A Level Computer Science Non-Examined Assessment (NEA)

Sam Leonard

Contents

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 (DHCPCD) 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

4

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.

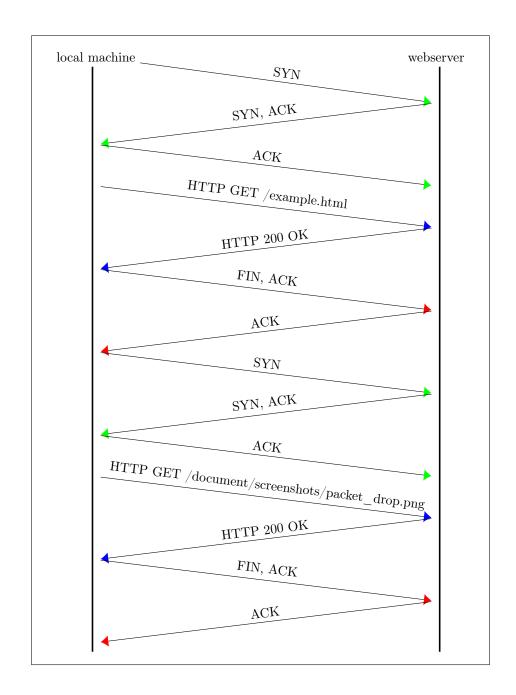Figure 5: A full chain of packets that shows retrieving a basic webpage from the server.

57790 | 57790 → 8000 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=2406037769 T... | 8000
57790 | 8000 → 57790 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=30... | 8000
57790 | 57790 → 8000 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=2406037770 TSecr=3055335585 | 8000
57790 | GET /example.html HTTP/1.1 | 8000
57790 | 8000 → 57790 [ACK] Seq=1 Ack=363 Win=64896 Len=0 TSval=3055335586 TSecr=2406037770 | 8000
57790 | 8000 → 57790 [PSH, ACK] Seq=1 Ack=363 Win=64896 Len=185 TSval=3055335588 TSecr=240... | 8000
57790 | 57790 → 8000 [ACK] Seq=363 Ack=186 Win=30336 Len=0 TSval=2406037772 TSecr=3055335... | 8000
57790 | HTTP/1.0 200 OK (text/html) | 8000
57790 | 57790 → 8000 [ACK] Seq=363 Ack=779 Win=31488 Len=0 TSval=2406037772 TSecr=3055335... | 8000
57790 | 57790 → 8000 [FIN, ACK] Seq=363 Ack=779 Win=31488 Len=0 TSval=2406037773 TSecr=3055... | 8000
57790 | 8000 → 57790 [ACK] Seq=779 Ack=364 Win=64896 Len=0 TSval=3055335589 TSecr=2406037... | 8000
57792 | 57792 → 8000 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=2406037796 T... | 8000
57792 | 8000 → 57792 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=30... | 8000
57792 | 57792 → 8000 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=2406037796 TSecr=3055335612 | 8000
57792 | GET /document/screenshots/packet_drop.png HTTP/1.1 | 8000
57792 | 8000 → 57792 [ACK] Seq=1 Ack=432 Win=64768 Len=0 TSval=3055335612 TSecr=2406037797 | 8000
57792 | 8000 → 57792 [PSH, ACK] Seq=1 Ack=432 Win=64768 Len=187 TSval=3055335615 TSecr=240... | 8000
57792 | 57792 → 8000 [ACK] Seq=432 Ack=188 Win=30336 Len=0 TSval=2406037799 TSecr=3055335... | 8000
57792 | 8000 → 57792 [ACK] Seq=188 Ack=432 Win=64768 Len=4344 TSval=3055335615 TSecr=2406... | 8000
57792 | 57792 → 8000 [ACK] Seq=432 Ack=4532 Win=39040 Len=0 TSval=2406037799 TSecr=305533... | 8000
57792 | 8000 → 57792 [ACK] Seq=4532 Ack=432 Win=64768 Len=5792 TSval=3055335615 TSecr=240... | 8000
57792 | 57792 → 8000 [ACK] Seq=432 Ack=10324 Win=50560 Len=0 TSval=2406037799 TSecr=30553... | 8000
57792 | 8000 → 57792 [ACK] Seq=10324 Ack=432 Win=64768 Len=2896 TSval=3055335615 TSecr=24... | 8000
57792 | 57792 → 8000 [ACK] Seq=432 Ack=13220 Win=56448 Len=0 TSval=2406037799 TSecr=30553... | 8000
57792 | 8000 → 57792 [ACK] Seq=13220 Ack=432 Win=64768 Len=2896 TSval=3055335615 TSecr=24... | 8000
57792 | 57792 → 8000 [ACK] Seq=432 Ack=16116 Win=62208 Len=0 TSval=2406037799 TSecr=30553... | 8000
57792 | 8000 → 57792 [ACK] Seq=16116 Ack=432 Win=64768 Len=5792 TSval=3055335615 TSecr=24... | 8000
57792 | 57792 → 8000 [ACK] Seq=432 Ack=21908 Win=73728 Len=0 TSval=2406037799 TSecr=30553... | 8000
57792 | 8000 → 57792 [ACK] Seq=21908 Ack=432 Win=64768 Len=4344 TSval=3055335615 TSecr=24... | 8000
57792 | 57792 → 8000 [ACK] Seq=432 Ack=26252 Win=82432 Len=0 TSval=2406037800 TSecr=30553... | 8000
57792 | 8000 → 57792 [ACK] Seq=26252 Ack=432 Win=64768 Len=5792 TSval=3055335615 TSecr=24... | 8000
57792 | 57792 → 8000 [ACK] Seq=432 Ack=32044 Win=94080 Len=0 TSval=2406037800 TSecr=30553... | 8000
57792 | 8000 → 57792 [ACK] Seq=32044 Ack=432 Win=64768 Len=5792 TSval=3055335615 TSecr=24... | 8000
57792 | 57792 → 8000 [ACK] Seq=432 Ack=37836 Win=105600 Len=0 TSval=2406037800 TSecr=3055... | 8000
57792 | HTTP/1.0 200 OK (PNG) | 8000
57792 | 57792 → 8000 [ACK] Seq=432 Ack=42612 Win=115200 Len=0 TSval=2406037800 TSecr=3055... | 8000
57792 | 57792 → 8000 [FIN, ACK] Seq=432 Ack=42612 Win=115200 Len=0 TSval=2406037800 TSecr=3... | 8000
57792 | 8000 → 57792 [ACK] Seq=42612 Ack=433 Win=64768 Len=0 TSval=3055335616 TSecr=24060... | 8000

Figure 6: Ladder diagram of figure 5.

8

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
```

10

```
24   <img id="img" src="document/screenshots/packet_drop.png">
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 estab-
lished 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 con-
nection 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 con-
nections 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.

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

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

16

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. Send ICMP ECHO requests to determine whether a machine is active or not.

3. Translate Classless Inter-Domain Routing (CIDR) specified subnets into a list of domains.

4. Detect whether a TCP port is open (can be connected to).

5. Detect whether a TCP port is closed (will refuse connections).

6. Detect whether a TCP port is filtered (a firewall is preventing or monitoring access).

7. Detect whether a UDP port is open (can be connected to).

8. Detect whether a UDP port is closed (will refuse connections).

9. Detect whether a UDP port is filtered (a firewall is preventing or monitoring access).

10. Detect the operating system of another machine on the network solely from sending packets to the machine and interpreting the responses.

11. Detect what service is listening behind a port.

12. 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 doingportscanning 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 finalportis 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,gnmap,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\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 lone. 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 than 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\x10\0\x03&quot;)%r
```

```
19  (DNSStatusRequestTCP,E,&quot;\0\x0c\0\0\x90\x04\0\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.
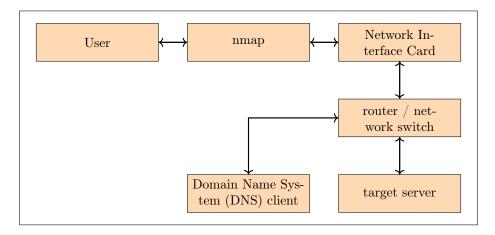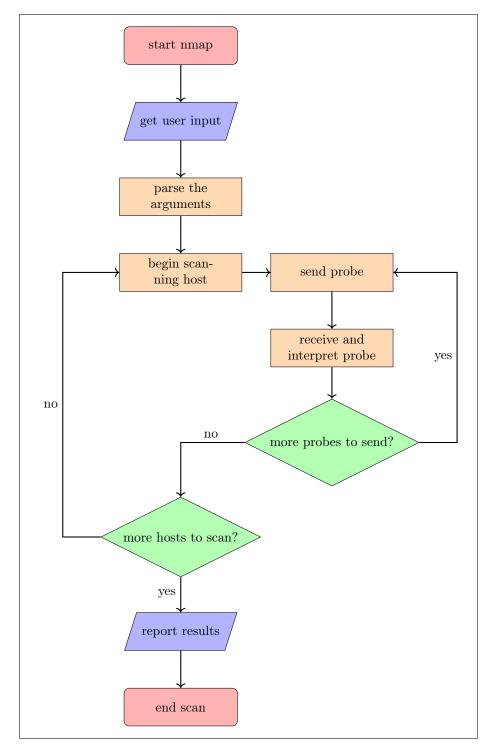

Figure 15: A block diagram showing how nmap sits in the software stack.

21

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

```python
>>> with open("hosts", "r") as f:
...     hosts = f.read().splitlines()
...
>>> import sys
>>> sys.getsizeof(hosts)
2216
>>> ports = list(range(1000))
>>> sys.getsizeof(ports)
9112
>>> len(hosts)*sys.getsizeof(ports) / 2**10 # 2*10 is one kibibyte
2278.0
>>> sys.getsizeof(True)
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 that 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
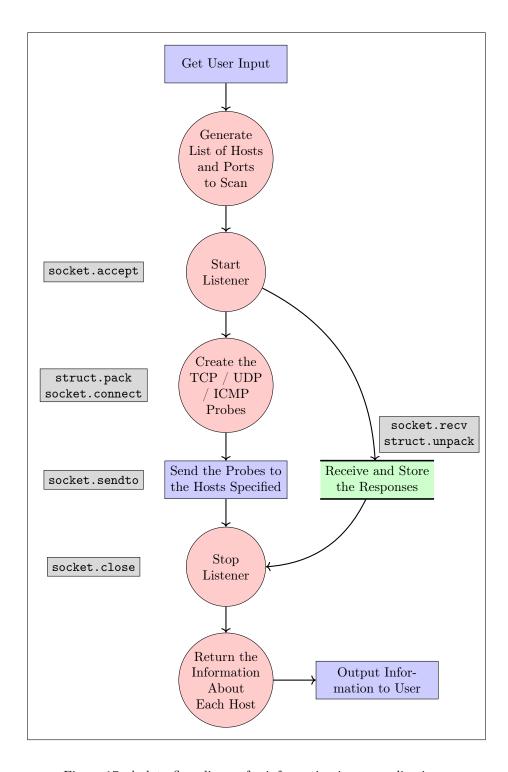
3. how my program sends data around between functions

24

Figure 17: A data flow digram 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 it's 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 figure4 is listening for packets.

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

```python
#!/usr/bin/python3.7
import socket
import struct
import os
import time
import array

from os import getcwd, getpid
import sys
sys.path.append("../modules/")

import ip_utils


ICMP_ECHO_REQUEST = 8

# 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_flgs_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 btyes.

```
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
```

```python
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]
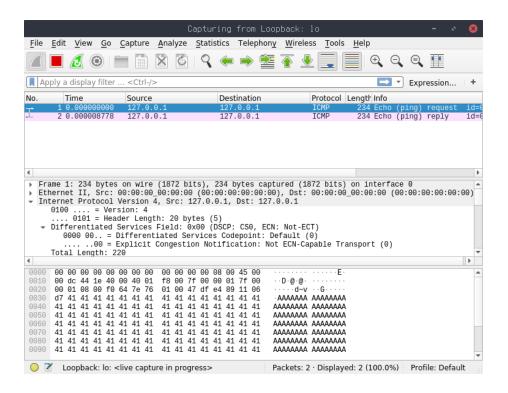```

Figure 18: Dissecting an ICMP ECHO REQUEST packet.

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

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
```

32

```python
from functools import partial
from itertools import repeat
from multiprocessing import Pool
from contextlib import closing
from math import log10, floor
from typing import List, Tuple
import struct
import time


def round_significant_figures(x: float, n: int) -> float:
    """
    rounds x to n significant figures.
    round_significant_figures(1234, 2) = 1200.0
    """
    return round(x, n-(1+int(floor(log10(abs(x))))))


def recieved_ping_from_addresses(ID: int, timeout: float) ->
        List[Tuple[str, float, int]]:
    """
    Takes in a process id and a timeout and returns the list of
        addresses which sent
    ICMP ECHO REPLY packets with the packed id matching ID in the time
        given by timeout.
    """
    ping_sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
        socket.IPPROTO_ICMP)
    time_remaining = timeout
    addresses = []
    while True:
        time_waiting = ip_utils.wait_for_socket(ping_sock,
            time_remaining)
        if time_waiting == -1:
            break
        time_recieved = time.time()
        recPacket, addr = ping_sock.recvfrom(1024)
        ip_header = recPacket[:20]
        ip_hp_ip_v, ip_dscp_ip_ecn, ip_len, ip_id, ip_flgs_ip_off,
            ip_ttl, ip_p, ip_sum, ip_src, ip_dst =
            struct.unpack('!BBHHHBBHII', ip_header)
        icmp_header = recPacket[20:28]
        msg_type, code, checksum, p_id, sequence =
            struct.unpack('bbHHh', icmp_header)
        time_remaining -= time_waiting
        time_sent = struct.unpack("d",
            recPacket[28:28+struct.calcsize("d")])[0]
        time_taken = time_recieved - time_sent
        if p_id == ID:
            addresses.append((str(addr[0]), float(time_taken),
```

```python
                int(ip_ttl)))
        elif time_remaining <= 0:
            break
        else:
            continue
    return addresses


with closing(socket.socket(socket.AF_INET, socket.SOCK_RAW,
    socket.IPPROTO_ICMP)) as ping_sock:
    addresses = ip_utils.ip_range("192.168.1.0/24")
    local_ip = ip_utils.get_local_ip()
    if addresses is not None:
        addresses_to_scan = filter(lambda x: x!=local_ip, addresses)
    else:
        print("error with ip range specification")
        exit()
    p = Pool(1)
    ID = getpid()&0xFFFF
    replied = p.apply_async(recieved_ping_from_addresses, (ID, 2))
    for address in zip(addresses_to_scan, repeat(1)):
        try:
            packet = ip_utils.make_icmp_packet(ID)
            ping_sock.sendto(packet, address)
        except PermissionError:
            pass
    p.close()
    p.join()
    hosts_up = replied.get()
    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)))
```
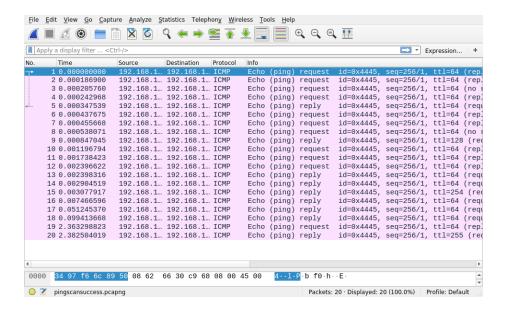
Figure 21: Screenshot of wireshark showing a successful ping scan.

Listing 7: The output of from the ping scanner on the run which generated the PCAP file in figure 21

```
1  $ sudo ./ping_scan.py
2  host: [192.168.1.1]   responded to an ICMP ECHO REQUEST in 0.00037
        seconds, ttl: [64]
3  host: [192.168.1.35] responded to an ICMP ECHO REQUEST in 0.00042
        seconds, ttl: [128]
4  host: [192.168.1.37] responded to an ICMP ECHO REQUEST in 0.002 seconds,
        ttl: [64]
5  host: [192.168.1.117] responded to an ICMP ECHO REQUEST in 0.0017
        seconds, ttl: [64]
6  host: [192.168.1.176] responded to an ICMP ECHO REQUEST in 0.0014
        seconds, ttl: [254]
7  host: [192.168.1.14] responded to an ICMP ECHO REQUEST in 0.0072
        seconds, ttl: [64]
8  host: [192.168.1.246] responded to an ICMP ECHO REQUEST in 0.049
        seconds, ttl: [64]
9  host: [192.168.1.8] responded to an ICMP ECHO REQUEST in 0.099 seconds,
        ttl: [64]
```

Now that I have done these prototypes I am fairly certain about how I will structure the rest of my scanners, how to interact with Python's socket programming interface and how I can use the struct module to make and dissect packets. My general plan for the scanners will be to start a process that listens

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 ofTCP 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 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.
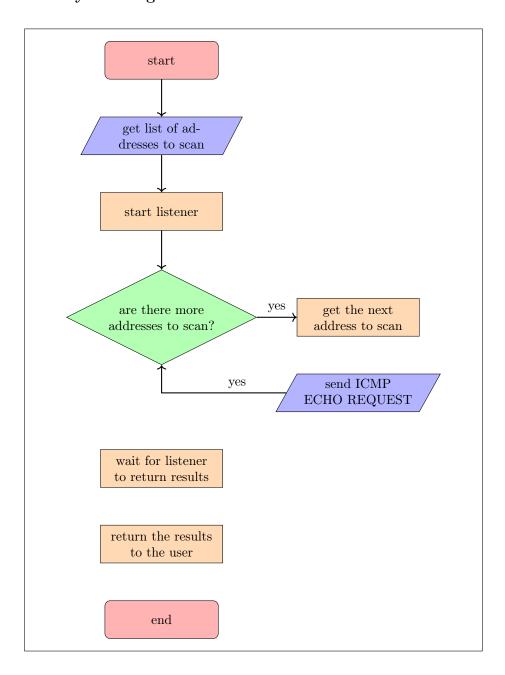
2.3   System Algorithms

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

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:         $mask \leftarrow mask + 2^{maskbit}$
7:     $lower\_bound \leftarrow ip$ AND $mask$ ▷ zero the last 32-$network\_bits$
8:     $upper\_bound \leftarrow ip$ OR ($mask$ XOR 0xFFFFFFFF)   ▷ 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 ofportnumbers such that ranges are specified as start-end instead of start,start+1,...,end

---

1: **procedure** COLLAPSE
2:   $port\_dictionary \leftarrow$ dictionary of lists ofportnumbers
3:   $key\_results \leftarrow$ empty list   ▷ 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** $index \leftarrow 1, (\text{length of } ports) - 1$ **do**
10:        $port = ports[index]$
11:        **if** $index = 0$ **then**
12:          $result \leftarrow result + ports[0]$ ▷ append the first element
13:          **if** $ports[index+1] = port + 1$ **then**
14:            $result \leftarrow result +$ "-"   ▷ begin a new sequence
15:          **else**
16:            $result \leftarrow result +$ ","   ▷ not a sequence
17:        **else if** $port + 1 \neq ports[index+1]$ **then**   ▷ 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:     $result \leftarrow result + ports[(\text{length of } ports)\text{-}1] +$ "}"
24:     append $result$ to $key\_results$
     return "{" + ($key\_results$ separated by ", ") + "}"

---

3 Technical Solution

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

4 Testing

4.1 Test Plan

4.2 Test Table / Testing Evidence (Core: lots of screenshots)

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 8: A prototype program for sending ICMP ECHO REQEST packets

```python
#!/usr/bin/env python
import socket
import struct
import os
import time
from modules.ip_utils import ip_checksum


def main() -> None:
    ICMP_ECHO_REQUEST = 8

    # opens a raw socket for the ICMP protocol
    ping_sock = socket.socket(
        socket.AF_INET,
        socket.SOCK_RAW,
        socket.IPPROTO_ICMP
    )
    # allows manual IP header creation
    # ping_sock.setsockopt(socket.SOL_IP, socket.IP_HDRINCL, 1)

    ID = os.getpid() & 0xFFFF

```

```
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 9: A prototype program for receiving ICMP ECHO REQEST 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,
            set
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 10: 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)
42          # time_waiting stores the time the socket took to become readable
```

44

```python
43          # or returns minus one if it ran out of time

45              if time_waiting == -1:
46                  break
47              time_recieved = time.time()
48              # store the time the packet was recieved
49              recPacket, addr = ping_sock.recvfrom(1024)
50              # recieve the packet
51              ip = headers.ip(recPacket[:20])
52              # unpack the IP header into its respective components
53              icmp = headers.icmp(recPacket[20:28])
54              # unpack the time from the packet.
55              time_sent = struct.unpack(
56                  "d",
57                  recPacket[28:28 + struct.calcsize("d")]
58              )[0]
59              # unpack the value for when the packet was sent
60              time_taken: float = time_recieved - time_sent
61              # calculate the round trip time taken for the packet
62              if icmp.id == ID:
63                  # if the ping was sent from this machine then add it to the
                        list of
64                  # responses
65                  ip_address, port = addr
66                  addresses.add((ip_address, time_taken, ip))
67              elif time_remaining <= 0:
68                  break
69              else:
70                  continue
71          # return a list of all the addesses that replied to our ICMP echo
                request.
72          return addresses


75  def main() -> None:
76      with closing(
77              socket.socket(
78                  socket.AF_INET,
79                  socket.SOCK_RAW,
80                  socket.IPPROTO_ICMP
81              )
82      ) as ping_sock:
83          ip_addresses = ["127.0.0.1"] # ip_utils.ip_range("192.168.43.0",
                24)
84          # generate the range of IP addresses to scan.
85          # get the local ip address
86          addresses = [
87              ip
88              for ip in ip_addresses
89              if (
```

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

6.3   subnet_to_addresses

Listing 11: 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()
```

```python
 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

`Connect()` scanning

Listing 12: prototype TCP Connect() scanner only attempting to detect the state of port 22

```python
 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} succedded")
18      except ConnectionRefusedError:
19          print(f"port {PORT} is closed")
```

Listing 13: A program that performs TCP Connect() scanning

```python
 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
                        target
22                  # address and port
23                  open_ports.append(port)
24                  # if the connection was successful then add the port to
                        the
25                  # list of open ports
26          except ConnectionRefusedError:
27              pass
28      return open_ports
29
30
31  def main() -> None:
32      open_ports = connect_scan("192.168.43.225", set(range(65535)))
33      print("\n".join(map(lambda x: f"port: [{x}]\tis open", open_ports)))
```

SYN scanning

Listing 14: 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 15: 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
                        it
41                  # did append it
42              else:
43                  continue
44          print("finished listening")
45      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,
                dest_ip, 2)
58          # create a TCP packet with the syn flag
59          with closing(
60                  socket.socket(
61                      socket.AF_INET,
62                      socket.SOCK_RAW,
63                      socket.IPPROTO_TCP
64                  )
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 16: 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 17: 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,
              filtered.
47      3 -> closed
48      0|1|2|9|10|13 -> filtered
49      -1 -> error with arguments
50      open|filtered means that they are either open or
51      filtered but return nothing.
52      """
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,
                  time_remaining)
64          # wait for socket to be readable
65          if time_waiting == -1:
66              break
67          else:
68              time_remaining -= time_waiting
69          recPacket, addr = ping_sock.recvfrom(1024)
70          # 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:
```

```python
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,
               open|filtered,
97      filtered, closed they are marked open|filtered if no response is
98      recieved at all.
99      """
100
101     local_ip = ip_utils.get_local_ip()
102     local_port = ip_utils.get_free_port()
103     # get local ip address and port number
104     ports: DefaultDict[str, Set[int]] = defaultdict(set)
105     ports["REMAINING"] = ports_to_scan
106     p = Pool(1)
107     udp_listen = p.apply_async(udp_listener, (dest_ip, 4))
108     # start the UDP listener
109     with closing(
110             socket.socket(
111                 socket.AF_INET,
112                 socket.SOCK_RAW,
113                 socket.IPPROTO_UDP
114             )
115     ) as s:
116         for _ in range(2):
117             # repeat 3 times because UDP scanning comes
118             # with a high chance of packet loss
119             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
                                  following",
134                          "argument resulting in a socket error.",
135                          f"\npacket: [{packet_bytes}]\n",
136                          "address: [{dest_ip, dest_port}])"
137                      )
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 18: 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 young one.", "utf-8"), addr)
```

6.6 version_detection

Listing 19: 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
```

```python
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          # seperate 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+)"),
```

```python
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
                            parse_ports(search.group(1)).items():
116                        directives.Probe.exclude[protocol].update(ports)
117                else:
118                    print(line)
119                    input()
120
121            # new probe directive
122            if line.startswith("Probe"):
123                # parse line into probe protocol, name and probestring
124                search = regexes["probe"].search(line)
125                if search:
126                    try:
127                        proto, name, string = search.groups()
128                    except ValueError:
129                        print(line)
130                        raise
131                    probes[name][proto] = directives.Probe(proto, name,
                            string)
132                    # assign current_probe to the most recently added probe
133                    current_probe = probes[name][proto]
134                else:
135                    print(line)
136                    input()
137
138            # 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
                            parse_ports(search.group(1)).items():
173                        current_probe.ports[protocol].update(ports)
174                else:
175                    print(line)
176                    input()
177            # new totalwaitms directive
178            elif line.startswith("totalwaitms"):
179                search = regexes["totalwaitms"].search(line)
180                if search:
181                    current_probe.totalwaitms = int(search.group(1))
182                else:
183                    print(line)
184                    input()
185
186            # new rarity directive
187            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 20: 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
               things.
47
48       def __init__(
49               self,
50               service: str,
51               pattern: bytes,
52               pattern_options: str,
53               version_info: str
54       ):
55           self.version_info: Dict[str, str] = dict()
56           self.cpes: Dict[str, Dict[str, str]] = dict()
57           self.service: str = service
```

```python
58          # bitwise or is used to combine flags
59          # pattern options will never be anything but a
60          # combination of s and i.
61          # the default value of re.V1 is so that
62          # re uses the newer matching engine.
63          flags = reduce(
64              operator.ior,
65              [
66                  self.options_to_flags[opt]
67                  for opt in pattern_options
68              ],
69              0
70          )
71          try:
72              self.pattern: Pattern = re.compile(
73                  pattern,
74                  flags=flags
75              )
76          except Exception as e:
77              print("Regex failed to compile:")
78              print(e)
79              print(pattern)
80              input()
81
82          vinfo_regex = re.compile(r"([pvihod]|cpe:)([/|])(.+?)\2([a]*)")
83          cpe_regex = re.compile(
84              ":?".join((
85                  "(?P<part>[aho])",
86                  "(?P<vendor>[^:]*)",
87                  "(?P<product>[^:]*)",
88                  "(?P<version>[^:]*)",
89                  "(?P<update>[^:]*)",
90                  "(?P<edition>[^:]*)",
91                  "(?P<language>[^:]*)"
92              ))
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
                            regex
101                     # are turned to strings
102                     self.cpes[Match.cpe_part_map[part]] = {
103                         key: value
104                         for key, value
105                         in search.groupdict().items()
106                     }
```

```python
107                     else:
108                         self.version_info[
109                             Match.letter_to_name[fieldname]
110                         ] = val
111
112         def __repr__(self) -> str:
113             return "Match(" + ", ".join((
114                     f"service={self.service}",
115                     f"pattern={self.pattern}",
116                     f"version_info={self.version_info}",
117                     f"cpes={self.cpes}"
118                 )) + ")"
119
120         def matches(self, string: bytes) -> bool:
121             def replace_groups(
122                     string: str,
123                     original_match: RE_Match
124             ) -> str:
125                 """
126                 This function takes in a string and the original
127                 regex search performed on the data recieved and
128                 replaces all of the $i, $SUBST, $I, $P occurances
129                 with the relavant formatted text that they produce.
130                 """
131                 def remove_unprintable(
132                         group: int,
133                         original_match: RE_Match
134                 ) -> bytes:
135                     """
136                     Mirrors the P function from nmap which
137                     is used to print only printable characters.
138                     i.e. W\OO\OR\OK\OG\OR\OO\OU\OP -> WORKGROUP
139                     """
140                     return b"".join(
141                         i for i in original_match.group(group)
142                         if ord(i) in (
143                             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
```

```python
        ) -> bytes:
            """
            Mirrors the SUBST function from nmap which is used to
            format some information found by the regex.
            by substituting all instances of `before` with `after`.
            """
            return original_match.group(group).replace(before, after)

        def unpack_uint(
                group: int,
                endianness: str,
                original_match: RE_Match
        ) -> bytes:
            """
            Mirrors the I function from nmap which is used to
            unpack an unsigned int from some bytes.
            """
            return bytes(struct.unpack(
                endianness + "I",
                original_match.group(group)
            ))

        text = bytes(string, "utf-8")
        # fill in the version information from the regex match
        # find all the dollar groups:
        dollar_regex = re.compile(r"\$(\d)")
        # find all the $i's in string
        numbers = set(int(i) for i in dollar_regex.findall(string))
        # for each $i found i
        for group in numbers:
            text = text.replace(
                bytes(f"${group}", "utf-8"),
                original_match.group(group)
            )
        # having replaced all of the groups we can now
        # start doing the SUBST, P and I commands.
        subst_regex = re.compile(rb"\$SUBST\((\d),(.+),(.+)\)")
        # iterate over all of the matches found by the SUBST regex
        for match in subst_regex.finditer(text):
            num, before, after = match.groups()
            # replace the full match (group 0)
            # with the output of substitute
            # with the specific arguments
            text.replace(
                match.group(0),
                substitute(int(num), before, after, original_match)
            )

        p_regex = re.compile(rb"\$P\((\d)\)")
        for match in p_regex.finditer(text):
```

```python
                num = match.group(1)
                # replace the full match (group 0)
                # with the output of remove_unprintable
                # with the specific arguments
                text.replace(
                    match.group(0),
                    remove_unprintable(int(num), original_match)
                )

            i_regex = re.compile(br"\$I\((\d),\"(\S)\"\)")
            for match in i_regex.finditer(text):
                num, endianness = match.groups()
                # this means replace group 0 -> the whole match
                # with the output of the unpack_uint
                # with the specified arguments
                text.replace(
                    match.group(0),
                    unpack_uint(
                        int(num.decode()),
                        endianness.decode(),
                        original_match
                    )
                )

            return text.decode()

        search = self.pattern.search(string)
        if search:
            # the fields to replace are all the CPE groups,
            # all of the version info fields.
            self.version_info = {
                key: replace_groups(value, search)
                for key, value in self.version_info.items()
            }
            self.cpes = {
                outer_key: {
                    inner_key: replace_groups(value, search)
                    for inner_key, value in outer_dict.items()
                }
                for outer_key, outer_dict in self.cpes.items()
            }

            return True
        else:
            return False


@dataclass
class Target:
    """
```

```python
      This class holds data about targets to
      scan. the dataclass decorator is simply
      a way of python automatically writing some
      of the basic methods a class for storing data
      has, such as __repr__ for printing information
      in the object etc.
      """
      address: str
      open_ports: DefaultDict[str, Set[int]]
      open_filtered_ports: DefaultDict[str, Set[int]]
      services: Dict[int, Match] = field(default_factory=dict)

      def __repr__(self) -> str:
          def collapse(port_dict: DefaultDict) -> str:
              """
              Collapse a list of port numbers so that
              only the unique ones and the start and end
              of a sequence are displayed.
              1,2,3,4,5,7,9,11,13,14,15,16,17 -> 1-5,7,9,11,13-17
              """
              store_results = list()
              for key in port_dict:
                  # items is a sorted list of a set of ports.
                  items: List[int] = sorted(port_dict[key])
                  key_result = f'"{key}":' + "{"
                  # if its an empty list return now to avoid errors
                  if len(items) != 0:
                      new_sequence = False
                      # enumerate up until the one before
                      # the last to prevent index errors.
                      for index, item in enumerate(items[:-1]):
                          # if its the first one add it on
                          if index == 0:
                              key_result += f"{item}"
                              # if its a sequence start one else put a comma
                              if items[index+1] == item+1:
                                  key_result += "-"
                              else:
                                  key_result += ","
                          # if the sequence breaks then put a comma
                          elif item+1 != items[index+1]:
                              key_result += f"{item},"
                              new_sequence = True
                          # if its a new sequence the put the '-'s in
                          elif item+1 == items[index+1] and new_sequence:
                              key_result += f"{item}-"
                              new_sequence = False
                      # because we only iterate to the one before
                      # the last element, add the last element on to the end.
                      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
            file.
325     It holds information such as the protocol to use, the string to send,
326     the ports to scan, the time to wait for a null TCP to return a
            banner,
327     the rarity of the probe (how often it will return a response) and the
328     probes to try if this one fails.
329     """
330
331     # a default dict is one which takes in a
332     # "default factory" which is called when
333     # a new key is introduced to the dict
334     # in this case the default factory is
335     # the set function meaning that when I
336     # do exclude[protocol].update(ports)
337     # but exclude[protocol] has not yet been defined
338     # it will be defined as an empty set
339     # allowing me to update it with ports.
340     exclude: DefaultDict[str, Set[int]] = defaultdict(set)
341     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
```

```python
355             else:
356                 raise ValueError(
357                     f"Probe object must have protocol TCP or UDP not
                        {protocol}.")
358         self.name: str = probename
359         self.string: str = probe
360         self.payload: bytes = bytes(probe, "utf-8")
361         self.matches: Set[Match] = set()
362         self.softmatches: Set[Match] = set()
363         self.ports: DefaultDict[str, Set[int]] = defaultdict(set)
364         self.totalwaitms: int = 6000
365         self.tcpwrappedms: int = 3000
366         self.rarity: int = -1
367         self.fallback: Set[str] = set()

369     def __repr__(self) -> str:
370         """
371         This is the function that is called when something
372         tries to print an instance of this class.
373         It is used to reveal information internal
374         to the class.
375         """
376         return ", ".join([
377             f"Probe({self.protocol}",
378             f"{self.name}",
379             f"\"{self.string}\"",
380             f"{len(self.matches)} matches",
381             f"{len(self.softmatches)} softmatches",
382             f"ports: {self.ports}",
383             f"rarity: {self.rarity}",
384             f"fallbacks: {self.fallback})"
385         ])

387     def scan(self, target: Target) -> Target:
388         """
389         scan takes in an object of class Target to
390         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.

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              "U": "UDP",
496              "T": "TCP"
497          }
498          if pairs:
499              def pair_to_ports(pair: Tuple[str, str]) -> Set[int]:
500                  """
501                  a function to go from a port pair i.e. (80-85)
502                  to the set of specified ports: {80,81,82,83,84,85}
503                  """
```

```
504              start, end = pair
505              return set(range(
506                  int(start),
507                  int(end)+1
508              ))
509          # ports contains the set of all ANY/TCP/UDP specified ports
510          ports[proto_map[protocol]] = set(reduce(
511              operator.ior,
512              map(pair_to_ports, pairs)
513          ))
514
515      singles = single_regex.findall(portstring)
516      # for each of the ports that are specified on their own
517      # cast them to int and update the set of all ports with
518      # that list.
519      ports[proto_map[protocol]].update(map(int, singles))
520
521  return ports
522
523
524 def parse_probes(probe_file: str) -> PROBE_CONTAINER:
525     """
526     Extracts all of the probe directives from the
527     file pointed to by probe_file.
528     """
529     # lines contains each line of the file which doesn't
530     # start with a # and is not empty.
531     lines = [
532         line
533         for line in open(probe_file).read().splitlines()
534         if line and not line.startswith("#")
535     ]
536
537     # list holding each of the probe directives.
538     probes: PROBE_CONTAINER = defaultdict(dict)
539
540     regexes: Dict[str, Pattern] = {
541         "probe":       re.compile(r"Probe (TCP|UDP) (\S+) q\|(.*)\|"),
542         "match":       re.compile(" ".join([
543             r"(?P<type>softmatch|match)",
544             r"(?P<service>\S+)",
545             r"m([@/%=|])(?P<regex>.+?)\3(?P<flags>[si]*)"
546         ])),
547         "rarity":      re.compile(r"rarity (\d+)"),
548         "totalwaitms": re.compile(r"totalwaitms (\d+)"),
549         "tcpwrappedms": re.compile(r"tcpwrappedms (\d+)"),
550         "fallback":    re.compile(r"fallback (\S+)"),
551         "ports":       re.compile(r"ports (\S+)"),
552         "exclude":     re.compile(r"Exclude T:(\S+)")
553     }
```

```
554
555     # parse the probes out from the file
556     for line in lines:
557         # add any ports to be excluded to the base probe class
558         if line.startswith("Exclude"):
559             search = regexes["exclude"].search(line)
560             if search:
561                 # parse the ports from the grouped output of
562                 # a search with the regex defined above.
563                 for protocol, ports in
                        parse_ports(search.group(1)).items():
564                     Probe.exclude[protocol].update(ports)
565             else:
566                 print(line)
567                 input()
568
569         # new probe directive
570         if line.startswith("Probe"):
571             # parse line into probe protocol, name and probestring
572             search = regexes["probe"].search(line)
573             if search:
574                 try:
575                     proto, name, string = search.groups()
576                 except ValueError:
577                     print(line)
578                     raise
579                 probes[name][proto] = Probe(proto, name, string)
580                 # assign current_probe to the most recently added probe
581                 current_probe = probes[name][proto]
582             else:
583                 print(line)
584                 input()
585
586         # new match directive
587         elif line.startswith("match") or line.startswith("softmatch"):
588             search = regexes["match"].search(line)
589             if search:
590                 # the remainder of the string after the match
591                 version_info = line[search.end()+1:]
592                 # escape the curly braces so the regex engine doesn't
593                 # consider them to be special characters
594                 pattern = bytes(search.group("regex"), "utf-8")
595                 # these replace the literal \n, \r and \t
596                 # strings with their actual characters
597                 # i.e. \n -> newline character
598                 pattern = pattern.replace(b"\\n", b"\n")
599                 pattern = pattern.replace(b"\\r", b"\r")
600                 pattern = pattern.replace(b"\\t", b"\t")
601                 matcher = Match(
602                     search.group("service"),
```

73

```python
                    pattern,
                    search.group("flags"),
                    version_info
                )
                if search.group("type") == "match":
                    current_probe.matches.add(matcher)
                else:
                    current_probe.softmatches.add(matcher)

            else:
                print(line)
                input()

        # new ports directive
        elif line.startswith("ports"):
            search = regexes["ports"].search(line)
            if search:
                for protocol, ports in
                        parse_ports(search.group(1)).items():
                    current_probe.ports[protocol].update(ports)
            else:
                print(line)
                input()
        # new totalwaitms directive
        elif line.startswith("totalwaitms"):
            search = regexes["totalwaitms"].search(line)
            if search:
                current_probe.totalwaitms = int(search.group(1))
            else:
                print(line)
                input()

        # new rarity directive
        elif line.startswith("rarity"):
            search = regexes["rarity"].search(line)
            if search:
                current_probe.rarity = int(search.group(1))
            else:
                print(line)
                input()

        # new fallback directive
        elif line.startswith("fallback"):
            search = regexes["fallback"].search(line)
            if search:
                current_probe.fallback = set(search.group(1).split(","))
            else:
                print(line)
                input()
    return probes
```

Listing 21: A python module I made to dissect and hold protocol headers.

```python
import struct
import socket
from typing import Dict


class ip:
    """
    A class for parsing, storing and displaying
    data from an IP header.
    """
    def __init__(self, header: bytes):
        # first unpack the IP header
        (
            ip_hp_ip_v,
            ip_dscp_ip_ecn,
            ip_len,
            ip_id,
            ip_flgs_ip_off,
            ip_ttl,
            ip_p,
            ip_sum,
            ip_src,
            ip_dst
        ) = struct.unpack('!BBHHHBBHII', header)
        # now deal with the sub-byte sized components
        hl_v = f"{ip_hp_ip_v:08b}"
        ip_v = int(hl_v[:4], 2)
        ip_hl = int(hl_v[4:], 2)
        # splits hl_v in ip_v and ip_hl which store the IP version
                number and
        # header length respectively
        dscp_ecn = f"{ip_dscp_ip_ecn:08b}"
        ip_dscp = int(dscp_ecn[:6], 2)
        ip_ecn = int(dscp_ecn[6:], 2)
        # splits dscp_ecn into ip_dscp and ip_ecn
        # which are two of the compenents
        # in an IP header
        flgs_off = f"{ip_flgs_ip_off:016b}"
        ip_flgs = int(flgs_off[:3], 2)
        ip_off = int(flgs_off[3:], 2)
        # splits flgs_off into ip_flgs and ip_off which represent the ip
                header
        # flags and the data offset
        src_addr = socket.inet_ntoa(struct.pack('!I', ip_src))
        dst_addr = socket.inet_ntoa(struct.pack('!I', ip_dst))
        self.version: int = ip_v
```

```python
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_flgs
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 advertisment"
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:
                      ({self.type/self.code})"
171
172          self.id: int
173          self.sequence: int
174          if self.type in {0, 8}:
175              self.id = socket.htons(remainder >> 16)
176              self.sequence = socket.htons(remainder & 0xFFFF)
177          else:
178              self.id = -1
179              self.sequence = -1
180
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:
```

```python
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("!HHIIBBHHH", 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 22: A python module I wrote to contain lots of useful functions which I found I was declaring in multiple places and makign 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
```

```python
        # cast them to a string and join them with '.'s
        return ".".join(
            str(
                (long >> (8*(3-i))) & 0xFF
            )
            for i in range(4)
        )


def dot_to_long(ip: str) -> int:
    """
    Take an ip address in dot notation and return the packed 32 bit int
        version
    i.e. dot_to_long("127.0.0.1") = 0x7F000001
    """

    # dot form ips: a.b.c.d must have each
    # part (a,b,c,d) between 0 and 255,
    # otherwise they are invalid

    parts = [int(i) for i in ip.split(".")]

    if not all(
            0 <= i <= 255
            for i in parts
    ):
        raise ValueError(f"Invalid dot form IP address: [{ip}]")

    else:
        # for each part of the dotted IP address
        # bit shift left each part by eight times
        # three minus it's position. This puts the bits
        # from each part in the right place in the final sum
        # a.b.c.d -> a<<3*8 + b<<2*8 + c<<1*8 + d<<0*8
        return sum(
            part << ((3-i)*8)
            for i, part in enumerate(parts)
        )


@singledispatch
def is_valid_ip(ip: Union[str, int]) -> bool:
    """
    checks whether a given IP address is valid.
    """


@is_valid_ip.register
def _(ip: int):
    # this is the int overload variant of
```

```python
 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
                65536.
114        """
115        # port numbers must be between 0 and 65535(2^16 - 1)
116        if 0 <= port_num < 2**16:
117            return True
118        else:
119            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
                error.
128        If the IP address is invalid according to is_valid_ip it raises an
                error.
129        """
130
```

```python
131        if not 0 <= network_bits <= 32:
132            raise ValueError(f"Invalid number of network bits:
                   [{network_bits}]")
133
134        if not is_valid_ip(ip):
135            raise ValueError(f"Invalid IP address: [{ip}]")
136        # get the ip as long form which is useful
137        # later on for using bitwise operators
138        # to isolate only the constant(network) bits
139        ip_long = dot_to_long(ip)
140
141        # generate the bit mask which specifies
142        # which bits to keep and which to discard
143        mask = int(
144            f"{'1'*network_bits:0<32s}",
145            base=2
146        )
147        lower_bound = ip_long & mask
148        upper_bound = ip_long | (mask ^ 0xFFFFFFFF)
149
150        # turn all the long form IP addresses between
151        # the lower and upper bound into dot form
152        return set(
153            long_to_dot(long_ip)
154            for long_ip in
155            range(lower_bound, upper_bound + 1)
156        )
157
158
159    def get_local_ip() -> str:
160        """
161        Connects to the google.com with UDP and gets
162        the IP address used to connect(the local address).
163        """
164        with closing(
165                socket.socket(
166                    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
                socket,
178        the socket is then closed and the port number assigned is returned.
```

```python
179        """
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
```

```python
        ICMP_ECHO_REQUEST = 8
        # pack the information for the dummy header needed
        # for the IP checksum
        dummy_header = struct.pack(
            "bbHHh",
            ICMP_ECHO_REQUEST,
            0,
            0,
            ID,
            1
        )
        # pack the current time into a double
        time_bytes = struct.pack("d", time.time())
        # define the bytes to repeat in the data section of the packet
        # this makes the packets easily identifiable in packet captures.
        bytes_to_repeat_in_data = map(ord, " y33t ")
        # calculate the number of bytes left for data
        data_bytes = (192 - struct.calcsize("d"))
        # first pack the current time into the start of the data section
        # the pack the identifiable data into the rest
        data = (
            time_bytes +
            bytes(islice(cycle(bytes_to_repeat_in_data), data_bytes))
        )
        # get the IP checksum for the dummy header and data
        # and switch the bytes into the order expected by the network
        checksum = socket.htons(ip_checksum(dummy_header + data))
        # pack the header with the correct checksum and information
        header = struct.pack(
            "bbHHh",
            ICMP_ECHO_REQUEST,
            0,
            checksum,
            ID,
            1
        )
        # concatonate the header bytes and the data bytes
        return header + data


def make_tcp_packet(
        src: int,
        dst: int,
        from_address: str,
        to_address: str,
        flags: int) -> bytes:
    """
    Takes in the source and destination port/ip address
    returns a tcp packet.
    flags:
```

```python
        2 => SYN
        18 => SYN:ACK
        4 => RST
        """
        # validate that the information passed in is valid
        if flags not in {2, 18, 4}:
            raise ValueError(
                f"Flags must be one of 2:SYN, 18:SYN,ACK, 4:RST. not:
                    [{flags}]"
            )
        if not is_valid_ip(from_address):
            raise ValueError(
                f"Invalid source IP address: [{from_address}]"
            )
        if not is_valid_ip(to_address):
            raise ValueError(
                f"Invalid destination IP address: [{to_address}]"
            )
        if not is_valid_port_number(src):
            raise ValueError(
                f"Invalid source port: [{src}]"
            )
        if not is_valid_port_number(dst):
            raise ValueError(
                f"Invalid destination port: [{dst}]"
            )
        # turn the ip addresses into long form
        src_addr = dot_to_long(from_address)
        dst_addr = dot_to_long(to_address)

        seq = ack = urg = 0
        data_offset = 6 << 4
        window_size = 1024
        max_segment_size = (2, 4, 1460)
        # pack the dummy header needed for the checksum calculation
        dummy_header = struct.pack(
            "!HHIIBBHHHBBH",
            src,
            dst,
            seq,
            ack,
            data_offset,
            flags,
            window_size,
            0,
            urg,
            *max_segment_size
        )
        # pack the psuedo header that is also needed for the checksum
        # just because TCP and why not
```

```python
    psuedo_header = struct.pack(
        "!IIBBH",
        src_addr,
        dst_addr,
        0,
        6,
        len(dummy_header)
    )

    checksum = ip_checksum(psuedo_header + dummy_header)
    # pack the final TCP packet with the relevant data and checksum
    return struct.pack(
        "!HHIIBBHHHBBH",
        src,
        dst,
        seq,
        ack,
        data_offset,
        flags,
        window_size,
        checksum,
        urg,
        *max_segment_size
    )


def make_udp_packet(
        src: int,
        dst: int,
        from_address: str,
        to_address: str
) -> bytes:
    """
    Takes in: source IP address and port, destination IP address and
        port.
    Returns: a UDP packet with those properties.
    the IP addresses are needed for calculating the checksum.
    """
    # validate data passed in
    if not is_valid_ip(from_address):
        raise ValueError(
            f"Invalid source IP address: [{from_address}]"
        )
    if not is_valid_ip(to_address):
        raise ValueError(
            f"Invalid destination IP address: [{to_address}]"
        )
    if not is_valid_port_number(src):
        raise ValueError(
            f"Invalid source port: [{src}]"
```

```
377              )
378          if not is_valid_port_number(dst):
379              raise ValueError(
380                  f"Invalid destination port: [{dst}]"
381              )
382
383          UDP_length = 8
384          # pack the dummy and psuedo headers needed for the checksum
385          dummy_header = struct.pack(
386              "!HHHH",
387              src,
388              dst,
389              UDP_length,
390              0
391          )
392          # 17 is the UDP protocol number
393          psuedo_header = struct.pack(
394              "!IIBBH",
395              src,
396              dst,
397              0,
398              17,
399              len(dummy_header)
400          )
401
402          checksum = ip_checksum(psuedo_header + dummy_header)
403          # pack the data and checksum into the right format
404          # and return the packed bytes
405          return struct.pack(
406              "!HHHH",
407              src,
408              dst,
409              UDP_length,
410              checksum
411          )
412
413
414  def wait_for_socket(sock: socket.socket, wait_time: float) -> float:
415      """
416      Wait for wait_time seconds or until the socket is readable.
417      If the socket is readable return a tuple of the socket and the time
              taken
418      otherwise return None.
419      """
420
421      start = time.time()
422      is_socket_readable = select.select([sock], [], [], wait_time)
423      taken = time.time() - start
424      if is_socket_readable[0] == []:
425          return float(-1)
```

```
426        else:
427            return taken
```

Listing 23: A python module I made to hold all of the listeners I had made for each of the different scanning types.

```python
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)
        # time_waiting stores the time the socket took to become readable
34      # or returns minus one if it ran out of time
35
36          if time_waiting == -1:
37              break
38          time_recieved = time.time()
39          # store the time the packet was recieved
40          recPacket, addr = ping_sock.recvfrom(1024)
41          # recieve the packet
42          ip = headers.ip(recPacket[:20])
43          # unpack the IP header into its respective components
```

```
44          icmp = headers.icmp(recPacket[20:28])
45          # unpack the time from the packet.
46          time_sent = struct.unpack(
47              "d",
48              recPacket[28:28 + struct.calcsize("d")]
49          )[0]
50          # unpack the value for when the packet was sent
51          time_taken: float = time_recieved - time_sent
52          # calculate the round trip time taken for the packet
53          if icmp.id == ID:
54              # if the ping was sent from this machine then add it to the
                    list of
55              # responses
56              ip_address, port = addr
57              # this is to prevent a bug where IPs were being added twice
58              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 addesses that replied to our ICMP echo
            request.
66      return addresses


def udp(dest_ip: str, timeout: float) -> Set[int]:
    """
    This listener detects UDP packets from dest_ip in the given timespan,
    all ports that send direct replies are marked as being open.
    Returns a list of open ports.
    """

    time_remaining = timeout
    ports: Set[int] = set()
    with socket.socket(
            socket.AF_INET,
            socket.SOCK_RAW,
            socket.IPPROTO_UDP
    ) as s:
        while True:
            time_taken = ip_utils.wait_for_socket(s, time_remaining)
            if time_taken == -1:
                break
            else:
                time_remaining -= time_taken
            packet = s.recv(1024)
            ip = headers.ip(packet[:20])
            udp = headers.udp(packet[20:28])
```

```python
                if dest_ip == ip.source and ip.protocol == 17:
                    ports.add(udp.src)

        return ports


    def icmp_unreachable(src_ip: str, timeout: float = 2) -> int:
        """
        This listener detects ICMP destination unreachable
        packets and returns the icmp code.
        This is later used to mark them as either close, open|filtered,
            filtered.
        3 -> closed
        0|1|2|9|10|13 -> filtered
        -1 -> error with arguments
        open|filtered means that they are either open or
        filtered but return nothing.
        """

        ping_sock = socket.socket(
            socket.AF_INET,
            socket.SOCK_RAW,
            socket.IPPROTO_ICMP
        )
        # open raw socket to listen for ICMP destination unrechable packets
        time_remaining = timeout
        code = -1
        while True:
            time_waiting = ip_utils.wait_for_socket(ping_sock,
                time_remaining)
            # wait for socket to be readable
            if time_waiting == -1:
                break
            else:
                time_remaining -= time_waiting
            recPacket, addr = ping_sock.recvfrom(1024)
            # recieve the packet
            ip = headers.ip(recPacket[:20])
            icmp = headers.icmp(recPacket[20:28])
            valid_codes = [0, 1, 2, 3, 9, 10, 13]
            if (
                    ip.source == src_ip
                    and icmp.type == 3
                    and icmp.code in valid_codes
            ):
                code = icmp.code
                break
            elif time_remaining <= 0:
                break
            else:
```

```
140              continue
141      ping_sock.close()
142      return code
143
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 24: 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
```

```python
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,
                      exiting")
52                  exit()
53          p.close()
54          p.join()
55          # close and join the process pool to so that all the values
56          # have been returned and the pool closed
57          return replied.get()
58
```

```python
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 specifed 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:
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      # start the TCP ACK listener in the background
100     for port in portlist:
101         # flag = 2 for syn scan
102         packet = ip_utils.make_tcp_packet(
103             src_port,
104             port,
105             local_ip,
106             dest_ip,
```

94

```
107                2
108            )
109        with closing(
110                socket.socket(
111                    socket.AF_INET,
112                    socket.SOCK_RAW,
113                    socket.IPPROTO_TCP
114                )
115        ) as s:
116            s.sendto(packet, (dest_ip, port))
117            # send the packet to its destination
118    p.close()
119    p.join()
120    ports = listener.get()
121    ports["FILTERED"] = portlist - ports["OPEN"] - ports["CLOSED"]
122    return ports


def udp(
        dest_ip: str,
        ports_to_scan: Set[int]
) -> listeners.PORTS:
    """
    Takes in a destination IP address in either dot or long form and
    a list of ports to scan. Sends UDP packets to each port specified
    in portlist and uses the listeners to mark them as open,
            open|filtered,
    filtered, closed they are marked open|filtered if no response is
    recieved at all.
    """

    local_ip = ip_utils.get_local_ip()
    local_port = ip_utils.get_free_port()
    # get local ip address and port number
    ports: listeners.PORTS = defaultdict(set)
    ports["REMAINING"] = ports_to_scan
    p = Pool(1)
    udp_listen = p.apply_async(listeners.udp, (dest_ip, 4))
    # start the UDP listener
    with closing(
            socket.socket(
                socket.AF_INET,
                socket.SOCK_RAW,
                socket.IPPROTO_UDP
            )
    ) as s:
        for _ in range(2):
            # repeat 3 times because UDP scanning comes
            # with a high chance of packet loss
            for dest_port in ports["REMAINING"]:
```

```python
                try:
                    packet = ip_utils.make_udp_packet(
                        local_port,
                        dest_port,
                        local_ip,
                        dest_ip
                    )
                    # create the UDP packet to send
                    s.sendto(packet, (dest_ip, dest_port))
                    # send the packet to the currently scanning address
                except socket.error:
                    packet_bytes = " ".join(map(hex, packet))
                    print(
                        "The socket modules sendto method with the
                            following",
                        "argument resulting in a socket error.",
                        f"\npacket: [{packet_bytes}]\n",
                        "address: [{dest_ip, dest_port}])"
                    )

    p.close()
    p.join()

    ports["OPEN"].update(udp_listen.get())

    ports["REMAINING"] -= ports["OPEN"]
    # only scan the ports which we know are not open
    with closing(
            socket.socket(
                socket.AF_INET,
                socket.SOCK_RAW,
                socket.IPPROTO_UDP
            )
    ) as s:
        for dest_port in ports["REMAINING"]:
            try:
                packet = ip_utils.make_udp_packet(
                    local_port,
                    dest_port,
                    local_ip,
                    dest_ip
                )
                # make a new UDP packet
                p = Pool(1)
                icmp_listen = p.apply_async(
                    listeners.icmp_unreachable,
                    dest_ip,
                )
                # start the ICMP listener
                time.sleep(1)
```

```
205                s.sendto(packet, (dest_ip, dest_port))
206                # send packet
207                p.close()
208                p.join()
209                icmp_code = icmp_listen.get()
210                # recieve ICMP code from the ICMP listener
211                if icmp_code in {0, 1, 2, 9, 10, 13}:
212                    ports["FILTERED"].add(dest_port)
213                elif icmp_code == 3:
214                    ports["CLOSED"].add(dest_port)
215            except socket.error:
216                packet_bytes = " ".join(map("{:02x}".format, packet))
217                ip_utils.eprint(
218                    "The socket modules sendto method with the following",
219                    "argument resulting in a socket error.",
220                    f"\npacket: [{packet_bytes}]\n",
221                    "address: [{dest_ip, dest_port}])"
222                )
223        # this creates a new set which contains all the elements that
224        # are in the list of ports to be scanned but have not yet
225        # been classified
226        ports["OPEN|FILTERED"] = (
227            ports["REMAINING"]
228            - ports["OPEN"]
229            - ports["FILTERED"]
230            - ports["CLOSED"]
231        )
232        # set comprehension to update the list of open filtered ports
233        return ports
234
235
236  def version_detect_scan(
237          target: directives.Target,
238          probes: directives.PROBE_CONTAINER
239  ) -> directives.Target:
240      for probe_dict in probes.values():
241          for proto in probe_dict:
242              target = probe_dict[proto].scan(target)
243      return target
```

6.8 examples

Listing 25: 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
                    examples"
36          )
37          print("\n".join(examples))
```

6.9   netscan

Listing 26: 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,
```

```python
        directives,
    )
from typing import (
    DefaultDict,
    Dict,
)

top_ports = directives.parse_ports(open("top_ports").read())
services: DefaultDict[str, Dict[int, str]] = defaultdict(dict)
for match in re.finditer(
        r"(\S+)\s+(\d+)/(\S+)",
        open("version_detection/nmap-services").read()
):
    service, portnum, protocol = match.groups()
    services[protocol.upper()][int(portnum)] = service

parser = ArgumentParser()
parser.add_argument(
    "target_spec",
    help="specify what to scan, i.e. 192.168.1.0/24"
)
parser.add_argument(
    "-Pn",
    help="assume hosts are up",
    action="store_true"
)
parser.add_argument(
    "-sL",
    help="list targets",
    action="store_true"
)
parser.add_argument(
    "-sn",
    help="disable port scanning",
    action="store_true"
)
parser.add_argument(
    "-sS",
    help="TCP SYN scan",
    action="store_true"
)
parser.add_argument(
    "-sT",
    help="TCP connect scan",
    action="store_true"
)
parser.add_argument(
    "-sU",
    help="UDP scan",
    action="store_true"
```

```python
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 =
        re.compile(r"(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/(\d{1,2})")
83  search = CIDR_regex.search(args.target_spec)
84  if search:
85      base_addr, network_bits = search.groups()
86      addresses = ip_utils.ip_range(
87          base_addr,
88          int(network_bits)
89      )
90  else:
91      base_addr = args.target_spec
92      addresses = {base_addr}
93
94
95  def error_exit(error_type: str) -> bool:
96      messages = {
97          "permission": "\n".join((
98              "You have insufficient permissions to run this type of scan",
99              "EXITING!"
100         ))
101     }
102     try:
103         print(messages[error_type])
104     except KeyError:
105         print(f"ERROR MESSAGE NOT FOUND: {error_type}")
106     exit(-1)
107
```

```
108
109   if args.sL:
110       print("Targets:")
111       print("\n".join(sorted(addresses, key=ip_utils.dot_to_long)))
112   else:
113       if args.sn:
114           def sig_figs(x: float, n: int) -> float:
115               """
116               rounds x to n significant figures.
117               sig_figs(1234, 2) = 1200.0
118               """
119               return round(x, n - (1 + int(floor(log10(abs(x))))))
120
121           try:
122               print("\n".join(
123                   f"host: [{host}]\t" +
124                   "responded to an ICMP ECHO REQUEST in " +
125                   f"{str(sig_figs(taken, 2))+'s':<10s} " +
126                   f"ttl: [{ip_head.time_to_live}]"
127                   for host, taken, ip_head in scanners.ping(addresses)
128               ))
129           except PermissionError:
130               error_exit("permission")
131
132       else:
133           if args.Pn:
134               targets = [
135                   directives.Target(
136                       addr,
137                       defaultdict(set),
138                       defaultdict(set)
139                   )
140                   for addr in addresses
141               ]
142           else:
143               try:
144                   targets = [
145                       directives.Target(
146                           addr,
147                           defaultdict(set),
148                           defaultdict(set),
149                       )
150                       for addr, _, _ in scanners.ping(addresses)
151                   ]
152               except PermissionError:
153                   error_exit("permission")
154           # define the ports to scan
155           if args.ports == "-":
156               # case they have specified all ports
157               ports = {
```

```
158                "UDP": set(range(1, 65536)),
159                "TCP": set(range(1, 65536)),
160            }
161        elif isinstance(args.ports, str):
162            # case they have specifed ports
163            ports = directives.parse_ports(args.ports)
164        else:
165            # default
166            ports = args.ports
167
168        # exclude all the ports speified to be excluded
169        to_exclude = directives.parse_ports(args.exclude_ports)
170        ports["TCP"] -= to_exclude["TCP"]
171        ports["TCP"] -= to_exclude["ANY"]
172        ports["UDP"] -= to_exclude["UDP"]
173        ports["UDP"] -= to_exclude["ANY"]
174
175        # if version scanning is desired
176        if args.sV:
177            probes = directives.parse_probes(
178                "./version_detection/nmap-service-probes"
179            )
180
181        for target in targets:
182            if not args.sU or args.sS:
183                try:
184                    tcp_ports = scanners.tcp(
185                        target.address,
186                        ports["TCP"]
187                    )
188                except PermissionError:
189                    error_exit("permission")
190                target.open_ports["TCP"].update(tcp_ports["OPEN"])
191                target.open_filtered_ports["TCP"].update(tcp_ports["FILTERED"])
192            if args.sT:
193                target.open_ports["TCP"].update(
194                    scanners.connect(target.address, ports["TCP"])
195                )
196            if args.sU:
197                try:
198                    udp_ports = scanners.udp(
199                        target.address,
200                        ports["UDP"]
201                    )
202                except PermissionError:
203                    error_exit("permission")
204
205                target.open_ports["UDP"].update(
206                    udp_ports["OPEN"]
207                )
```

```
208                    target.open_filtered_ports["UDP"].update(
209                        udp_ports["FILTERED"]
210                    )
211                    target.open_filtered_ports["UDP"].update(
212                        udp_ports["OPEN|FILTERED"]
213                    )
214            if args.sV:
215                target = scanners.version_detect_scan(target, probes)
216            # display scan info
217            print()
218            print(f"Scan report for: {target.address}")
219            # print(target)
220            print("Open ports:")
221            for proto, open_ports in target.open_ports.items():
222                for port in open_ports:
223                    try:
224                        service_name = services[proto][port]
225                    except KeyError:
226                        service_name = "unknown"
227                    if port in target.services:
228                        exact_match = target.services[port]
229                        print(
230                            f"{port}/{proto}{exact_match.service:>8s}"
231                        )
232                        # print version information
233                        for key, val in exact_match.version_info.items():
234                            print(f"{key}: {val}")
235                        if exact_match.cpes:
236                            print()
237                            print("CPE:")
238                            for cpe_type, cpe_vals in
                                 exact_match.cpes.items():
239                                print(cpe_type)
240                                try:
241                                    del(cpe_vals["part"])
242                                except KeyError:
243                                    pass
244                                for key, val in cpe_vals.items():
245                                    print(f"{key}: {val}")
246                    print()
```

6.10 tests

Listing 27: A few unit tests I wrote to cover some basic functions that I use as underpinnings for a lot of my other work so I wanted to make sure I hadn't accidentally broken any of them.

```
1  from ip_utils import (
2      dot_to_long,
```

```python
        long_to_dot,
        ip_range,
        is_valid_ip,
        is_valid_port_number
    )


def test_dot_to_long() -> None:
    assert(dot_to_long("127.0.0.1") == 0x7F000001)


def test_long_to_dot() -> None:
    assert(long_to_dot(0x7F000001) == "127.0.0.1")


def test_is_valid_ip() -> None:
    assert(
        is_valid_ip(0x7F000001)
        and is_valid_ip("127.0.0.1")
        and not is_valid_ip("0.0.0.0.0")
        and not is_valid_ip(0xFF_FF_FF_FF_FF)
    )


def test_is_valid_port_number() -> None:
    assert(
        is_valid_port_number(0)
        and is_valid_port_number(65535)
        and not is_valid_port_number(-1)
        and not is_valid_port_number(2**16)
    )


def test_ip_range() -> None:
    assert(
        ip_range("192.168.1.0", 28) == {
            "192.168.1.0",
            "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",
```

```
53            "192.168.1.14",
54            "192.168.1.15"
55        }
56    )
```

Glossary

API Applications Programming Interface 3, 26

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... 23

black box Looking at something from an outsider's perspective knowing nothing about how it works internally. 2, 17

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. 18, 36

CIDR Classless Inter-Domain Routing 17, 23, 40

CPE Common Platform Enumeration 36

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

dbus-daemon A daemon which enable a common interface for inter-process communication. 2

DHCP Dynamic Host Configuration Protocol 2, 3

DHCPCD Dynamic Host Configuration Protocol Client Daemon 2

DNS Domain Name System 21

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

FTP File Transfer Protocol 3, 18

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, 6

HTTP Hypertext transfer Protocol 3, 5, 15

HTTPS Hypertext transfer Protocol Secure 15

ICMP Internet Control Message Protocol 16, 17, 25, 26, 27, 30, 31, 32, 39

IDS Intrusion Detection System 18

IP Internet Protocol 32

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, 5, 15, 40

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

NIC Network Interface Card 2, 4

OSI model Open Systems Interconnection model 3, 26

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. 2, 3, 4, 5, 7, 10, 11, 15, 16, 18, 36, 37

PCAP Packet CAPture 35

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, 17, 18, 37, 40, 41

port knocking Port knocking is where packets must be sent to a sequence of ports before access to the desired port is granted. 18

SCTP Stream Control Transmission Protocol 18

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. 2, 3, 23

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, 11, 18, 23, 36, 37

subnet A subnet is simply the sub-network of every possible IP address that will be used for communication on a particular network. 2, 3, 40

systemd A daemon for controlling what is run when the system starts. 2

TCP Transmission Control Protocol 5, 11, 14, 15, 16, 17, 18, 25, 32, 37