  from contextlib import closing
8  from multiprocessing import Pool
9  from typing import Set, DefaultDict
10
11
12  def udp_listener(dest_ip: str, timeout: float) -> Set[int]:
13      """
14      This listener detects UDP packets from dest_ip in the given timespan,
15      all ports that send direct replies are marked as being open.
16      Returns a list of open ports.
17      """
18
19      time_remaining = timeout
20      ports: Set[int] = set()
21      with socket.socket(
22          socket.AF_INET,

```

```

23         socket.SOCK_RAW,
24         socket.IPPROTO_UDP
25     ) as s:
26         while True:
27             time_taken = ip_utils.wait_for_socket(s, time_remaining)
28             if time_taken == -1:
29                 break
30             else:
31                 time_remaining -= time_taken
32             packet = s.recv(1024)
33             ip = headers.ip(packet[:20])
34             udp = headers.udp(packet[20:28])
35             # unpack the UDP header
36             if dest_ip == ip.source and ip.protocol == 17:
37                 ports.add(udp.src)
38
39     return ports
40
41
42 def icmp_listener(src_ip: str, timeout: float = 2) -> int:
43     """
44     This listener detects ICMP destination unreachable
45     packets and returns the icmp code.
46     This is later used to mark them as either close, open|filtered,
47     filtered.
48     3 -> closed
49     0|1|2|9|10|13 -> filtered
50     -1 -> error with arguments
51     open|filtered means that they are either open or
52     filtered but return nothing.
53     """
54     ping_sock = socket.socket(
55         socket.AF_INET,
56         socket.SOCK_RAW,
57         socket.IPPROTO_ICMP
58     )
59     # open raw socket to listen for ICMP destination unrechable packets
60     time_remaining = timeout
61     code = -1
62     while True:
63         time_waiting = ip_utils.wait_for_socket(ping_sock,
64             time_remaining)
65         # wait for socket to be readable
66         if time_waiting == -1:
67             break
68         else:
69             time_remaining -= time_waiting
70             recPacket, addr = ping_sock.recvfrom(1024)
71             # recieve the packet

```



```

71     ip = headers.ip(recPacket[:20])
72     icmp = headers.icmp(recPacket[20:28])
73     valid_codes = [0, 1, 2, 3, 9, 10, 13]
74     if (
75         ip.source == src_ip
76         and icmp.type == 3
77         and icmp.code in valid_codes
78     ):
79         code = icmp.code
80         break
81     elif time_remaining <= 0:
82         break
83     else:
84         continue
85 ping_sock.close()
86 return code
87
88
89 def udp_scan(
90     dest_ip: str,
91     ports_to_scan: Set[int]
92 ) -> DefaultDict[str, Set[int]]:
93     """
94     Takes in a destination IP address in either dot or long form and
95     a list of ports to scan. Sends UDP packets to each port specified
96     in portlist and uses the listeners to mark them as open,
97     open|filtered,
98     filtered, closed they are marked open|filtered if no response is
99     recieved at all.
100     """
101
102     local_ip = ip_utils.get_local_ip()
103     local_port = ip_utils.get_free_port()
104     # get local ip address and port number
105     ports: DefaultDict[str, Set[int]] = defaultdict(set)
106     ports["REMAINING"] = ports_to_scan
107     p = Pool(1)
108     udp_listen = p.apply_async(udp_listener, (dest_ip, 4))
109     # start the UDP listener
110     with closing(
111         socket.socket(
112             socket.AF_INET,
113             socket.SOCK_RAW,
114             socket.IPPROTO_UDP
115         )
116     ) as s:
117         for _ in range(2):
118             # repeat 3 times because UDP scanning comes
119             # with a high chance of packet loss
120             for dest_port in ports["REMAINING"]:

```

```

120         try:
121             packet = ip_utils.make_udp_packet(
122                 local_port,
123                 dest_port,
124                 local_ip,
125                 dest_ip
126             )
127             # create the UDP packet to send
128             s.sendto(packet, (dest_ip, dest_port))
129             # send the packet to the currently scanning address
130         except socket.error:
131             packet_bytes = " ".join(map(hex, packet))
132             print(
133                 "The socket modules sendto method with the\n"
134                 "following",
135                 "argument resulting in a socket error.",
136                 f"\npacket: [{packet_bytes}]\n",
137                 "address: [{dest_ip, dest_port}]"
138             )
139     p.close()
140     p.join()
141
142     ports["OPEN"].update(udp_listen.get())
143
144     ports["REMAINING"] -= ports["OPEN"]
145     # only scan the ports which we know are not open
146     with closing(
147         socket.socket(
148             socket.AF_INET,
149             socket.SOCK_RAW,
150             socket.IPPROTO_UDP
151         )
152     ) as s:
153         for dest_port in ports["REMAINING"]:
154             try:
155                 packet = ip_utils.make_udp_packet(
156                     local_port,
157                     dest_port,
158                     local_ip,
159                     dest_ip
160                 )
161                 # make a new UDP packet
162                 p = Pool(1)
163                 icmp_listen = p.apply_async(icmp_listener, (dest_ip,))
164                 # start the ICMP listener
165                 time.sleep(1)
166                 s.sendto(packet, (dest_ip, dest_port))
167                 # send packet
168                 p.close()

```

```

169         p.join()
170         icmp_code = icmp_listen.get()
171         # recieve ICMP code from the ICMP listener
172         if icmp_code in {0, 1, 2, 9, 10, 13}:
173             ports["FILTERED"].add(dest_port)
174         elif icmp_code == 3:
175             ports["CLOSED"].add(dest_port)
176     except socket.error:
177         packet_bytes = " ".join(map("{:02x}".format, packet))
178         ip_utils.eprint(
179             "The socket modules sendto method with the following",
180             "argument resulting in a socket error.",
181             f"\npacket: [{packet_bytes}]\n",
182             "address: [{dest_ip, dest_port}]"
183         )
184     # this creates a new set which contains all the elements that
185     # are in the list of ports to be scanned but have not yet
186     # been classified
187     ports["OPEN|FILTERED"] = (
188         ports["REMAINING"]
189         - ports["OPEN"]
190         - ports["FILTERED"]
191         - ports["CLOSED"]
192     )
193     # set comprehension to update the list of open filtered ports
194     return ports
195
196
197 def main() -> None:
198     ports = udp_scan("127.0.0.1", {22, 68, 53, 6969})
199     print(f"Open ports: {ports['OPEN']}")
200     print(f"Open or filtered ports: {ports['OPEN|FILTERED']}")
201     print(f"Filtered ports: {ports['FILTERED']}")
202     print(f"Closed ports: {ports['CLOSED']}")

```

Listing 19: A program I made to open a port via UDP for testing my UDP scanner.

```

1  #!/usr/bin/env python
2
3  import socket
4  from contextlib import closing
5
6  with closing(
7      socket.socket(
8          socket.AF_INET,
9          socket.SOCK_DGRAM
10     )
11  ) as s:
12     s.bind(("127.0.0.1", 12345))

```

```

13     print("opened port 12345 on localhost")
14     while True:
15         data, addr = s.recvfrom(1024)
16         s.sendto(bytes("Well hello there good sir.", "utf-8"), addr)

```

6.6 version_detection

Listing 20: A program which does version detection on services.

```

1  #!/usr/bin/env python
2  from typing import Dict, Set, Pattern, Tuple, DefaultDict
3  from functools import reduce
4  from collections import defaultdict
5  from modules import directives
6  import re
7  import operator
8
9  # type annotaion for the container which
10 # holds the probes. I have abstracted it from
11 # the function definition because multiple functions
12 # depend on it and they weren't all getting updated
13 # if I needed to change the function signature.
14 PROBE_CONTAINER = DefaultDict[str, Dict[str, directives.Probe]]
15
16
17 def parse_ports(portstring: str) -> DefaultDict[str, Set[int]]:
18     """
19     This function takes in a port directive
20     and returns a set of the ports specified.
21     A set is used because it is O(1) for contains
22     operations as opposed for O(N) for lists.
23     """
24     # matches both the num-num port range format
25     # and the plain num port specification
26     # num-num form must come first otherwise it breaks.
27     proto_regex = re.compile(r"([ TU]):?([0-9,-]+)")
28     # THE SPACE IS IMPORTANT!!!
29     # it allows ports specified before TCP/UDP ports
30     # to be specified globally as in for all protocols.
31
32     pair_regex = re.compile(r"(\d+)-(\d+)")
33     single_regex = re.compile(r"(\d+)")
34     ports: DefaultDict[str, Set[int]] = defaultdict(set)
35     # searches contains the result of trying the pair_regex
36     # search against all of the command seperated
37     # port strings
38
39     for protocol, portstring in proto_regex.findall(portstring):
40         pairs = pair_regex.findall(portstring)

```

```

41     # for each pair of numbers in the pairs list
42     # separate each number and cast them to int
43     # then generate the range of numbers from x[0]
44     # to x[1]+1 then cast this range to a list
45     # and "reduce" the list of lists by joining them
46     # with operator.ior (inclusive or) and then let
47     # ports be the set of all the ports in that list.
48     proto_map = {
49         " ": "ANY",
50         "U": "UDP",
51         "T": "TCP"
52     }
53     if pairs:
54         def pair_to_ports(pair: Tuple[int, int]) -> Set[int]:
55             """
56             a function to go from a port pair i.e. (80-85)
57             to the set of specified ports: {80,81,82,83,84,85}
58             """
59             start, end = pair
60             return set(range(start, end+1))
61         # ports contains the set of all ANY/TCP/UDP specified ports
62         ports[proto_map[protocol]] = set(reduce(
63             operator.ior,
64             map(pair_to_ports, pairs)
65         ))
66
67         singles = single_regex.findall(portstring)
68         # for each of the ports that are specified on their own
69         # cast them to int and update the set of all ports with
70         # that list.
71         ports[proto_map[protocol]].update(map(int, singles))
72
73     return ports
74
75
76 def parse_probes(probe_file: str) -> PROBE_CONTAINER:
77     """
78     Extracts all of the probe directives from the
79     file pointed to by probe_file.
80     """
81     # lines contains each line of the file which doesn't
82     # start with a # and is not empty.
83     lines = [
84         line
85         for line in open(probe_file).read().splitlines()
86         if line and not line.startswith("#")
87     ]
88
89     # list holding each of the probe directives.
90     probes: PROBE_CONTAINER = defaultdict(dict)

```

```

91
92 regexes: Dict[str, Pattern] = {
93     "probe": re.compile(r"Probe (TCP|UDP) (\S+) q\|(.*)\|"),
94     "match": re.compile(" ".join([
95         r"(?P<type>softmatch|match)",
96         r"(?P<service>\S+)",
97         r"m([\/%=|])(?P<regex>.+?)\3(?P<flags>[si]*)"
98     ])),
99     "rarity": re.compile(r"rarity (\d+)"),
100     "totalwaitms": re.compile(r"totalwaitms (\d+)"),
101     "tcpwrappedms": re.compile(r"tcpwrappedms (\d+)"),
102     "fallback": re.compile(r"fallback (\S+)"),
103     "ports": re.compile(r"ports (\S+)"),
104     "exclude": re.compile(r"Exclude T:(\S+)")
105 }
106
107 # parse the probes out from the file
108 for line in lines:
109     # add any ports to be excluded to the base probe class
110     if line.startswith("Exclude"):
111         search = regexes["exclude"].search(line)
112         if search:
113             # parse the ports from the grouped output of
114             # a search with the regex defined above.
115             for protocol, ports in
116                 parse_ports(search.group(1)).items():
117                 directives.Probe.exclude[protocol].update(ports)
118         else:
119             print(line)
120             input()
121
122 # new probe directive
123 if line.startswith("Probe"):
124     # parse line into probe protocol, name and probestring
125     search = regexes["probe"].search(line)
126     if search:
127         try:
128             proto, name, string = search.groups()
129         except ValueError:
130             print(line)
131             raise
132         probes[name][proto] = directives.Probe(proto, name,
133             string)
134         # assign current_probe to the most recently added probe
135         current_probe = probes[name][proto]
136     else:
137         print(line)
138         input()
139
140 # new match directive

```

```

139 elif line.startswith("match") or line.startswith("softmatch"):
140     search = regexes["match"].search(line)
141     if search:
142         # the remainder of the string after the match
143         version_info = line[search.end()+1:]
144         # escape the curly braces so the regex engine doesn't
145         # consider them to be special characters
146         pattern = bytes(search.group("regex"), "utf-8")
147         # these replace the literal \n, \r and \t
148         # strings with their actual characters
149         # i.e. \n -> newline character
150         pattern = pattern.replace(b"\\n", b"\n")
151         pattern = pattern.replace(b"\\r", b"\r")
152         pattern = pattern.replace(b"\\t", b"\t")
153         matcher = directives.Match(
154             search.group("service"),
155             pattern,
156             search.group("flags"),
157             version_info
158         )
159         if search.group("type") == "match":
160             current_probe.matches.add(matcher)
161         else:
162             current_probe.softmatches.add(matcher)
163
164     else:
165         print(line)
166         input()
167
168 # new ports directive
169 elif line.startswith("ports"):
170     search = regexes["ports"].search(line)
171     if search:
172         for protocol, ports in
173             parse_ports(search.group(1)).items():
174             current_probe.ports[protocol].update(ports)
175     else:
176         print(line)
177         input()
178
179 # new totalwaitms directive
180 elif line.startswith("totalwaitms"):
181     search = regexes["totalwaitms"].search(line)
182     if search:
183         current_probe.totalwaitms = int(search.group(1))
184     else:
185         print(line)
186         input()
187
188 # new rarity directive
189 elif line.startswith("rarity"):

```

```

188         search = regexes["rarity"].search(line)
189         if search:
190             current_probe.rarity = int(search.group(1))
191         else:
192             print(line)
193             input()
194
195         # new fallback directive
196         elif line.startswith("fallback"):
197             search = regexes["fallback"].search(line)
198             if search:
199                 current_probe.fallback = set(search.group(1).split(","))
200             else:
201                 print(line)
202                 input()
203     return probes
204
205
206 def version_detect_scan(
207     target: directives.Target,
208     probes: PROBE_CONTAINER
209 ) -> directives.Target:
210     for probe_dict in probes.values():
211         for proto in probe_dict:
212             target = probe_dict[proto].scan(target)
213     return target
214
215
216 def main() -> None:
217     print("reached here")
218     probes = parse_probes("./version_detection/nmap-service-probes")
219     open_ports: DefaultDict[str, Set[int]] = defaultdict(set)
220     open_filtered_ports: DefaultDict[str, Set[int]] = defaultdict(set)
221     open_filtered_ports["TCP"].add(22)
222     open_ports["TCP"].update([1, 2, 3, 4, 5, 6, 8, 65,
223                             20, 21, 23, 24, 25])
224
225     target = directives.Target(
226         "127.0.0.1",
227         open_ports,
228         open_filtered_ports
229     )
230     target.open_ports["TCP"].update([1, 2, 3])
231     print("BEFORE")
232     print(target)
233     scanned = version_detect_scan(target, probes)
234     print("AFTER")
235     print(scanned)

```

6.7 modules

Listing 21: A python module I wrote for parsing and holding the version detection probes from the `nmap_service_probes` file.

```
1  #!/usr/bin/env python
2  from collections import defaultdict
3  from contextlib import closing
4  from dataclasses import dataclass, field
5  from functools import reduce
6  from string import whitespace, printable
7  from typing import (
8          DefaultDict,
9          Dict,
10         Set,
11         List,
12         Pattern,
13         Match as RE_Match,
14         Tuple
15 )
16 from . import ip_utils
17 import operator
18 import re
19 import socket
20 import struct
21
22
23 class Match:
24     """
25         This is a class for both Matches and
26         Softmatches as they are actually the same
27         thing except that softmatches have less information.
28     """
29     options_to_flags = {
30             "i": re.IGNORECASE,
31             "s": re.DOTALL
32     }
33     letter_to_name = {
34             "p": "vendorproductname",
35             "v": "version",
36             "i": "info",
37             "h": "hostname",
38             "o": "operatingsystem",
39             "d": "devicetype"
40     }
41     cpe_part_map: Dict[str, str] = {
42             "a": "applications",
43             "h": "hardware platforms",
44             "o": "operating systems"
45     }
```

```

46     # look into match.expand when looking at the substring version info
47     things.
48
49     def __init__(
50         self,
51         service: str,
52         pattern: bytes,
53         pattern_options: str,
54         version_info: str
55     ):
56         self.version_info: Dict[str, str] = dict()
57         self.cpes: Dict[str, Dict[str, str]] = dict()
58         self.service: str = service
59         # bitwise or is used to combine flags
60         # pattern options will never be anything but a
61         # combination of s and i.
62         # the default value of re.V1 is so that
63         # re uses the newer matching engine.
64         flags = reduce(
65             operator.ior,
66             [
67                 self.options_to_flags[opt]
68                 for opt in pattern_options
69             ],
70             0
71         )
72         try:
73             self.pattern: Pattern = re.compile(
74                 pattern,
75                 flags=flags
76             )
77         except Exception as e:
78             print("Regex failed to compile:")
79             print(e)
80             print(pattern)
81             input()
82
83         vinfo_regex = re.compile(r"([pvihod] |cpe:)([/|])(.+?)\2([a]*)")
84         cpe_regex = re.compile(
85             "?:?".join((
86                 "(?P<part>[aho])",
87                 "(?P<vendor>[~:]*)",
88                 "(?P<product>[~:]*)",
89                 "(?P<version>[~:]*)",
90                 "(?P<update>[~:]*)",
91                 "(?P<edition>[~:]*)",
92                 "(?P<language>[~:]*)"
93             ))
94         )

```

```

95     for fieldname, _, val, opts in vinfo_regex.findall(version_info):
96         if fieldname == "cpe:":
97             search = cpe_regex.search(val)
98             if search:
99                 part = search.group("part")
100                 # this next bit is so that the bytes produced by the
101                 # regex
102                 # are turned to strings
103                 self.cpes[Match.cpe_part_map[part]] = {
104                     key: value
105                     for key, value
106                     in search.groupdict().items()
107                 }
108             else:
109                 self.version_info[
110                     Match.letter_to_name[fieldname]
111                 ] = val
112
113 def __repr__(self) -> str:
114     return "Match(" + ", ".join((
115         f"service={self.service}",
116         f"pattern={self.pattern}",
117         f"version_info={self.version_info}",
118         f"cpes={self.cpes}"
119     )) + ")"
120
121 def matches(self, string: bytes) -> bool:
122     def replace_groups(
123         string: str,
124         original_match: RE_Match
125     ) -> str:
126         """
127         This function takes in a string and the original
128         regex search performed on the data recieved and
129         replaces all of the $i, $SUBST, $I, $P occurrences
130         with the relavant formatted text that they produce.
131         """
132         def remove_unprintable(
133             group: int,
134             original_match: RE_Match
135         ) -> bytes:
136             """
137             Mirrors the P function from nmap which
138             is used to print only printable characters.
139             i.e. W\00\OR\OK\OG\OR\00\OU\OP -> WORKGROUP
140             """
141             return b"".join(
142                 i for i in original_match.group(group)
143                 if ord(i) in (
144                     set(printable)

```

```

144         - set(whitespace)
145         | {" "}
146     )
147 )
148 # if i in the set of all printable characters,
149 # excluding those of which that are whitespace characters
150 # but including space.
151
152 def substitute(
153     group: int,
154     before: bytes,
155     after: bytes,
156     original_match: RE_Match
157 ) -> bytes:
158     """
159     Mirrors the SUBST function from nmap which is used to
160     format some information found by the regex.
161     by substituting all instances of 'before' with 'after'.
162     """
163     return original_match.group(group).replace(before, after)
164
165 def unpack_uint(
166     group: int,
167     endianness: str,
168     original_match: RE_Match
169 ) -> bytes:
170     """
171     Mirrors the I function from nmap which is used to
172     unpack an unsigned int from some bytes.
173     """
174     return bytes(struct.unpack(
175         endianness + "I",
176         original_match.group(group)
177     ))
178
179 text = bytes(string, "utf-8")
180 # fill in the version information from the regex match
181 # find all the dollar groups:
182 dollar_regex = re.compile(r"\$(\d)")
183 # find all the $i's in string
184 numbers = set(int(i) for i in dollar_regex.findall(string))
185 # for each $i found i
186 for group in numbers:
187     text = text.replace(
188         bytes(f"${group}", "utf-8"),
189         original_match.group(group)
190     )
191 # having replaced all of the groups we can now
192 # start doing the SUBST, P and I commands.
193 subst_regex = re.compile(rb"\$SUBST\((\d),(.+),(.+)\)")

```

```

194         # iterate over all of the matches found by the SUBST regex
195         for match in subst_regex.finditer(text):
196             num, before, after = match.groups()
197             # replace the full match (group 0)
198             # with the output of substitute
199             # with the specific arguments
200             text.replace(
201                 match.group(0),
202                 substitute(int(num), before, after, original_match)
203             )
204
205         p_regex = re.compile(rb"\$P\((\d)\)")
206         for match in p_regex.finditer(text):
207             num = match.group(1)
208             # replace the full match (group 0)
209             # with the output of remove_unprintable
210             # with the specific arguments
211             text.replace(
212                 match.group(0),
213                 remove_unprintable(int(num), original_match)
214             )
215
216         i_regex = re.compile(br"\$I\((\d),\s*(\S)\s*\)")
217         for match in i_regex.finditer(text):
218             num, endianness = match.groups()
219             # this means replace group 0 -> the whole match
220             # with the output of the unpack_uint
221             # with the specified arguments
222             text.replace(
223                 match.group(0),
224                 unpack_uint(
225                     int(num.decode()),
226                     endianness.decode(),
227                     original_match
228                 )
229             )
230
231         return text.decode()
232
233     search = self.pattern.search(string)
234     if search:
235         # the fields to replace are all the CPE groups,
236         # all of the version info fields.
237         self.version_info = {
238             key: replace_groups(value, search)
239             for key, value in self.version_info.items()
240         }
241         self.cpes = {
242             outer_key: {
243                 inner_key: replace_groups(value, search)

```

```

244         for inner_key, value in outer_dict.items()
245     }
246     for outer_key, outer_dict in self.cpes.items()
247 }
248
249     return True
250 else:
251     return False
252
253
254 @dataclass
255 class Target:
256     """
257     This class holds data about targets to
258     scan. the dataclass decorator is simply
259     a way of python automatically writing some
260     of the basic methods a class for storing data
261     has, such as __repr__ for printing information
262     in the object etc.
263     """
264     address: str
265     open_ports: DefaultDict[str, Set[int]]
266     open_filtered_ports: DefaultDict[str, Set[int]]
267     services: Dict[int, Match] = field(default_factory=dict)
268
269     def __repr__(self) -> str:
270         def collapse(port_dict: DefaultDict) -> str:
271             """
272             Collapse a list of port numbers so that
273             only the unique ones and the start and end
274             of a sequence are displayed.
275             1,2,3,4,5,7,9,11,13,14,15,16,17 -> 1-5,7,9,11,13-17
276             """
277             store_results = list()
278             for key in port_dict:
279                 # items is a sorted list of a set of ports.
280                 items: List[int] = sorted(port_dict[key])
281                 key_result = f'"{key}":' + "{"
282                 # if its an empty list return now to avoid errors
283                 if len(items) != 0:
284                     new_sequence = False
285                     # enumerate up until the one before
286                     # the last to prevent index errors.
287                     for index, item in enumerate(items[:-1]):
288                         # if its the first one add it on
289                         if index == 0:
290                             key_result += f"{item}"
291                         # if its a sequence start one else put a comma
292                         if items[index+1] == item+1:
293                             key_result += "-"

```

```

294         else:
295             key_result += ","
296             # if the sequence breaks then put a comma
297             elif item+1 != items[index+1]:
298                 key_result += f"{item},"
299                 new_sequence = True
300             # if its a new sequence the put the '-s in
301             elif item+1 == items[index+1] and new_sequence:
302                 key_result += f"{item}-"
303                 new_sequence = False
304             # because we only iterate to the one before
305             # the last element, add the last element on to the end.
306             key_result += f"{items[-1]}" + "}"
307             store_results.append(key_result)
308         # format the final result
309         result = "{" + ", ".join(store_results) + "}"
310         return result
311
312     open_ports = collapse(self.open_ports)
313     open_filtered_ports = collapse(self.open_filtered_ports)
314     return ", ".join((
315         f"Target(address=[{self.address}]",
316         f"open_ports=[{open_ports}]",
317         f"open_filtered_ports=[{open_filtered_ports}]",
318         f"services={self.services})"
319     ))
320
321
322 class Probe:
323     """
324     This class represents the Probe directive of the nmap-service-probes
325     file.
326
327     It holds information such as the protocol to use, the string to send,
328     the ports to scan, the time to wait for a null TCP to return a
329     banner,
330
331     the rarity of the probe (how often it will return a response) and the
332     probes to try if this one fails.
333     """
334
335     # a default dict is one which takes in a
336     # "default factory" which is called when
337     # a new key is introduced to the dict
338     # in this case the default factory is
339     # the set function meaning that when I
340     # do exclude[protocol].update(ports)
341     # but exclude[protocol] has not yet been defined
342     # it will be defined as an empty set
343     # allowing me to update it with ports.
344     exclude: DefaultDict[str, Set[int]] = defaultdict(set)
345     proto_to_socket_type: Dict[str, int] = {

```

```

342         "TCP": socket.SOCK_STREAM,
343         "UDP": socket.SOCK_DGRAM
344     }
345
346     def __init__(self, protocol: str, probename: str, probe: str):
347         """
348         This is the initial function that is called by the
349         constructor of the Probe class, it is used to define
350         the variables that are specific to each instance of
351         the class.
352         """
353         if protocol in {"TCP", "UDP"}:
354             self.protocol = protocol
355         else:
356             raise ValueError(
357                 f"Probe object must have protocol TCP or UDP not
358                     {protocol}."
359             )
360         self.name: str = probename
361         self.string: str = probe
362         self.payload: bytes = bytes(probe, "utf-8")
363         self.matches: Set[Match] = set()
364         self.softmatches: Set[Match] = set()
365         self.ports: DefaultDict[str, Set[int]] = defaultdict(set)
366         self.totalwaitms: int = 6000
367         self.tcpwrappedms: int = 3000
368         self.rarity: int = -1
369         self.fallback: Set[str] = set()
370
371     def __repr__(self) -> str:
372         """
373         This is the function that is called when something
374         tries to print an instance of this class.
375         It is used to reveal information internal
376         to the class.
377         """
378         return ", ".join([
379             f"Probe({self.protocol}",
380             f"{self.name}",
381             f"\n{self.string}\n",
382             f"{len(self.matches)} matches",
383             f"{len(self.softmatches)} softmatches",
384             f"ports: {self.ports}",
385             f"rarity: {self.rarity}",
386             f"fallbacks: {self.fallback}"
387         ])
388
389     def scan(self, target: Target) -> Target:
390         """
391         scan takes in an object of class Target to
392         probe and attempts to detect the version of

```



```

391     any services running on the machine.
392     """
393     # this constructs the set of all ports,
394     # that are either open or open_filtered,
395     # and are in the set of ports to scan for
396     # this particular probe, this means that,
397     # we are only connecting to ports that we
398     # know are not closed and are not to be excluded.
399
400     ports_to_scan: Set[int] = (
401         (
402             target.open_filtered_ports[self.protocol]
403             | target.open_ports[self.protocol]
404         )
405     ) - Probe.exclude[self.protocol] - Probe.exclude["ANY"]
406     # if the probe defines a set of ports to scan
407     # then don't scan any that aren't defined for it
408     if self.ports[self.protocol] != set():
409         ports_to_scan &= self.ports[self.protocol]
410     for port in ports_to_scan:
411         # open a self closing IPV4 socket
412         # for the correct protocol for this probe.
413         with closing(
414             socket.socket(
415                 socket.AF_INET,
416                 self.proto_to_socket_type[self.protocol]
417             )
418         ) as sock:
419             # setup the connection to the target
420             try:
421                 sock.connect((target.address, port))
422                 # if the connection fails then continue scanning
423                 # the next ports, this shouldn't really happen.
424             except ConnectionError:
425                 continue
426             # send the payload to the target
427             sock.send(self.payload)
428             # wait for the target to send a response
429             time_taken = ip_utils.wait_for_socket(
430                 sock,
431                 self.totalwaitms/1000
432             )
433             # if the response didn't time out
434             if time_taken != -1:
435                 # if the port was in open_filtered move it to open
436                 if port in target.open_filtered_ports[self.protocol]:
437                     target.open_filtered_ports[self.protocol].remove(port)
438                     target.open_ports[self.protocol].add(port)
439
440             # recieve the data and decode it to a string

```

```

441         data_recieved = sock.recv(4096)
442         # print("Recieved", data_recieved)
443         service = ""
444         # try and softmatch the service first
445         for softmatch in self.softmatches:
446             if softmatch.matches(data_recieved):
447                 service = softmatch.service
448                 target.services[port] = softmatch
449                 break
450         # try and get a full match for the service
451         for match in self.matches:
452             if service in match.service.lower():
453                 if match.matches(data_recieved):
454                     target.services[port] = match
455                     break
456         return target
457
458
459 PROBE_CONTAINER = DefaultDict[str, Dict[str, Probe]]
460
461
462 def parse_ports(portstring: str) -> DefaultDict[str, Set[int]]:
463     """
464     This function takes in a port directive
465     and returns a set of the ports specified.
466     A set is used because it is O(1) for contains
467     operations as opposed for O(N) for lists.
468     """
469     # matches both the num-num port range format
470     # and the plain num port specification
471     # num-num form must come first otherwise it breaks.
472     proto_regex = re.compile(r"([ TU]?):?([0-9,-]+)")
473     # THE SPACE IS IMPORTANT!!!
474     # it allows ports specified before TCP/UDP ports
475     # to be specified globally as in for all protocols.
476
477     pair_regex = re.compile(r"(\d+)-(\d+)")
478     single_regex = re.compile(r"(\d+)")
479     ports: DefaultDict[str, Set[int]] = defaultdict(set)
480     # searches contains the result of trying the pair_regex
481     # search against all of the command seperated
482     # port strings
483
484     for protocol, portstring in proto_regex.findall(portstring):
485         pairs = pair_regex.findall(portstring)
486         # for each pair of numbers in the pairs list
487         # seperate each number and cast them to int
488         # then generate the range of numbers from x[0]
489         # to x[1]+1 then cast this range to a list
490         # and "reduce" the list of lists by joining them

```

```

491     # with operator.ior (inclusive or) and then let
492     # ports be the set of all the ports in that list.
493     proto_map = {
494         "": "ANY",
495         " ": "ANY",
496         "U": "UDP",
497         "T": "TCP"
498     }
499     if pairs:
500         def pair_to_ports(pair: Tuple[str, str]) -> Set[int]:
501             """
502             a function to go from a port pair i.e. (80-85)
503             to the set of specified ports: {80,81,82,83,84,85}
504             """
505             start, end = pair
506             return set(range(
507                 int(start),
508                 int(end)+1
509             ))
510             # ports contains the set of all ANY/TCP/UDP specified ports
511             ports[proto_map[protocol]] = set(reduce(
512                 operator.ior,
513                 map(pair_to_ports, pairs)
514             ))
515
516             singles = single_regex.findall(portstring)
517             # for each of the ports that are specified on their own
518             # cast them to int and update the set of all ports with
519             # that list.
520             ports[proto_map[protocol]].update(map(int, singles))
521
522     return ports
523
524
525 def parse_probes(probe_file: str) -> PROBE_CONTAINER:
526     """
527     Extracts all of the probe directives from the
528     file pointed to by probe_file.
529     """
530     # lines contains each line of the file which doesn't
531     # start with a # and is not empty.
532     lines = [
533         line
534         for line in open(probe_file).read().splitlines()
535         if line and not line.startswith("#")
536     ]
537
538     # list holding each of the probe directives.
539     probes: PROBE_CONTAINER = defaultdict(dict)
540

```

```

541 regexes: Dict[str, Pattern] = {
542     "probe": re.compile(r"Probe (TCP|UDP) (\S+) q\|(.*)\|"),
543     "match": re.compile(" ".join([
544         r"(?P<type>softmatch|match)",
545         r"(?P<service>\S+)",
546         r"m([\@/%=|])(?P<regex>.+?)\3(?P<flags>[si]*)"
547     ])),
548     "rarity": re.compile(r"rarity (\d+)"),
549     "totalwaitms": re.compile(r"totalwaitms (\d+)"),
550     "tcpwrappedms": re.compile(r"tcpwrappedms (\d+)"),
551     "fallback": re.compile(r"fallback (\S+)"),
552     "ports": re.compile(r"ports (\S+)"),
553     "exclude": re.compile(r"Exclude T:(\S+)")
554 }
555
556 # parse the probes out from the file
557 for line in lines:
558     # add any ports to be excluded to the base probe class
559     if line.startswith("Exclude"):
560         search = regexes["exclude"].search(line)
561         if search:
562             # parse the ports from the grouped output of
563             # a search with the regex defined above.
564             for protocol, ports in
565                 parse_ports(search.group(1)).items():
566                 Probe.exclude[protocol].update(ports)
567         else:
568             print(line)
569             input()
570
571 # new probe directive
572 if line.startswith("Probe"):
573     # parse line into probe protocol, name and probestring
574     search = regexes["probe"].search(line)
575     if search:
576         try:
577             proto, name, string = search.groups()
578         except ValueError:
579             print(line)
580             raise
581         probes[name][proto] = Probe(proto, name, string)
582         # assign current_probe to the most recently added probe
583         current_probe = probes[name][proto]
584     else:
585         print(line)
586         input()
587
588 # new match directive
589 elif line.startswith("match") or line.startswith("softmatch"):
590     search = regexes["match"].search(line)

```

```

590         if search:
591             # the remainder of the string after the match
592             version_info = line[search.end()+1:]
593             # escape the curly braces so the regex engine doesn't
594             # consider them to be special characters
595             pattern = bytes(search.group("regex"), "utf-8")
596             # these replace the literal \n, \r and \t
597             # strings with their actual characters
598             # i.e. \n -> newline character
599             pattern = pattern.replace(b"\\n", b"\n")
600             pattern = pattern.replace(b"\\r", b"\r")
601             pattern = pattern.replace(b"\\t", b"\t")
602             matcher = Match(
603                 search.group("service"),
604                 pattern,
605                 search.group("flags"),
606                 version_info
607             )
608             if search.group("type") == "match":
609                 current_probe.matches.add(matcher)
610             else:
611                 current_probe.softmatches.add(matcher)
612
613         else:
614             print(line)
615             input()
616
617     # new ports directive
618     elif line.startswith("ports"):
619         search = regexes["ports"].search(line)
620         if search:
621             for protocol, ports in
622                 parse_ports(search.group(1)).items():
623                 current_probe.ports[protocol].update(ports)
624         else:
625             print(line)
626             input()
627
628     # new totalwaitms directive
629     elif line.startswith("totalwaitms"):
630         search = regexes["totalwaitms"].search(line)
631         if search:
632             current_probe.totalwaitms = int(search.group(1))
633         else:
634             print(line)
635             input()
636
637     # new rarity directive
638     elif line.startswith("rarity"):
639         search = regexes["rarity"].search(line)
640         if search:

```

```

639         current_probe.rarity = int(search.group(1))
640     else:
641         print(line)
642         input()
643
644     # new fallback directive
645     elif line.startswith("fallback"):
646         search = regexes["fallback"].search(line)
647         if search:
648             current_probe.fallback = set(search.group(1).split(","))
649         else:
650             print(line)
651             input()
652     return probes

```

Listing 22: A python module I made to dissect and hold protocol headers.

```

1  import struct
2  import socket
3  from typing import Dict
4
5
6  class ip:
7      """
8      A class for parsing, storing and displaying
9      data from an IP header.
10     """
11     def __init__(self, header: bytes):
12         # first unpack the IP header
13         (
14             ip_hp_ip_v,
15             ip_dscp_ip_ecn,
16             ip_len,
17             ip_id,
18             ip_flg_ip_off,
19             ip_ttl,
20             ip_p,
21             ip_sum,
22             ip_src,
23             ip_dst
24         ) = struct.unpack('!BBHHHBBHII', header)
25         # now deal with the sub-byte sized components
26         hl_v = f"{ip_hp_ip_v:08b}"
27         ip_v = int(hl_v[:4], 2)
28         ip_hl = int(hl_v[4:], 2)
29         # splits hl_v in ip_v and ip_hl which store the IP version
30         # number and
31         # header length respectively
32         dscp_ecn = f"{ip_dscp_ip_ecn:08b}"
33         ip_dscp = int(dscp_ecn[:6], 2)

```

```

33     ip_ecn = int(dscp_ecn[6:], 2)
34     # splits dscp_ecn into ip_dscp and ip_ecn
35     # which are two of the components
36     # in an IP header
37     flgs_off = f"{ip_flg_ip_off:016b}"
38     ip_flg = int(flgs_off[:3], 2)
39     ip_off = int(flgs_off[3:], 2)
40     # splits flgs_off into ip_flg and ip_off which represent the ip
        header
41     # flags and the data offset
42     src_addr = socket.inet_ntoa(struct.pack('!I', ip_src))
43     dst_addr = socket.inet_ntoa(struct.pack('!I', ip_dst))
44     self.version: int = ip_v
45     self.header_length: int = ip_hl
46     self.dscp: int = ip_dscp
47     self.ecn: int = ip_ecn
48     self.len: int = ip_len
49     self.id: int = ip_id
50     self.flags: int = ip_flg
51     self.data_offset: int = ip_off
52     self.time_to_live: int = ip_ttl
53     self.protocol: int = ip_p
54     self.checksum: int = ip_sum
55     self.source: str = src_addr
56     self.destination: str = dst_addr
57
58     def __repr__(self) -> str:
59         return "\n\t".join((
60             "IP header:",
61             f"Version: [{self.version}]",
62             f"Internet Header Length: [{self.header_length}]",
63             f"Differentiated Services Point Code: [{self.dscp}]",
64             f"Explicit Congestion Notification: [{self.ecn}]",
65             f"Total Length: [{self.len}]",
66             f"Identification: [{self.id:04x}]",
67             f"Flags: [{self.flags:03b}]",
68             f"Fragment Offset: [{self.data_offset}]",
69             f"Time To Live: [{self.time_to_live}]",
70             f"Protocol: [{self.protocol}]",
71             f"Header Checksum: [{self.checksum:04x}]",
72             f"Source Address: [{self.source}]",
73             f"Destination Address: [{self.destination}]"
74         ))
75
76
77     class icmp:
78         """
79         A class for parsing, storing and displaying
80         data from an IP header.
81         """

```

```

82     # relates the type and code to the message
83     messages: Dict[int, Dict[int, str]] = {
84         0: {
85             0: "Echo reply."
86         },
87         3: {
88             0: "Destination network unreachable.",
89             1: "Destination host unreachable",
90             2: "Destination protocol unreachable",
91             3: "Destination port unreachable",
92             4: "Fragmentation required, and DF flag set.",
93             5: "Source route failed.",
94             6: "Destination network unknown.",
95             7: "Destination host unknown.",
96             8: "Source host isolated.",
97             9: "Network administratively prohibited.",
98             10: "Host administratively prohibited.",
99             11: "Network unreachable for ToS.",
100            12: "Host unreachable for ToS.",
101            13: "Communication administratively prohibited.",
102            14: "Host precedence violation.",
103            15: "Precedence cutoff in effect."
104        },
105        4: {
106            0: "Source quench."
107        },
108        5: {
109            0: "Redirect datagram for the network",
110            1: "Redirect datagram for the host.",
111            2: "Redirect datagram for the ToS & network.",
112            3: "Redirect datagram for the ToS & host."
113        },
114        8: {
115            0: "Echo request."
116        },
117        9: {
118            0: "Router advertisement"
119        },
120        10: {
121            0: "Router discovery/selection/solicitation."
122        },
123        11: {
124            0: "TTL expired in transit",
125            1: "Fragment reassembly time exceeded."
126        },
127        12: {
128            0: "Bad IP header: pointer indicates error.",
129            1: "Bad IP header: missing a required option.",
130            2: "Bad IP header: Bad length."
131        },

```



```

132         13: {
133             0: "Timestamp"
134         },
135         14: {
136             0: "Timestamp reply"
137         },
138         15: {
139             0: "Information request."
140         },
141         16: {
142             0: "Information reply."
143         },
144         17: {
145             0: "Address mask request."
146         },
147         18: {
148             0: "Address mask reply."
149         }
150     }
151
152     def __init__(self, header: bytes):
153         (
154             ICMP_type,
155             code,
156             csum,
157             remainder
158         ) = struct.unpack('!bbHI', header)
159
160         self.type: int = ICMP_type
161         self.code: int = code
162         self.checksum: int = csum
163
164         self.message: str
165         try:
166             self.message = icmp.messages[self.type][self.code]
167         except KeyError:
168             # if we can't assign a message then just set a description
169             # as to what caused the failure.
170             self.message = f"Failed to assign message:
171                             ({self.type/self.code})"
172
173         self.id: int
174         self.sequence: int
175         if self.type in {0, 8}:
176             self.id = socket.htons(remainder >> 16)
177             self.sequence = socket.htons(remainder & 0xFFFF)
178         else:
179             self.id = -1
180             self.sequence = -1

```

```

181     def __repr__(self) -> str:
182         return "\n\t".join((
183             "ICMP header:",
184             f"Message: [{self.message}]",
185             f"Type: [{self.type}]",
186             f"Code: [{self.code}]",
187             f"Checksum: [{self.checksum:04x}]",
188             f"ID: [{self.id}]",
189             f"Sequence: [{self.sequence}]"
190         ))
191
192
193     class tcp:
194         def __init__(self, header: bytes):
195             (
196                 src_prt,
197                 dst_prt,
198                 seq,
199                 ack,
200                 data_offset,
201                 flags,
202                 window_size,
203                 checksum,
204                 urg
205             ) = struct.unpack("!HHIIBBHH", header)
206
207             self.source: int = src_prt
208             self.destination: int = dst_prt
209             self.seq: int = seq
210             self.ack: int = ack
211             self.data_offset: int = data_offset >> 4
212             self.flags: int = flags + ((data_offset & 0x01) << 8)
213             self.window_size: int = window_size
214             self.checksum: int = checksum
215             self.urg: int = urg
216
217         def __repr__(self) -> str:
218             return "\n\t".join((
219                 "TCP header:",
220                 f"Source port: [{self.source}]",
221                 f"Destination port: [{self.destination}]",
222                 f"Sequence number: [{self.seq}]",
223                 f"Acknowledgement number: [{self.ack}]",
224                 f"Data offset: [{self.data_offset}]",
225                 f"Flags: [{self.flags:08b}]",
226                 f"Window size: [{self.window_size}]",
227                 f"Checksum: [{self.checksum:04x}]",
228                 f"Urgent: [{self.urg}]"
229             ))
230

```

```

231
232 class udp:
233     def __init__(self, header: bytes):
234         # parse udp header
235         (
236             src_port,
237             dest_port,
238             length,
239             checksum
240         ) = struct.unpack("!HHHH", header)
241
242         self.src: int = src_port
243         self.dest: int = dest_port
244         self.length: int = length
245         self.checksum: int = checksum
246
247     def __repr__(self) -> str:
248         return "\n\t".join((
249             "UDP header:",
250             f"Source port: {self.src}",
251             f"Destination port: {self.dest}",
252             f"Length: {self.length}",
253             f"Checksum: {self.checksum:04x}"
254         ))

```

Listing 23: A python module I wrote to contain lots of useful functions which I found I was declaring in multiple places and making changes so I decided to keep an up to date central one.

```

1  import array
2  import socket
3  import struct
4  import select
5  import time
6
7  from contextlib import closing
8  from functools import singledispatch
9  from itertools import islice, cycle
10 from sys import stderr
11 from typing import Set, Union
12
13
14 def eprint(*args: str, **kwargs: str) -> None:
15     """
16     Mirrors print exactly but prints to stderr
17     instead of stdout.
18     """
19     print(*args, file=stderr, **kwargs) # type: ignore
20
21

```

```

22 def long_to_dot(long: int) -> str:
23     """
24     Take in an IP address in packed 32 bit int form
25     and return that address in dot notation.
26     i.e. long_to_dot(0x7F000001) = 127.0.0.1
27     """
28     # these are long form values for 0.0.0.0
29     # and 255.255.255.255
30     if not 0 <= long <= 0xFFFFFFFF:
31         raise ValueError(f"Invalid long form IP address: [{long:08x}]")
32     else:
33         # shift the long form IP along 0, 8, 16, 24 bits
34         # take only the first 8 bits of the newly shifted number
35         # cast them to a string and join them with '.'s
36         return ".".join(
37             str(
38                 (long >> (8*(3-i))) & 0xFF
39             )
40             for i in range(4)
41         )
42
43
44 def dot_to_long(ip: str) -> int:
45     """
46     Take an ip address in dot notation and return the packed 32 bit int
47     version
48     i.e. dot_to_long("127.0.0.1") = 0x7F000001
49     """
50     # dot form ips: a.b.c.d must have each
51     # part (a,b,c,d) between 0 and 255,
52     # otherwise they are invalid
53
54     parts = [int(i) for i in ip.split(".")]
55
56     if not all(
57         0 <= i <= 255
58         for i in parts
59     ):
60         raise ValueError(f"Invalid dot form IP address: [{ip}]")
61
62     else:
63         # for each part of the dotted IP address
64         # bit shift left each part by eight times
65         # three minus it's position. This puts the bits
66         # from each part in the right place in the final sum
67         # a.b.c.d -> a<<3*8 + b<<2*8 + c<<1*8 + d<<0*8
68         return sum(
69             part << ((3-i)*8)
70             for i, part in enumerate(parts)

```

```

71         )
72
73
74     @singledispatch
75     def is_valid_ip(ip: Union[str, int]) -> bool:
76         """
77         checks whether a given IP address is valid.
78         """
79
80
81     @is_valid_ip.register
82     def _(ip: int):
83         # this is the int overload variant of
84         # the is_valid_ip function.
85         try:
86             # try to turn the long form ip address
87             # to a dot form one, if it fails,
88             # then return False, else return True
89             long_to_dot(ip)
90             return True
91         except ValueError:
92             return False
93
94
95     # the type ignore comment is required to stop
96     # mypy exploding over the fact I have defined '_' twice.
97     @is_valid_ip.register # type: ignore
98     def _(ip: str):
99         # this is the string overload variant
100         # of the is_valid_ip function.
101         try:
102             # try to turn the dot form ip address
103             # to a long form one, if it fails,
104             # then return False, else return True
105             dot_to_long(ip)
106             return True
107         except ValueError:
108             return False
109
110
111     def is_valid_port_number(port_num: int) -> bool:
112         """
113         Checks whether the given port number is valid i.e. between 0 and
114         65536.
115         """
116         # port numbers must be between 0 and 65535(216 - 1)
117         if 0 <= port_num < 2**16:
118             return True
119         else:
120             return False

```

```

120
121
122 def ip_range(ip: str, network_bits: int) -> Set[str]:
123     """
124     Takes a Classless Inter Domain Routing(CIDR) address subnet
125     specification and returns the list of addresses specified
126     by the IP/network bits format.
127     If the number of network bits is not between 0 and 32 it raises an
128     error.
129     If the IP address is invalid according to is_valid_ip it raises an
130     error.
131     """
132     if not 0 <= network_bits <= 32:
133         raise ValueError(f"Invalid number of network bits:
134             [{network_bits}]")
135
136     if not is_valid_ip(ip):
137         raise ValueError(f"Invalid IP address: [{ip}]")
138     # get the ip as long form which is useful
139     # later on for using bitwise operators
140     # to isolate only the constant(network) bits
141     ip_long = dot_to_long(ip)
142
143     # generate the bit mask which specifies
144     # which bits to keep and which to discard
145     mask = int(
146         f"{ '1'*network_bits:0<32s}",
147         base=2
148     )
149     lower_bound = ip_long & mask
150     upper_bound = ip_long | (mask ^ 0xFFFFFFFF)
151
152     # turn all the long form IP addresses between
153     # the lower and upper bound into dot form
154     return set(
155         long_to_dot(long_ip)
156         for long_ip in
157         range(lower_bound+1, upper_bound)
158     )
159
160 def get_local_ip() -> str:
161     """
162     Connects to the google.com with UDP and gets
163     the IP address used to connect(the local address).
164     """
165     with closing(
166         socket.socket(
167             socket.AF_INET,

```

```

167         socket.SOCK_DGRAM
168     )
169 ) as s:
170     s.connect(("google.com", 80))
171     ip, _ = s.getsockname()
172     return ip
173
174
175 def get_free_port() -> int:
176     """
177     Attempts to bind to port 0 which assigns a free port number to the
178     socket,
179     the socket is then closed and the port number assigned is returned.
180     """
181     with closing(
182         socket.socket(
183             socket.AF_INET,
184             socket.SOCK_STREAM
185         )
186     ) as s:
187         s.bind(('', 0))
188         _, port = s.getsockname()
189     return port
190
191
192 def ip_checksum(packet: bytes) -> int:
193     """
194     ip_checksum function takes in a packet
195     and returns the checksum.
196     """
197     if len(packet) % 2 == 1:
198         # if the length of the packet is even, add a NULL byte
199         # to the end as padding
200         packet += b"\0"
201
202     total = 0
203     for first, second in (
204         packet[i:i+2]
205         for i in range(0, len(packet), 2)
206     ):
207         total += (first << 8) + second
208
209     # calculate the number of times a
210     # carry bit was added and add it back on
211     carried = (total - (total & 0xFFFF)) >> 16
212     total &= 0xFFFF
213     total += carried
214
215     if total > 0xFFFF:

```

```

216         # adding the carries generated a carry
217         total &= 0xFFFF
218
219     # invert the checksum and take the last 16 bits.
220     return (~total & 0xFFFF)
221
222
223 def make_icmp_packet(ID: int) -> bytes:
224     """
225     Takes an argument of the process ID of the calling process.
226     Returns an ICMP ECHO REQUEST packet created with this ID
227     """
228
229     ICMP_ECHO_REQUEST = 8
230     # pack the information for the dummy header needed
231     # for the IP checksum
232     dummy_header = struct.pack(
233         "bbHHh",
234         ICMP_ECHO_REQUEST,
235         0,
236         0,
237         ID,
238         1
239     )
240     # pack the current time into a double
241     time_bytes = struct.pack("d", time.time())
242     # define the bytes to repeat in the data section of the packet
243     # this makes the packets easily identifiable in packet captures.
244     bytes_to_repeat_in_data = map(ord, " y33t ")
245     # calculate the number of bytes left for data
246     data_bytes = (192 - struct.calcsize("d"))
247     # first pack the current time into the start of the data section
248     # the pack the identifiable data into the rest
249     data = (
250         time_bytes +
251         bytes(islice(cycle(bytes_to_repeat_in_data), data_bytes))
252     )
253     # get the IP checksum for the dummy header and data
254     # and switch the bytes into the order expected by the network
255     checksum = socket.htons(ip_checksum(dummy_header + data))
256     # pack the header with the correct checksum and information
257     header = struct.pack(
258         "bbHHh",
259         ICMP_ECHO_REQUEST,
260         0,
261         checksum,
262         ID,
263         1
264     )
265     # concatonate the header bytes and the data bytes

```



```

266     return header + data
267
268
269 def make_tcp_packet(
270     src: int,
271     dst: int,
272     from_address: str,
273     to_address: str,
274     flags: int) -> bytes:
275     """
276     Takes in the source and destination port/ip address
277     returns a tcp packet.
278     flags:
279     2 => SYN
280     18 => SYN:ACK
281     4 => RST
282     """
283     # validate that the information passed in is valid
284     if flags not in {2, 18, 4}:
285         raise ValueError(
286             f"Flags must be one of 2:SYN, 18:SYN:ACK, 4:RST. not:
287                 [{flags}]"
288         )
289     if not is_valid_ip(from_address):
290         raise ValueError(
291             f"Invalid source IP address: [{from_address}]"
292         )
293     if not is_valid_ip(to_address):
294         raise ValueError(
295             f"Invalid destination IP address: [{to_address}]"
296         )
297     if not is_valid_port_number(src):
298         raise ValueError(
299             f"Invalid source port: [{src}]"
300         )
301     if not is_valid_port_number(dst):
302         raise ValueError(
303             f"Invalid destination port: [{dst}]"
304         )
305     # turn the ip addresses into long form
306     src_addr = dot_to_long(from_address)
307     dst_addr = dot_to_long(to_address)
308
309     seq = ack = urg = 0
310     data_offset = 6 << 4
311     window_size = 1024
312     max_segment_size = (2, 4, 1460)
313     # pack the dummy header needed for the checksum calculation
314     dummy_header = struct.pack(
315         "!HHIIBBHHHBBH",

```

```

315         src,
316         dst,
317         seq,
318         ack,
319         data_offset,
320         flags,
321         window_size,
322         0,
323         urg,
324         *max_segment_size
325     )
326     # pack the psuedo header that is also needed for the checksum
327     # just because TCP and why not
328     psuedo_header = struct.pack(
329         "!IIBBH",
330         src_addr,
331         dst_addr,
332         0,
333         6,
334         len(dummy_header)
335     )
336
337     checksum = ip_checksum(psuedo_header + dummy_header)
338     # pack the final TCP packet with the relevant data and checksum
339     return struct.pack(
340         "!HHIIBBHHHBBH",
341         src,
342         dst,
343         seq,
344         ack,
345         data_offset,
346         flags,
347         window_size,
348         checksum,
349         urg,
350         *max_segment_size
351     )
352
353
354 def make_udp_packet(
355     src: int,
356     dst: int
357 ) -> bytes:
358     """
359     Takes in: source IP address and port, destination IP address and
360         port.
361     Returns: a UDP packet with those properties.
362     the IP addresses are needed for calculating the checksum.
363     """
364     # validate data passed in

```

```

364     if not is_valid_port_number(src):
365         raise ValueError(
366             f"Invalid source port: [{src}]"
367         )
368     if not is_valid_port_number(dst):
369         raise ValueError(
370             f"Invalid destination port: [{dst}]"
371         )
372     data = b"Most services don't respond to an empty data field"
373     # pack the data
374     # and return the packed bytes
375     # UDP checksum is optional over IPv4
376     return struct.pack(
377         "!HHHH",
378         src,
379         dst,
380         8+len(data),
381         0
382     ) + data
383
384
385 def wait_for_socket(sock: socket.socket, wait_time: float) -> float:
386     """
387     Wait for wait_time seconds or until the socket is readable.
388     If the socket is readable return a tuple of the socket and the time
389     taken
390     otherwise return None.
391     """
392     start = time.time()
393     is_socket_readable = select.select([sock], [], [], wait_time)
394     taken = time.time() - start
395     if is_socket_readable[0] == []:
396         return float(-1)
397     else:
398         return taken

```

Listing 24: A python module I made to hold all of the listeners I had made for each of the different scanning types.

```

1  from modules import headers
2  from modules import ip_utils
3  import socket
4  import struct
5  import time
6  from collections import defaultdict
7  from contextlib import closing
8  from typing import Tuple, Set, DefaultDict
9
10

```

```

11 PORTS = DefaultDict[str, Set[int]]
12
13
14 def ping(
15     ID: int,
16     timeout: float
17 ) -> Set[Tuple[str, float, headers.ip]]:
18     """
19     Takes in a process id and a timeout and returns
20     a list of addresses which sent ICMP ECHO REPLY
21     packets with the packed id matching ID in the time given by timeout.
22     """
23     ping_sock = socket.socket(
24         socket.AF_INET,
25         socket.SOCK_RAW,
26         socket.IPPROTO_ICMP)
27     # opens a raw socket for sending ICMP protocol packets
28     time_remaining = timeout
29     addresses = set()
30     recieved_from = set()
31     while True:
32         time_waiting = ip_utils.wait_for_socket(ping_sock,
33         time_remaining)
34         # time_waiting stores the time the socket took to become readable
35         # or returns minus one if it ran out of time
36
37         if time_waiting == -1:
38             break
39         time_recieved = time.time()
40         # store the time the packet was recieved
41         recPacket, addr = ping_sock.recvfrom(1024)
42         # recieve the packet
43         ip = headers.ip(recPacket[:20])
44         # unpack the IP header into its respective components
45         icmp = headers.icmp(recPacket[20:28])
46         # unpack the time from the packet.
47         time_sent = struct.unpack(
48             "d",
49             recPacket[28:28 + struct.calcsize("d")]
50         )[0]
51         # unpack the value for when the packet was sent
52         time_taken: float = time_recieved - time_sent
53         # calculate the round trip time taken for the packet
54         if icmp.id == ID:
55             # if the ping was sent from this machine then add it to the
56             # list of
57             # responses
58             ip_address, port = addr
59             # this is to prevent a bug where IPs were being added twice
60             if ip_address not in recieved_from:

```

```

59         addresses.add((ip_address, time_taken, ip))
60         recieved_from.add(ip_address)
61     elif time_remaining <= 0:
62         break
63     else:
64         continue
65     # return a list of all the addresses that replied to our ICMP echo
66     request.
67     return addresses
68
69 def udp(dest_ip: str, timeout: float) -> Set[int]:
70     """
71     This listener detects UDP packets from dest_ip in the given timespan,
72     all ports that send direct replies are marked as being open.
73     Returns a list of open ports.
74     """
75
76     time_remaining = timeout
77     ports: Set[int] = set()
78     with socket.socket(
79         socket.AF_INET,
80         socket.SOCK_RAW,
81         socket.IPPROTO_UDP
82     ) as s:
83         while True:
84             time_taken = ip_utils.wait_for_socket(s, time_remaining)
85             if time_taken == -1:
86                 break
87             else:
88                 time_remaining -= time_taken
89             packet = s.recv(1024)
90             ip = headers.ip(packet[:20])
91             udp = headers.udp(packet[20:28])
92             if dest_ip == ip.source and ip.protocol == 17:
93                 ports.add(udp.src)
94
95     return ports
96
97
98 def icmp_unreachable(src_ip: str, timeout: float = 2) -> int:
99     """
100     This listener detects ICMP destination unreachable
101     packets and returns the icmp code.
102     This is later used to mark them as either close, open|filtered,
103     filtered.
104     3 -> closed
105     0|1|2|9|10|13 -> filtered
106     -1 -> error with arguments
107     open|filtered means that they are either open or

```

```

107     filtered but return nothing.
108     """
109
110     ping_sock = socket.socket(
111         socket.AF_INET,
112         socket.SOCK_RAW,
113         socket.IPPROTO_ICMP
114     )
115     # open raw socket to listen for ICMP destination unreachable packets
116     time_remaining = timeout
117     code = -1
118     while True:
119         time_waiting = ip_utils.wait_for_socket(ping_sock,
120             time_remaining)
121         # wait for socket to be readable
122         if time_waiting == -1:
123             break
124         else:
125             time_remaining -= time_waiting
126             recPacket, addr = ping_sock.recvfrom(1024)
127             # recieve the packet
128             ip = headers.ip(recPacket[:20])
129             icmp = headers.icmp(recPacket[20:28])
130             valid_codes = [0, 1, 2, 3, 9, 10, 13]
131             if (
132                 ip.source == src_ip
133                 and icmp.type == 3
134                 and icmp.code in valid_codes
135             ):
136                 code = icmp.code
137                 break
138             elif time_remaining <= 0:
139                 break
140             else:
141                 continue
142     ping_sock.close()
143     return code
144
145 def tcp(address: Tuple[str, int], timeout: float) -> PORTS:
146     """
147     This function is run asynchronously and listens for
148     TCP ACK responses to the sent TCP SYN msg.
149     """
150     ports: DefaultDict[str, Set[int]] = defaultdict(set)
151     with closing(
152         socket.socket(
153             socket.AF_INET,
154             socket.SOCK_RAW,
155             socket.IPPROTO_TCP

```

```

156         )) as s:
157     s.bind(address)
158     # bind the raw socket to the listening address
159     time_remaining = timeout
160     while True:
161         time_taken = ip_utils.wait_for_socket(s, time_remaining)
162         # wait for the socket to become readable
163         if time_taken == -1:
164             break
165         else:
166             time_remaining -= time_taken
167         packet = s.recv(1024)
168         # recieve the packet data
169         tcp = headers.tcp(packet[20:40])
170         if tcp.flags & 2: # syn flags set
171             ports["OPEN"].add(tcp.source)
172         elif tcp.flags & 4:
173             ports["CLOSED"].add(tcp.source)
174         else:
175             continue
176     return ports

```

Listing 25: A python module I made to hold all of the scanners I had made for each of the different scanning types.

```

1  import socket
2  import time
3  from modules import directives
4  from modules import headers
5  from modules import ip_utils
6  from modules import listeners
7  from collections import defaultdict
8  from contextlib import closing
9  from itertools import repeat
10 from multiprocessing import Pool
11 from os import getpid
12 from typing import Set, Tuple
13
14
15 def ping(addresses: Set[str]) -> Set[Tuple[str, float, headers.ip]]:
16     """
17     Send an ICMP ECHO REQUEST to each address
18     in the set addresses. Then return a set which
19     contains all the addresses which replied and
20     which have the correct ID.
21     """
22     with closing(
23         socket.socket(
24             socket.AF_INET,
25             socket.SOCK_RAW,

```

```

26         socket.IPPROTO_ICMP
27     )
28 ) as ping_sock:
29     # get the local ip address
30     addresses = {
31         ip
32         for ip in addresses
33         if (
34             not ip.endswith(".0")
35             and not ip.endswith(".255")
36         )
37     }
38
39     # initialise a process pool
40     p = Pool(1)
41     # get the local process id for use in creating packets.
42     ID = getpid() & 0xFFFF
43     # run the listeners.ping function asynchronously
44     replied = p.apply_async(listeners.ping, (ID, 5))
45     time.sleep(0.01)
46     for address in zip(addresses, repeat(1)):
47         try:
48             packet = ip_utils.make_icmp_packet(ID)
49             ping_sock.sendto(packet, address)
50         except PermissionError:
51             ip_utils.eprint("raw sockets require root priveleges,
52                             exiting")
53             exit()
54     p.close()
55     p.join()
56     # close and join the process pool to so that all the values
57     # have been returned and the pool closed
58     return replied.get()
59
60 def connect(address: str, ports: Set[int]) -> Set[int]:
61     """
62     This is the most basic kind of scan
63     it simply connects to every specififed port
64     and identifies whether they are open.
65     """
66     import socket
67     from contextlib import closing
68     open_ports: Set[int] = set()
69     for port in ports:
70         # loop through each port in the list of ports to scan
71         try:
72             with closing(
73                 socket.socket(
74                     socket.AF_INET,

```



```

75         socket.SOCK_STREAM
76     )
77     ) as s:
78         # open an IPV4 TCP socket
79         s.connect((address, port))
80         # attempt to connect the newly created socket to the
            target
81         # address and port
82         open_ports.add(port)
83         # if the connection was successful then add the port to
            the
84         # list of open ports
85     except (ConnectionRefusedError, OSError) as e:
86         pass
87     return open_ports
88
89
90 def tcp(dest_ip: str, portlist: Set[int]) -> listeners.PORTS:
91     src_port = ip_utils.get_free_port()
92     # request a local port to connect from
93     if "127.0.0.1" == dest_ip:
94         local_ip = "127.0.0.1"
95     else:
96         local_ip = ip_utils.get_local_ip()
97     p = Pool(1)
98     listener = p.apply_async(listeners.tcp, ((local_ip, src_port), 5))
99     time.sleep(0.01)
100    # start the TCP ACK listener in the background
101    for port in portlist:
102        # flag = 2 for syn scan
103        packet = ip_utils.make_tcp_packet(
104            src_port,
105            port,
106            local_ip,
107            dest_ip,
108            2
109        )
110        with closing(
111            socket.socket(
112                socket.AF_INET,
113                socket.SOCK_RAW,
114                socket.IPPROTO_TCP
115            )
116        ) as s:
117            s.sendto(packet, (dest_ip, port))
118            # send the packet to its destination
119    p.close()
120    p.join()
121    ports = listener.get()
122    ports["FILTERED"] = portlist - ports["OPEN"] - ports["CLOSED"]

```

```

123     if local_ip == "127.0.0.1":
124         ports["OPEN"] -= set([src_port])
125
126     return ports
127
128
129 def udp(
130     dest_ip: str,
131     ports_to_scan: Set[int]
132 ) -> listeners.PORTS:
133     """
134     Takes in a destination IP address in either dot or long form and
135     a list of ports to scan. Sends UDP packets to each port specified
136     in portlist and uses the listeners to mark them as open,
137     open|filtered,
138     filtered, closed they are marked open|filtered if no response is
139     recieved at all.
140     """
141
142     local_port = ip_utils.get_free_port()
143     # get port number
144     ports: listeners.PORTS = defaultdict(set)
145     ports["REMAINING"] = ports_to_scan
146     p = Pool(1)
147     udp_listen = p.apply_async(listeners.udp, (dest_ip, 4))
148     time.sleep(0.01)
149     # start the UDP listener
150     with closing(
151         socket.socket(
152             socket.AF_INET,
153             socket.SOCK_RAW,
154             socket.IPPROTO_UDP
155         )
156     ) as s:
157         for _ in range(2):
158             # repeat 3 times because UDP scanning comes
159             # with a high chance of packet loss
160             for dest_port in ports["REMAINING"]:
161                 try:
162                     packet = ip_utils.make_udp_packet(
163                         local_port,
164                         dest_port
165                     )
166                     # create the UDP packet to send
167                     s.sendto(packet, (dest_ip, dest_port))
168                     # send the packet to the currently scanning address
169                 except socket.error:
170                     packet_bytes = " ".join(map(hex, packet))
171                     print(
172                         "The socket modules sendto method with the

```

```

172         following",
173         "argument resulting in a socket error.",
174         f"\npacket: [{packet_bytes}]\n",
175         "address: [{dest_ip, dest_port}]]"
176     )
177 p.close()
178 p.join()
179
180 ports["OPEN"].update(udp_listen.get())
181 # if we are on localhost remove the scanning port
182 if dest_ip == "127.0.0.1":
183     ports["OPEN"] -= set([local_port])
184 ports["REMAINING"] -= ports["OPEN"]
185 # only scan the ports which we know are not open
186 with closing(
187     socket.socket(
188         socket.AF_INET,
189         socket.SOCK_RAW,
190         socket.IPPROTO_UDP
191     )
192 ) as s:
193     for dest_port in ports["REMAINING"]:
194         try:
195             packet = ip_utils.make_udp_packet(
196                 local_port,
197                 dest_port
198             )
199             # make a new UDP packet
200             p = Pool(1)
201             icmp_listen = p.apply_async(
202                 listeners.icmp_unreachable,
203                 (dest_ip,),
204             )
205             # start the ICMP listener
206             time.sleep(0.01)
207             s.sendto(packet, (dest_ip, dest_port))
208             # send packet
209             p.close()
210             p.join()
211             icmp_code = icmp_listen.get()
212             # receive ICMP code from the ICMP listener
213             if icmp_code in {0, 1, 2, 9, 10, 13}:
214                 ports["FILTERED"].add(dest_port)
215             elif icmp_code == 3:
216                 ports["CLOSED"].add(dest_port)
217         except socket.error:
218             packet_bytes = " ".join(map("{:02x}".format, packet))
219             ip_utils.eprint(
220                 "The socket modules sendto method with the following",

```

```

221         "argument resulting in a socket error.",
222         f"\npacket: [{packet_bytes}]\n",
223         "address: [{dest_ip, dest_port}]]"
224     )
225     # this creates a new set which contains all the elements that
226     # are in the list of ports to be scanned but have not yet
227     # been classified
228     ports["OPEN|FILTERED"] = (
229         ports["REMAINING"]
230         - ports["OPEN"]
231         - ports["FILTERED"]
232         - ports["CLOSED"]
233     )
234     del(ports["REMAINING"])
235     # set comprehension to update the list of open filtered ports
236     return ports
237
238
239 def version_detect_scan(
240     target: directives.Target,
241     probes: directives.PROBE_CONTAINER
242 ) -> directives.Target:
243     for probe_dict in probes.values():
244         for proto in probe_dict:
245             target = probe_dict[proto].scan(target)
246     return target

```

6.8 examples

Listing 26: A program I wrote to run all of the example scripts I made from one main script to solve the issue of the PATH being used for determining import when I could use Pythons built in module structure instead.

```

1  #!/usr/bin/env python
2  from icmp_ping import icmp_echo_recv, icmp_echo_send
3  from ping_scanner import ping_scan
4  from tcp_scan.connect_scan import scan_port_list as connect_scan_list
5  from tcp_scan.syn_scan import scan_port_list as syn_scan_list
6  from udp_scan import scan_port_list as udp_scan_list
7  from version_detection import version_detection
8
9  examples = {
10     "icmp_echo_recv": icmp_echo_recv.main,
11     "icmp_echo_send": icmp_echo_send.main,
12     "ping_scanner": ping_scan.main,
13     "connect_scan": connect_scan_list.main,
14     "syn_scan": syn_scan_list.main,
15     "udp_scan": udp_scan_list.main,
16     "version_detection": version_detection.main,

```

```

17 }
18
19 print("\n\t".join(("Programs:", *examples)))
20
21 while True:
22     print()
23     program = input("Enter the name of the example program to run: ")
24     if program.lower() in {"quit", "q", "end", "exit"}:
25         break
26     found = False
27     for name in examples:
28         if name.startswith(program.lower()):
29             program = name
30             print(f"Running: {program}")
31             examples[program]()
32             found = True
33     if not found:
34         print(
35             "The program name must exactly match one of the following
36             examples"
37         )
38     print("\n".join(examples))

```

6.9 netscan

Listing 27: The program which provides the command line user interface for my projects functionality.

```

1  #!/usr/bin/env python
2  import re
3  from argparse import ArgumentParser
4  from collections import defaultdict
5  from math import floor, log10
6  from modules import (
7      scanners,
8      ip_utils,
9      directives,
10 )
11 from typing import (
12     DefaultDict,
13     Dict,
14 )
15
16 top_ports = directives.parse_ports(open("top_ports").read())
17 services: DefaultDict[str, Dict[int, str]] = defaultdict(dict)
18 for match in re.finditer(
19     r"(\S+)\s+(\d+)/(\S+)",
20     open("version_detection/nmap-services").read()
21 ):

```

```

22     service, portnum, protocol = match.groups()
23     services[protocol.upper()][int(portnum)] = service
24
25     parser = ArgumentParser()
26     parser.add_argument(
27         "target_spec",
28         help="specify what to scan, i.e. 192.168.1.0/24"
29     )
30     parser.add_argument(
31         "-Pn",
32         help="assume hosts are up",
33         action="store_true"
34     )
35     parser.add_argument(
36         "-sL",
37         help="list targets",
38         action="store_true"
39     )
40     parser.add_argument(
41         "-sn",
42         help="disable port scanning",
43         action="store_true"
44     )
45     parser.add_argument(
46         "-sS",
47         help="TCP SYN scan",
48         action="store_true"
49     )
50     parser.add_argument(
51         "-sT",
52         help="TCP connect scan",
53         action="store_true"
54     )
55     parser.add_argument(
56         "-sU",
57         help="UDP scan",
58         action="store_true"
59     )
60     parser.add_argument(
61         "-sV",
62         help="version scan",
63         action="store_true"
64     )
65     parser.add_argument(
66         "-p",
67         "--ports",
68         help="scan specified ports",
69         required=False,
70         default=top_ports
71     )

```

```

72 parser.add_argument(
73     "--exclude_ports",
74     help="ports to exclude from the scan",
75     required=False,
76     default=""
77 )
78
79 args = parser.parse_args()
80
81 # check whether the address spec is in CIDR form
82 CIDR_regex =
83     re.compile(r"(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/(\d{1,2})")
84 search = CIDR_regex.search(args.target_spec)
85 if search:
86     base_addr, network_bits = search.groups()
87     addresses = ip_utils.ip_range(
88         base_addr,
89         int(network_bits)
90     )
91 else:
92     base_addr = args.target_spec
93     addresses = {base_addr}
94
95 def error_exit(error_type: str, scan_type: str, scanning: str) -> bool:
96     messages = {
97         "permission": "\n".join((
98             "You have insufficient permissions to run this type of scan",
99             "EXITING!"
100         ))
101     }
102     print(f"You tried to scan {scanning} using scan type: {scan_type}")
103     try:
104         print(messages[error_type])
105     except KeyError:
106         print(f"ERROR MESSAGE NOT FOUND: {error_type}")
107     exit(-1)
108
109
110 if args.sL:
111     print("Targets:")
112     print("\n".join(sorted(addresses, key=ip_utils.dot_to_long)))
113 else:
114     if args.sn:
115         def sig_figs(x: float, n: int) -> float:
116             """
117             rounds x to n significant figures.
118             sig_figs(1234, 2) = 1200.0
119             """
120             return round(x, n - (1 + int(floor(log10(abs(x))))))

```

```

121
122     try:
123         print("\n".join(
124             f"host: [{host}]\t" +
125             "responded to an ICMP ECHO REQUEST in " +
126             f"{str(sig_figs(taken, 2))+ 's':<10s} " +
127             f"ttl: [{ip_head.time_to_live}]"
128             for host, taken, ip_head in scanners.ping(addresses)
129         ))
130     except PermissionError:
131         error_exit("permission", "ping scan", str(addresses))
132
133 else:
134     if args.Pn:
135         targets = [
136             directives.Target(
137                 addr,
138                 defaultdict(set),
139                 defaultdict(set)
140             )
141             for addr in addresses
142         ]
143     else:
144         try:
145             targets = [
146                 directives.Target(
147                     addr,
148                     defaultdict(set),
149                     defaultdict(set),
150                 )
151                 for addr, _, _ in scanners.ping(addresses)
152             ]
153         except PermissionError:
154             error_exit("permission", "ping_scan", str(addresses))
155     # define the ports to scan
156     if args.ports == "-":
157         # case they have specified all ports
158         ports = {
159             "UDP": set(range(1, 65536)),
160             "TCP": set(range(1, 65536)),
161         }
162     elif isinstance(args.ports, str):
163         # case they have specified ports
164         ports = directives.parse_ports(args.ports)
165     else:
166         # default
167         ports = args.ports
168
169     # exclude all the ports specified to be excluded
170     to_exclude = directives.parse_ports(args.exclude_ports)

```



```

171     ports["TCP"] -= to_exclude["TCP"]
172     ports["TCP"] -= to_exclude["ANY"]
173     ports["UDP"] -= to_exclude["UDP"]
174     ports["UDP"] -= to_exclude["ANY"]
175
176     # if version scanning is desired
177     if args.sV:
178         probes = directives.parse_probes(
179             "./version_detection/nmap-service-probes"
180         )
181
182     for target in targets:
183         if not args.sU and not args.sT or args.sS:
184             try:
185                 tcp_ports = scanners.tcp(
186                     target.address,
187                     ports["TCP"] | ports["ANY"]
188                 )
189             except PermissionError:
190                 error_exit("permission", "tcp_scan", target.address)
191             target.open_ports["TCP"].update(tcp_ports["OPEN"])
192             target.open_filtered_ports["TCP"].update(tcp_ports["FILTERED"])
193         if args.sT:
194             target.open_ports["TCP"].update(
195                 scanners.connect(
196                     target.address,
197                     ports["TCP"] | ports["ANY"]
198                 )
199             )
200         if args.sU:
201             try:
202                 udp_ports = scanners.udp(
203                     target.address,
204                     ports["UDP"] | ports["ANY"]
205                 )
206             except PermissionError:
207                 error_exit("permission", "udp_scan", target.address)
208
209             target.open_ports["UDP"].update(
210                 udp_ports["OPEN"]
211             )
212             target.open_filtered_ports["UDP"].update(
213                 udp_ports["FILTERED"]
214             )
215             target.open_filtered_ports["UDP"].update(
216                 udp_ports["OPEN|FILTERED"]
217             )
218         if args.sV:
219             target = scanners.version_detect_scan(target, probes)
220         # display scan info

```

```

221     print()
222     print(f"Scan report for: {target.address}")
223     # print(target)
224     print("Open ports:")
225     for proto, open_ports in target.open_ports.items():
226         for port in open_ports:
227             try:
228                 service_name = services[proto][port]
229             except KeyError:
230                 service_name = "unknown"
231         if port in target.services:
232             exact_match = target.services[port]
233             print(
234                 f"{port}/{proto}{exact_match.service:>8s}"
235             )
236             # print version information
237             for key, val in exact_match.version_info.items():
238                 print(f"{key}: {val}")
239             if exact_match.cpes:
240                 print()
241                 print("CPE:")
242                 for cpe_type, cpe_vals in
243                     exact_match.cpes.items():
244                     print(cpe_type)
245                     try:
246                         del(cpe_vals["part"])
247                     except KeyError:
248                         pass
249                     for key, val in cpe_vals.items():
250                         print(f"{key}: {val}")
251                 print()
252             else:
253                 print(f"{port} service: {service_name}?")
254
255     print("Filtered ports:")
256     for proto, filtered_ports in
257         target.open_filtered_ports.items():
258         for port in filtered_ports:
259             try:
260                 service_name = services[proto][port]
261             except KeyError:
262                 service_name = "unknown"
263             print(f"{port} service: {service_name}?")

```

6.10 tests

Listing 28: Unit tests I wrote for the ip_utils module.

```

1 from modules.ip_utils import (

```

```

2     dot_to_long,
3     long_to_dot,
4     ip_range,
5     is_valid_ip,
6     is_valid_port_number,
7     ip_checksum,
8     make_tcp_packet,
9     make_udp_packet,
10    make_icmp_packet,
11 )
12 from binascii import unhexlify
13
14
15 def test_dot_to_long_private_ip() -> None:
16     assert(dot_to_long("192.168.1.0") == 0xC0A80100)
17
18
19 def test_long_to_dot_private_ip() -> None:
20     assert(long_to_dot(0xC0A80100) == "192.168.1.0")
21
22
23 def test_dot_to_long_localhost() -> None:
24     assert(dot_to_long("127.0.0.1") == 0x7F000001)
25
26
27 def test_long_to_dot_localhost() -> None:
28     assert(long_to_dot(0x7F000001) == "127.0.0.1")
29
30
31 def test_is_valid_ip_localhost_long() -> None:
32     assert is_valid_ip(0x7F000001)
33
34
35 def test_is_valid_ip_localhost() -> None:
36     assert is_valid_ip("127.0.0.1")
37
38
39 def test_is_not_valid_ip_5_zeros_dotted() -> None:
40     assert not is_valid_ip("0.0.0.0.0")
41
42
43 def test_is_not_valid_ip_5_255s_long() -> None:
44     assert not is_valid_ip(0xFF_FF_FF_FF_FF)
45
46
47 def test_is_valid_port_number_0() -> None:
48     assert is_valid_port_number(0)
49
50
51 def test_is_valid_port_number_65535() -> None:

```

```

52     assert is_valid_port_number(65535)
53
54
55     def test_is_not_valid_port_number_negative_one() -> None:
56         assert not is_valid_port_number(-1)
57
58
59     def test_is_not_valid_port_number_65536() -> None:
60         assert not is_valid_port_number(65536)
61
62
63     def test_ip_range() -> None:
64         assert(
65             ip_range("192.168.1.0", 28) == {
66                 "192.168.1.1",
67                 "192.168.1.2",
68                 "192.168.1.3",
69                 "192.168.1.4",
70                 "192.168.1.5",
71                 "192.168.1.6",
72                 "192.168.1.7",
73                 "192.168.1.8",
74                 "192.168.1.9",
75                 "192.168.1.10",
76                 "192.168.1.11",
77                 "192.168.1.12",
78                 "192.168.1.13",
79                 "192.168.1.14",
80             }
81         )
82
83
84     def test_ip_checksum_verify() -> None:
85         packet = unhexlify(
86             "45000073000040004011b861c0a80001c0a800c7"
87         )
88         assert ip_checksum(packet) == 0
89
90
91     def test_ip_checksum_generate() -> None:
92         packet = unhexlify(
93             "450000730000400040110000c0a80001c0a800c7"
94         )
95         assert ip_checksum(packet) == 0xB861
96
97
98     def test_make_tcp_packet() -> None:
99         correct = unhexlify(
100             "e54700500000000000000000600204002af50000020405b4"
101         )

```

```

102     info = 58695, 80, "192.168.1.45", "192.168.1.28", 2
103     assert correct == make_tcp_packet(*info)
104
105
106 def test_make_udp_packet() -> None:
107     correct = unhexlify(
108         "e5470050003a0000"
109     )
110     info = 58695, 80
111     # clipping the packet at 8 simply removes the data section
112     assert correct == make_udp_packet(*info)[:8]

```

Listing 29: Unit tests I wrote for the directives module.

```

1 from modules.directives import (
2     parse_ports
3 )
4 from collections import defaultdict
5 from typing import DefaultDict
6
7
8 def test_parse_probes_single() -> None:
9     portstring = "12345"
10    expected: DefaultDict[str, set] = defaultdict(set)
11    expected["ANY"] = set([12345])
12    assert expected == parse_ports(portstring)
13
14
15 def test_parse_probes_range() -> None:
16     portstring = "10-20"
17     expected: DefaultDict[str, set] = defaultdict(set)
18     expected["ANY"] = set(range(10, 21))
19     assert expected == parse_ports(portstring)
20
21
22 def test_parse_probes_single_and_range() -> None:
23     portstring = "1,2,3,10-20,6,7,8"
24     expected: DefaultDict[str, set] = defaultdict(set)
25     expected["ANY"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
26     assert expected == parse_ports(portstring)
27
28
29 def test_parse_probes_tcp_single() -> None:
30     portstring = "T:12345"
31     expected: DefaultDict[str, set] = defaultdict(set)
32     expected["TCP"] = set([12345])
33     assert expected == parse_ports(portstring)
34
35
36 def test_parse_probes_tcp_range() -> None:

```

```

37     portstring = "T:10-20"
38     expected: DefaultDict[str, set] = defaultdict(set)
39     expected["TCP"] = set(range(10, 21))
40     assert expected == parse_ports(portstring)
41
42
43 def test_parse_probes_tcp_single_and_range() -> None:
44     portstring = "T:1,2,3,10-20,6,7,8"
45     expected: DefaultDict[str, set] = defaultdict(set)
46     expected["TCP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
47     assert expected == parse_ports(portstring)
48
49
50 def test_parse_probes_udp_single() -> None:
51     portstring = "U:12345"
52     expected: DefaultDict[str, set] = defaultdict(set)
53     expected["UDP"] = set([12345])
54     assert expected == parse_ports(portstring)
55
56
57 def test_parse_probes_udp_range() -> None:
58     portstring = "U:10-20"
59     expected: DefaultDict[str, set] = defaultdict(set)
60     expected["UDP"] = set(range(10, 21))
61     assert expected == parse_ports(portstring)
62
63
64 def test_parse_probes_udp_single_and_range() -> None:
65     portstring = "U:1,2,3,10-20,6,7,8"
66     expected: DefaultDict[str, set] = defaultdict(set)
67     expected["UDP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
68     assert expected == parse_ports(portstring)
69
70
71 def test_parse_probes_any_and_tcp_single() -> None:
72     portstring = "12345 T:12345"
73     expected: DefaultDict[str, set] = defaultdict(set)
74     expected["TCP"] = set([12345])
75     expected["ANY"] = set([12345])
76     assert expected == parse_ports(portstring)
77
78
79 def test_parse_probes_any_and_tcp_range() -> None:
80     portstring = "10-20 T:10-20"
81     expected: DefaultDict[str, set] = defaultdict(set)
82     expected["TCP"] = set(range(10, 21))
83     expected["ANY"] = set(range(10, 21))
84     assert expected == parse_ports(portstring)
85
86

```

```

87 def test_parse_probes_any_and_tcp_single_and_range() -> None:
88     portstring = "1,2,3,10-20,6,7,8 T:1,2,3,10-20,6,7,8"
89     expected: DefaultDict[str, set] = defaultdict(set)
90     expected["TCP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
91     expected["ANY"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
92     assert expected == parse_ports(portstring)
93
94
95 def test_parse_probes_any_and_udp_single() -> None:
96     portstring = "12345 U:12345"
97     expected: DefaultDict[str, set] = defaultdict(set)
98     expected["UDP"] = set([12345])
99     expected["ANY"] = set([12345])
100    assert expected == parse_ports(portstring)
101
102
103 def test_parse_probes_any_and_udp_range() -> None:
104     portstring = "10-20 U:10-20"
105     expected: DefaultDict[str, set] = defaultdict(set)
106     expected["UDP"] = set(range(10, 21))
107     expected["ANY"] = set(range(10, 21))
108     assert expected == parse_ports(portstring)
109
110
111 def test_parse_probes_any_and_udp_single_and_range() -> None:
112     portstring = "1,2,3,10-20,6,7,8 U:1,2,3,10-20,6,7,8"
113     expected: DefaultDict[str, set] = defaultdict(set)
114     expected["UDP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
115     expected["ANY"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
116     assert expected == parse_ports(portstring)
117
118
119 def test_parse_probes_udp_and_tcp_single() -> None:
120     portstring = "U:12345 T:12345"
121     expected: DefaultDict[str, set] = defaultdict(set)
122     expected["TCP"] = set([12345])
123     expected["UDP"] = set([12345])
124     assert expected == parse_ports(portstring)
125
126
127 def test_parse_probes_udp_and_tcp_range() -> None:
128     portstring = "U:10-20 T:10-20"
129     expected: DefaultDict[str, set] = defaultdict(set)
130     expected["TCP"] = set(range(10, 21))
131     expected["UDP"] = set(range(10, 21))
132     assert expected == parse_ports(portstring)
133
134
135 def test_parse_probes_udp_and_tcp_single_and_range() -> None:
136     portstring = "U:1,2,3,10-20,6,7,8 T:1,2,3,10-20,6,7,8"

```

```

137     expected: DefaultDict[str, set] = defaultdict(set)
138     expected["TCP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
139     expected["UDP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
140     assert expected == parse_ports(portstring)
141
142
143     def test_parse_probes_all_single() -> None:
144         portstring = "12345 U:12345 T:12345"
145         expected: DefaultDict[str, set] = defaultdict(set)
146         expected["TCP"] = set([12345])
147         expected["UDP"] = set([12345])
148         expected["ANY"] = set([12345])
149         assert expected == parse_ports(portstring)
150
151
152     def test_parse_probes_all_range() -> None:
153         portstring = "10-20 U:10-20 T:10-20"
154         expected: DefaultDict[str, set] = defaultdict(set)
155         expected["TCP"] = set(range(10, 21))
156         expected["UDP"] = set(range(10, 21))
157         expected["ANY"] = set(range(10, 21))
158         assert expected == parse_ports(portstring)
159
160
161     def test_parse_probes_all_single_and_range() -> None:
162         portstring = "1,2,3,10-20,6,7,8 U:1,2,3,10-20,6,7,8
163                     T:1,2,3,10-20,6,7,8"
164         expected: DefaultDict[str, set] = defaultdict(set)
165         expected["TCP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
166         expected["UDP"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
167         expected["ANY"] = set([1, 2, 3, *range(10, 21), 6, 7, 8])
168         assert expected == parse_ports(portstring)

```

Glossary

API Applications Programming Interface 4, 27

ARP Address Resolution Protocol 51, 52

banner A short piece of text which a service with send to identify itself when it receives a connection request. Often contains information such as version number etc... 24

black box Looking at something from an outsider's perspective knowing nothing about how it works internally. 2, 18

checksum A checksum is a value calculated from a mathematical algorithm which is sent with the packet to its destination to allow the recipient to check whether the packet was corrupted on the way. 19, 37

CIDR Classless Inter-Domain Routing 18, 24, 45

CPE Common Platform Enumeration 37, 58

daemon A process that runs forever in the background to facilitate other programs. 2, 3

dbus-daemon A daemon which enable a common interface for inter-process communication. 3

DHCP Dynamic Host Configuration Protocol 2, 3, 4

DHCPD Dynamic Host Configuration Protocol Client Daemon 3

DNS Domain Name System 22

driver A tiny software module which is loaded into the kernel when the computer boots up, They mainly interface with hardware and are often very specific for each piece of hardware. 2, 3

FTP File Transfer Protocol 4, 19

header A header is the first few bytes at the start of a packet often consisting of information on where to send the packet next, can also contain information though. 5

HTML Hypertext Markup Language 5, 7

HTTP Hypertext transfer Protocol 4, 5, 6, 16

HTTPS Hypertext transfer Protocol Secure 16

ICMP Internet Control Message Protocol 17, 18, 26, 27, 28, 31, 32, 33, 40, 43, 51, 56, 57

IDS Intrusion Detection System 19

IP Internet Protocol 33

IP address Every computer on a network has a unique IP address assigned to them, which is used to identify where exactly message sent by computers are meant to go. 2, 3, 5, 16, 45

kernel The kernel is the foundation of an operating system and it serves as the main interface between the software running on the system and the underlying hardware it performs task such as processor scheduling and managing input/output operations. 2, 3

MAC Media Access Control 51

NIC Network Interface Card 2, 4, 51

OSI model Open Systems Interconnection model 4, 27

packet Packets are simply a list of bytes which contains packed values such as to and from address and they are the basis for almost all inter-computer communications. 3, 5, 6, 8, 11, 12, 16, 17, 19, 37, 38

PCAP Packet CAPture 36

PHP PHP Hypertext Processor 4

port Computers have “ports” for each protocol which can be connected to separately, this makes up part of a “socket” connection. 5, 6, 18, 19, 38, 45, 46

port knocking Port knocking is where packets must be sent to a sequence of ports before access to the desired port is granted. 19

SCTP Stream Control Transmission Protocol 19

server A server is any computer which it’s purpose is to provide resources to others, either humans or other computers for purposes from hosting website or just as a resource of large computational power. 3, 24

service A service is something running on a machine that offers a service to either other programs on the computer or to people on the internet. 2, 12, 19, 24, 37, 38

SSH Secure SHell 58, 59

subnet A subnet is simply the sub-network of every possible IP address that will be used for communication on a particular network. 3, 45

systemd A daemon for controlling what is run when the system starts. 3

TCP Transmission Control Protocol 5, 6, 12, 15, 16, 17, 18, 19, 26, 33, 38, 42, 51, 53, 54, 55, 58, 59

UDP User Datagram Protocol 6, 17, 19, 26, 43, 56, 57, 58

upowerd Manages the power supplied to the system: charging, battery usage etc... 3

XML eXtensible Markup Language 21