A Level Computer Science Non-Examined Assessment (NEA)

Sam Leonard

# Analysis

## Identification and Background to the Problem

The problem my project tries to solve is how to look at devices on a network from a “bbox” perspective and gain information about what services are running. Services are programs whose 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 a number of steps a device has to go through from when it is turned on until it can connect to the internet. There are many more steps than those listed below, but the most important ones are.

1. Loading networking drivers
2. Starting dhcp daemon
3. Broadcasting dhcp request for an ipaddr
4. Obtaining assigned an ipaddr

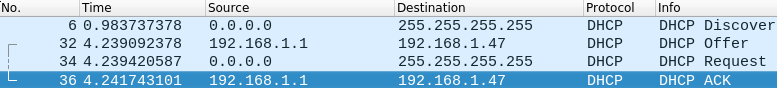
Starting from a Linux computer being switched on, the first step is that the kernel needs to load the networking drivers. The kernel is the basis for the operating system, it is what interacts with the hardware in the most fundamental way. Drivers are small bits of code which the kernel can load in order to interact with certain hardware modules. These can range from graphics drivers for games to use graphics cards, to networking drivers which interact with the nic.

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: upow (power management), sysd (manages the creation of all processes), dbus (manages inter-process communication), iwd (manages my WiFi connections) and finally dhcpcd, which manages all interactions with the network around dhcp.

Once the daemons are all started, the dhcp client can 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 ipaddr and negotiates with new computers trying to join a network to get them a free ipaddr. 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 pkt 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 ipaddr for that client and then responds with a dhcp offer that 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 (see Figure [[dhcpdiagram]](#dhcpdiagram)).

Figure [[dhcp\_negotiate]](#dhcp_negotiate) shows a pkt capture from my laptop where I turned WiFi off, started Wireshark listening and plugged in an Ethernet cable. Wireshark is a program which intercepts all the network communications on a single computer and records them to a file. It also displays them to the user, and is capable of performing an analysis and dissection of each 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. For a definition of pkts see Page .

For the sake of clarity, the Figure displays Wireshark showing only the dhcp pkts, so that the dhcp negotiation can be clearly seen, including the  
255.255.255.255 limited broadcast destination address and the 0.0.0.0 unassigned address in the source column.



*dhcp address negotiation.*

[dhcp\_negotiate]

All computer networking is encapsulated in the osi which has 7 layers:

Application: apis, etc…

Presentation: encryption/decryption, encoding/decoding, decompression etc…

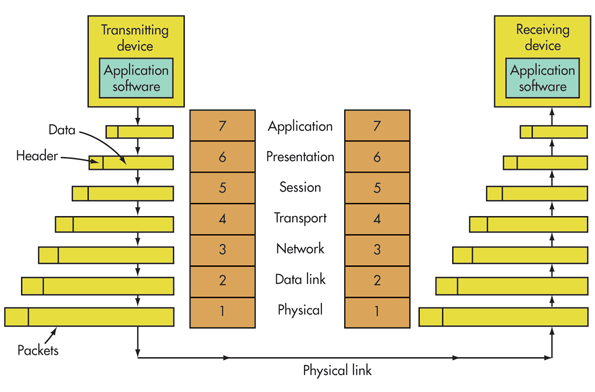
Session: Managing sessions, php session IDs etc…

Transport: TCP and UDP among others.

Network: ICMP and IP among others.

Data Link: MAC addressing, Ethernet protocol etc…

Physical: The physical Ethernet cabling/nic.



*osi diagram, source: https://www.electronicdesign.com.*

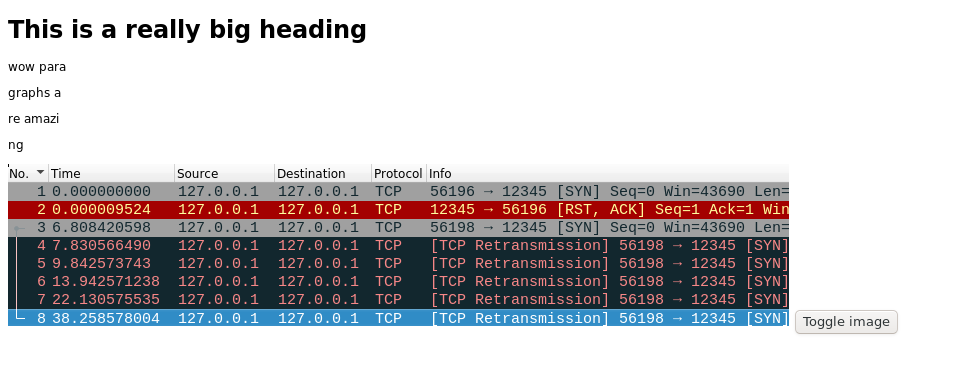
[osi\_model]

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 pkt.

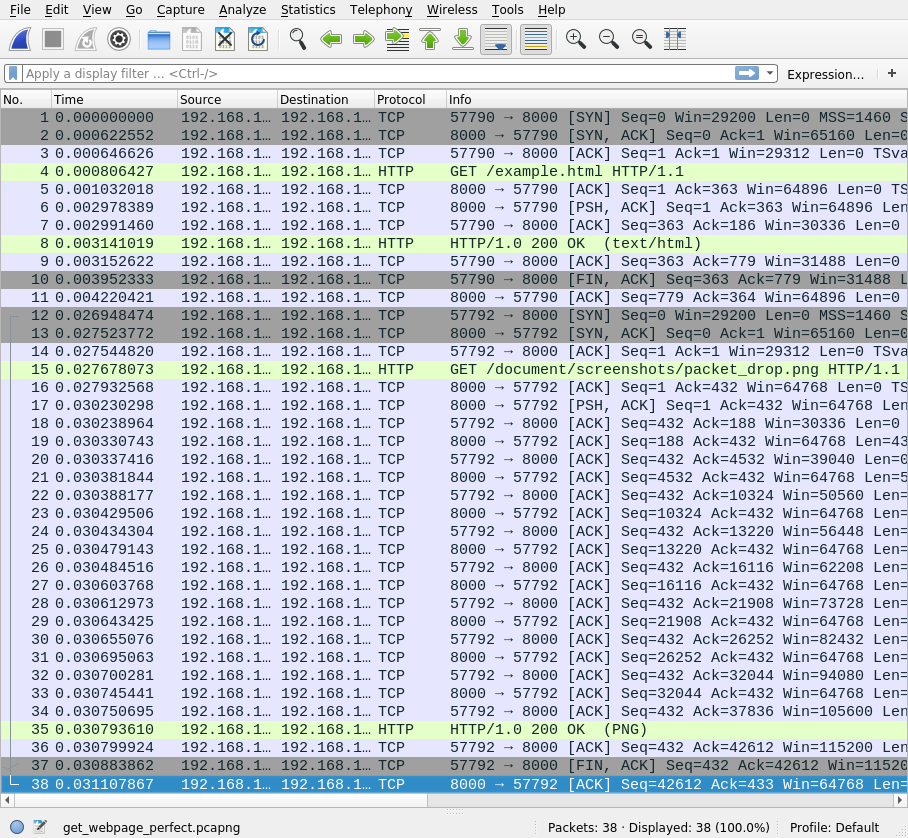
Packets[packetdef] are sequences of ones and zeros sent between computers which are used to transfer data as well as to control how networks function. They consist of different layers of information; the innermost layer contains the data being transferred, and the other layers specify where the packet should go next. When pkts are sent between computers, a certain number of layers are stripped off by, each computer so that it knows where to send the pkt 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 anything. As an example, http transfers websites using html files and images. In particular, there are two pieces of information stored in headers which together define the final destination of the packet: the ipaddr and the port number. The ipaddr 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 were a hotel, the ipaddr would be the address 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. There are this many ports because the port field in tcp and udp is 16 bits long and the maximum value for a 16 bit unsigned integer in 65535. Both tcp and 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. For this reason, udp is used for purposes where speed is more important and missing some data is inconsequential, such as video streaming and playing online games.

I would like to illustrate how ports and packets work using the example of getting a very simple static HTML page with an image inside. The code for the page is shown in Listing [[examplepage]](#examplepage). In Figure [1](#basicwebpage) you can see how the page renders. However, far more interesting is how the browser retrieved the page. In Figure [2](#getrequest) you can see the full sequence of pkts that were exchanged for the browser to get the resources it needed to render the page. The page is hosted using Python3’s http.server module, which is a quick and easy way to serve HTML pages and other content locally, without the hassle of setting up a full blown web server such as Nginx or Apache. Python’s http.server makes the current directory open on port 8000. From there, navigating to /example.html will render the page. Breaking Figure [2](#getrequest) down, pkt 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 pkt 4. The server then acknowledges the request and sends a pkt with the PSH flag set, as shown in pkts 6 and 7. The PSH flag in the tcp header is used to notify the client that the server is ready to push the data (in this case example.html) to the client. The browser then sends back an acknowledgement and the server sends the page as shown in pkts 7 and 8. Finally, the browser sends an acknowledgement of having received the page before initiating a graceful session teardown by sending a FIN ACK pkt, which indicates the end of a session. Having received the FIN ACK packet, the server acknowledges this by sending an ACK packet back to the client, completing the graceful teardown. This process is repeated when the browser parses the HTML and identifies there is an image which it needs to get from the server as well. Because the image is a large file, it takes more pkts to transfer the required data. In Figure [[ladder]](#ladder) you can see a ladder diagram which shows the entire transaction symbolically. I have also colour coded Figure [[ladder]](#ladder) with green arrow heads to the initial handshakes, blue for the HTTP protocol transactions and red for the TCP connection teardown packets.

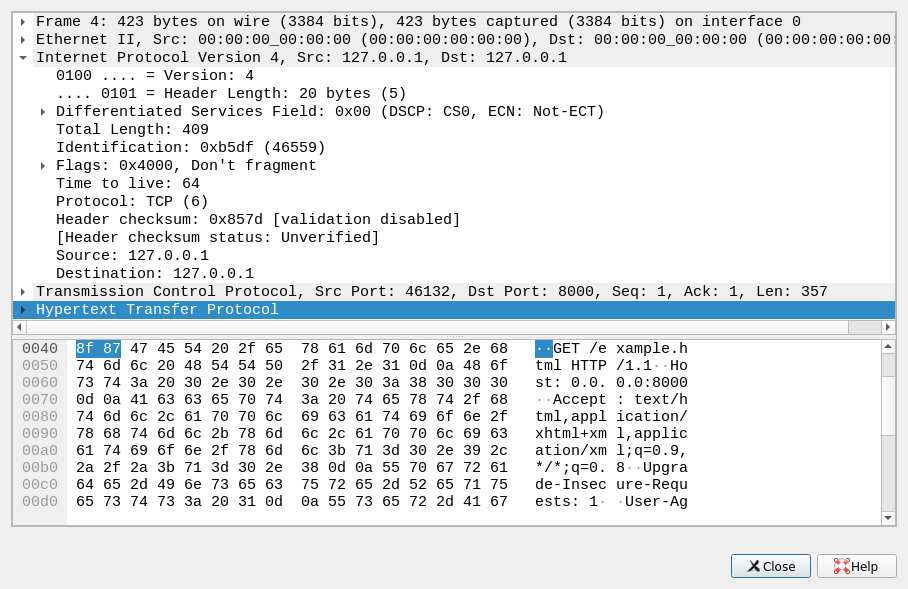
The osi described in Figure [[osi\_model]](#osi_model) can be seen in action in Figures [1](#basicwebpage), [2](#getrequest) and [3](#deconstructed). Figure [1](#basicwebpage) shows levels 6 and 7 of the osi. Levels 4 and 5 can be seen in figure [2](#getrequest) in the form of the tcp session negotiation and transferring the picture and example.html. Level 3/2/1 are shown in Figure [3](#deconstructed) 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.



*A basic static html webpage.*



*A full chain of pkts that shows retrieving a basic webpage from the server.*



*A look inside a TCP pkt.*

<!DOCTYPE html>  
<html>  
<head>  
<title>Wow I can add titles</title>  
</head>  
<body>  
  
<h1>This is a really big heading</h1>  
<p>wow para</p>  
<p>graphs a</p>  
<p>re amazi</p>  
<p>ng</p>  
 <script type="text/javascript">  
 function imgtog() {  
 if (document.getElementById("img").style.display == "none") {  
 document.getElementById("img").style = "block"  
 } else {  
 document.getElementById("img").style.display = "none"  
 }  
 }  
  
 </script>  
  
<img id="img" src="document/screenshots/packet\_drop.png">  
  
<button onclick="imgtog()">Toggle image</button>  
  
  
</body>  
</html>

## Analysis of problem

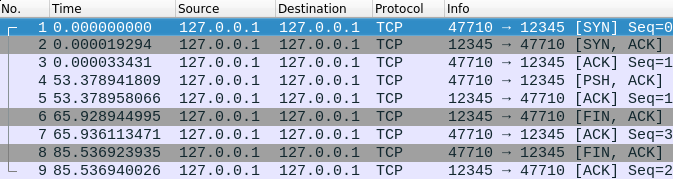
To reiterate, the problem my project tries to solve is how to look at devices on a network from a “bbox” perspective and gain information about what services are running. The difficulty with looking at a network from the outside is that the purpose of the network is to allow communication within the network, thus very little is exposed externally. This presents a challenge, as we want to know not only what machines are on the network, but also what services are running on each machine. This is not always possible, owing to the limited information that services reveal about themselves. Firewalls also play a large part in making network scanning difficult, as sometimes they simply drop pkts instead of sending a tcp RST pkt (reset connection pkt). Dropping a packet means that when a packet is received, no response is sent back as if the connection was just “dropped”. When firewalls drop pkts, it becomes exponentially more difficult to determine the state of any port on the target machine, as you don’t know whether your pkt 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.

### Successful connection over TCP

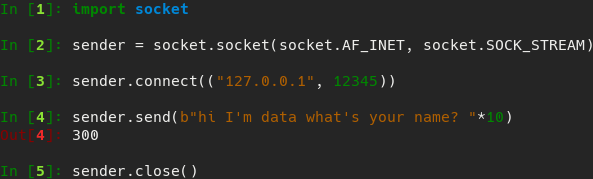
For a tcp connection to be established there is a three-way handshake between the communicating machines. Initially, the machine trying to establish the connection sends a tcp SYN packet to the other machine. This packet holds a dual purpose: to ask for a connection, and, if it is accepted, to SYNchronise 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 SYN, and ACKnowledges the connection request. The sending machine then acknowledges this by sending a final tcp ACK packet back. This connection initialisation is shown in Figure [[data\_transfer]](#data_transfer) 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 [6](#data), where Wireshark allows us to take a look inside the packet to see the data being sent, along with the PSH and ACK flags being set.

The code I used to generate these packet captures is shown in Figures [4](#sender) and [5](#receiver). Breaking the code down, Figure [5](#receiver) shows me initialising a socket object, then I bind it to localhost (127.0.0.1) port 12345. Localhost is just an address which allows connections between programs running on the same computer to be looped back onto the current machine, hence its alternative name: the loopback address. The next line of code instructs the machine to listen for incoming connections. The program accepts the connection in Figure [4](#sender), 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 [4](#sender) then sends some data which we then see printed to the screen in Figure [5](#receiver), both programs then close the connection.

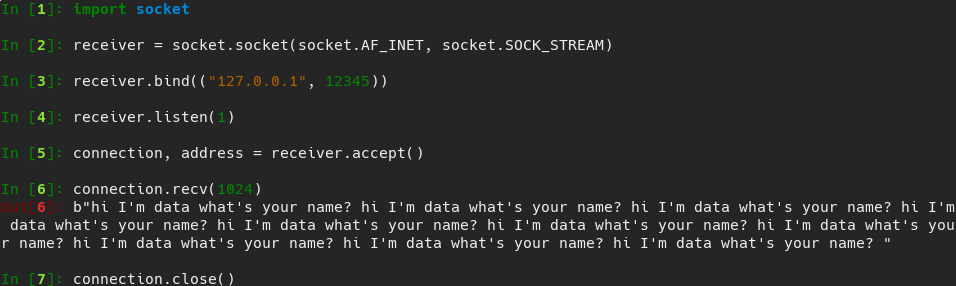


*Packets starting a TCP session, transferring some data then ending it.*

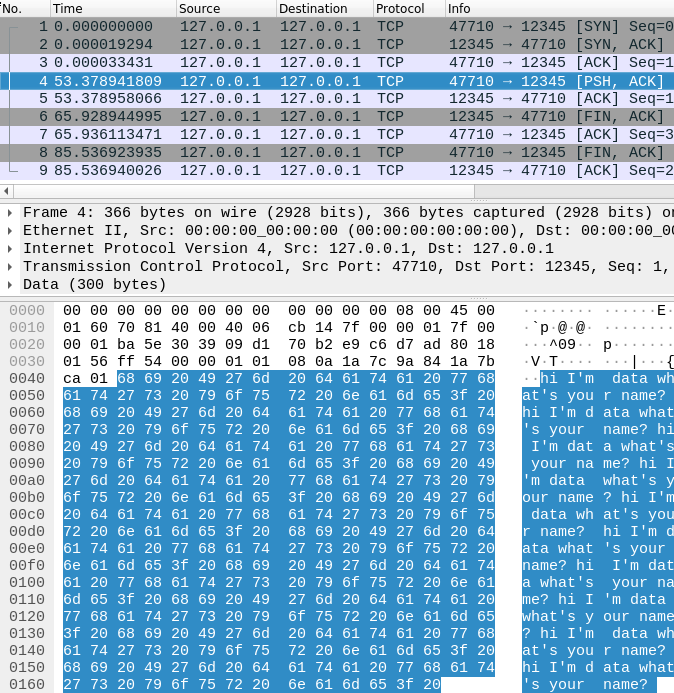
[data\_transfer]



*Transferring some basic text data over a TCP connection.*



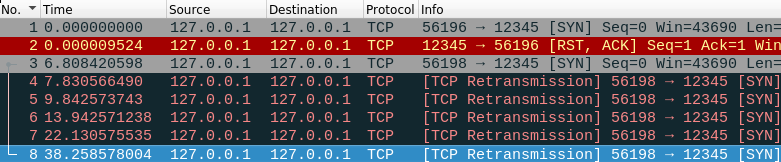
*Receiving some basic text data over a TCP connection.*



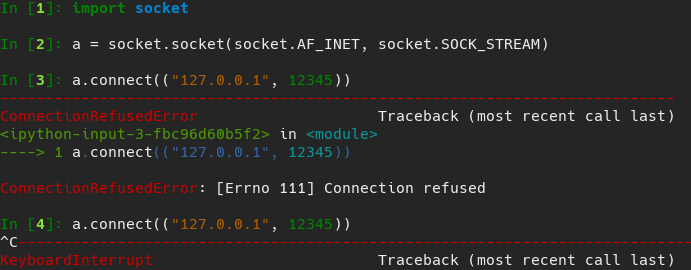
*Highlighted packet carrying the data being transferred in Figure*[*4*](#sender)*.*

### An attempted connection to a closed port

In Figure [7](#firewall) shows my program beginning by sending the same tcp SYN packet as we saw in the attempted connection to an open port discussed above. The difference comes in the next packet with the tcp RST flag being sent back. This flag resets 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 [7](#firewall). The code used to generate this is shown in Figure [[firewall\_code]](#firewall_code); 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 [7](#firewall).



*Attempted connection to a closed port with and without firewall rule to drop pkts.*



*The code used to produce firewall pkt dropping example in Figure*[*7*](#firewall)*.*

[firewall\_code]

### An attempted connection with a firewall rule to drop packets

I would like to commence this section by explaining a bit about firewalls and how they work. Firewalls are essentially the gatekeepers of the internet; they decide whether or not a packet is passed on. Firewalls work by a set of rules which decide what to do with a particular packet. Such a rule might be that it is coming from a certain ipaddr or has a particular destination port. The actions taken after the packet has had its 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 and lets the packet through; DROP simply 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 https. For demonstration purposes I will be using a Linux utility called iptables for implementing all firewall rules on my system. Packet number three in Figure [7](#firewall) corresponds to line 4 of the code in Figure [[firewall\_code]](#firewall_code), the difference being that in Figure [[firewall\_code]](#firewall_code) I have enabled a firewall rule to drop all pkts 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 directs that all pkts arriving (-I INPUT) with source address 127.0.0.1 (-s 127.0.0.1) are dropped, with no response sent (-j DROP). With this firewall rule in place you can see in Figure [7](#firewall), pkt 3 receives no response, and as such Python assumes that the pkt just got lost and tries to send the pkt again repeatedly. This continued for more than 30 seconds before I stopped it, as shown by the time column in Figure [7](#firewall), and the final KeyboardInterrupt in Figure [[firewall\_code]](#firewall_code). The amount of time that a system will continue trying to reconnect depends on the OS and other factors, but the minimum time is 100 seconds, as specified by RFC 1122. On most systems, the timeout is between 13 and 30 minutes, according to the Linux manual page on tcp reproduced below.

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.

### Project aims and methods

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 of ports on remote machines (as in whether the port is open/closed or filtered etc.); detect which hosts are up on a subnet; and 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 such as f-strings which are not in even fairly recent versions such as 3.5. F-strings allow for a clear and consistent string formatting syntax, which I will use extensively. I have chosen Python in particular, because it is very readable and has extensive low level bindings to kernel syscalls with the socket module allowing me to write code quickly that is easily understandable and has a clear purpose. Python enables me to use low level networking functions, and even change 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 icmp. These features combine to make Python a great language for writing networking software with a high level of abstraction. In regards to the osi, my code will sit with the user interface at level 7 specifying what to do at a high level, and the actual scanning takes place at levels 3, 4 and 5, with host detection being at level 3. Port scanning will be taking place at level 4 for tcp SYN scanning and udp scanning, whereas connect() scanning and version detection will sit at level 5. Finally, I will look at what is actually handling all of the networking on my machine. My machine runs Linux, and as such all networking is handled by system calls to the Linux kernel. For example, the socket.connect method is just a call to the underlying Linux kernel’s connect syscall, but presents a kinder call signature to the user, as the Python socket library does some processing before the syscall is made.

## Success Criteria

1. Probe another computer’s networking from a bbox perspective. [blackbox]
2. To help the user with usage/help messages when prompted. [usage]
3. Translate cidr-specified subnets into a list of domains. [cidr]
4. Send icmp ECHO requests to determine whether a machine is active or not. [ping]
5. Perform any scan type without first checking whether the host is up. [nocheck]
6. Detect whether a TCP port is open (can be connected to). [tcpopen]
7. Detect whether a TCP port is closed (will refuse connections). [tcpclosed]
8. Detect whether a TCP port is filtered (a firewall is preventing or monitoring access). [tcpfiltered]
9. Detect whether a UDP port is open (can be connected to). [udpopen]
10. Detect whether a UDP port is closed (will refuse connections). [udpclosed]
11. Detect whether a UDP port is filtered (a firewall is preventing or monitoring access). [udpfiltered]
12. Detect the operating system of another machine on the network solely from sending packets to the machine and interpreting the responses. [osdetect]
13. Detect what service is listening behind a port. [servicedetect]
14. Detect the version of the service running behind a port. [versiondetect]

## Description of existing solutions

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

* tcp: SYN
* tcp: Connect()
* tcp: ACK
* tcp: Window
* tcp: Maimon
* tcp: Null
* tcp: FIN
* tcp: Xmas
* udp
* Zombie host/idle
* sctp: INIT
* sctp: COOKIE-ECHO
* IP protocol scan
* ftp: bounce scan

As well as supporting a vast array of scanning types, Nmap can also perform service and version detection, and operating system detection via custom probes. It 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 pkts must be sent to a sequence of ports in order before access to the finalportis allowed). It supports a plethora of options to avoid firewalls or ids, such as sending pkts with spoofed csums/source addresses, and sending decoy probes. Nmap can do many more things than I have listed above, and indeed there is an entire book on using nmap (<https://nmap.org/book/>). Figure [[nmapflow]](#nmapflow) shows the logical structure behind nmap scanning a network.

The following paragraphs discuss an example nmap scan I did on my home network I did for this project. The command I used was:  
nmap -sC -sV -oA networkscan 192.168.1.0/24. The purpose of the command line flags was 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. The program outputs to three files: networkscan.(nmap,gnmap,xml). Before I go into what each file contains, I will explain some terminology: something is greppable if it can be easily searched with the Linux utility grep. Grep stands for Globally search a Regular Expression and Print, which instructs the computer to search the specified file for lines that contain a certain word or pattern; for example, the command to find all lines with the word “hi” in them in the file “document” would be grep `hi' document.

Returning to the files nmap created: 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(DNSVSF: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(DNSStatusReq  
uestTCP,E,"\0\x0c\0\0\x90\x04\0\0SF:\0\0\0\0\0\0");  
Service Info: CPE: cpe:/o:asus:wrt\_firmware

The above is the report for only one device on the network. In it, you can see information such as which ports are open, and what services are running behind them. As this example device is network router, you can see port 8443, which nmap has recognised to be hosting the ASUS web admin, from which you can configure the router. There follows some other associated information extracted from the server. Most of this extra information is derived from the -sC flag, which enables 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. 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 explains why nmap is so good at recognising services: it has a lot of data to look at and learn from in regards to service fingerprinting.

Networkscan.gnmap contains exactly the same information as  
networkscan.nmap, but the output is formatted to enable easy searching with the grep utility. I reproduce part of the output file below:

# 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:  
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)

As you can see above, all of the information is on a single line for each type of scan. This is useful: for example, if you want to scan a large number of hosts and just want to know which hosts are up you could use  
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

This shows the hosts which are online and their host names. Other ways to use the greppable output format would be to search for 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, the .gnmap output is useful for when you want to use grep to filter results.

Finally, we have the xml format file,  
networkscan.xml part of which is reproduced below:

<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE nmaprun>  
<?xml-stylesheet href="file:///usr/bin/../share/nmap/nmap.xsl" type="text/xsl"?>  
<!-- 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 -->  
<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">  
<verbose level="0"/>  
<debugging level="0"/>  
<host starttime="1554921379" endtime="1554923187"><status state="up" reason="syn-ack" reason\_ttl="0"/>  
<address addr="192.168.1.1" addrtype="ipv4"/>  
<hostnames>  
<hostname name="router.asus.com" type="PTR"/>  
</hostnames>  
<ports><extraports state="closed" count="995">  
<extrareasons reason="conn-refused" count="995"/>  
</extraports>  
<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  
-pc-Linux-gnu%r(DNSVersionBindReqTCP,20,&quot;\0\x1e\0\x06\x85\x85\0  
\x01\0\0\0\0\0\0\x07version\x04bind\0\0\x10\0\x03&quot;)%r  
(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>  
</script></port>

It can be seen that this file is extremely verbose. It contains the reason why each port has the state it does, as well as a vast amount of other data that the other scans did not include. The consequence is that this output format is not very easy for humans to read, meaning that this format is available because it is easier for other programs to parse than the other formats. As well as this the extra information 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, it is an application at level 7 when the user interacts, with it but uses several libraries that interact at level 2, which it uses to get the raw headers of the packets being sent and thus gain information from them (see Figure [[nmapblock]](#nmapblock)). Nmap has virtually no competitors in the Linux ecosystem, other than possibly Angry IP Scanner, which is another open source network scanner, except it has a much smaller user base.

Before describing my program in detail I would like to explain some terminology I will use: “parse the arguments” means taking the string of text that the user enters after the program name i.e. program <text>. It is these texts that represent the arguments. Parsing the arguments means turning those strings into useful information that the program can use. For example, my program will allow people to enter the port number(s) they want to scan. I want them to be able to do this by specifying a range of ports. If the user specifies 10-20, this would mean ports 10, 11, …, 20. Thus an example of parsing would be the turning of 10-20 into the list of numbers from 10 to 20 as shown in Algorithm [[argumentparsing]](#argumentparsing) below. “Probes” refer to the actual packets being sent to the server; I will refer to anything sent from my code to another machine as being a “probe”. I will use the term “hosts” to mean the other machines on the network that we are scanning.

## Prospective Users

The prospective users of my program would be system administrators, penetration testers or network engineers. In my particular case, prospective users would be my school’s system administrators. It would allow them to see an outsider’s 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. Banners are short strings of text which a service or program will send to identify itself when it receives a new connection. They often contain information such as protocol version etc., which allows the connecting client to know how to communicate with the service. However, they can also reveal too much information, such as the version number of the service running. If the service version is old, then it is likely that bugs will have been found in that version of the program. This information could allow an attacker to gain access to the server by exploiting the vulnerability in that service. This can obviously be prevented by keeping services up to date; however, that is not always possible, so as a best practice banners should reveal the minimum amount of information possible such that the client can interact with the service.

I plan to use my school’s system administrators users in order to gain some feedback as to the usability and performance of my program.

## Data Dictionary

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)
* The probes to be used for version detection

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 1000 ports of each machine. For the purpose of RAM estimation I will not consider version detection, as I am unsure of how I will implement it currently. To measure the size of an 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.

>>> 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  
>>> len(hosts)\*(sys.getsizeof(True)) / 2\*\*10  
7.0  
>>> pings[0]  
'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'  
>>> from binascii import unhexlify  
>>> ping = unhexlify(pings[0].replace(" ", "")) # turn the string of numbers into a bytes object  
>>> sys.getsizeof(ping)  
117  
>>> len(hosts)\*sys.getsizeof(ping) / 2\*\*10  
29.25  
>>> 2278.0 + 7.0 + 29.25 + 2.22  
2316.47

As shown above in Listing [[testingsize]](#testingsize), 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 over than ninety eight percent of the total space used by the mock data I created. However, a total of 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 |

## Data Flow Diagram

In my application there will be three-way information flow:

1. Sending packets out from my application
2. Receiving packets back from the targets
3. Transferring data between functions

My program will only hold information in memory and provides no utility for saving the information from scans. This is because on the target systems (Linux/Unix based machines), the shell which is used to run commands has a very simple way of placing output in files by use of Unix “pipes”, which are how Unix-based operating systems handle interprocess communications. If nmap did not have a dedicated saving utility, the following command could be used to save the output of nmap to a text file called outputfile:  
nmap 192.168.1.0 > outputfile. Thus, in my application, for the sake of simplicity, this is the method I will use to save output.

Figure [[dataflow]](#dataflow) below shows the proposed data flow in my program.

## Description of Solution Details

As already stated above on page , I will be using Python version 3.7.2 for my project because I am already familiar with Python’s syntax and its socket library has a very nice high level api for making system calls to the kernel’s low level networking functions. This makes it ideal for a networking project like mine, as it allows me to prototype easily, and explore many ideas about how I could implement my solution in a time-efficient manner.

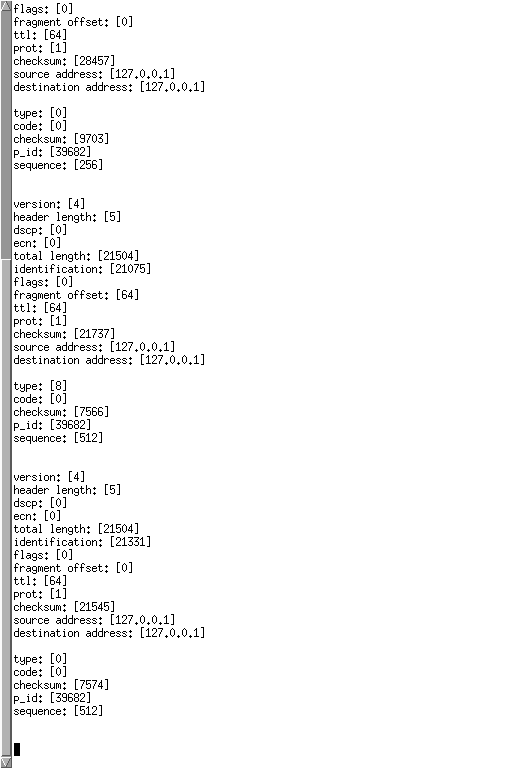
I decided to start my project by researching how to write code for receiving and sending icmp echo requests (i.e. pings). icmp sits at layer 3 of the osi. This means that it functions at a layer below that to which you are normally given access in the socket module. Sending an icmp echo request requires a raw socket. A raw socket is one which will return everything contained by the ethernet/wifi frame, including the raw ip headers. The bytes object received from a raw socket thus requires unpacking to extract relevant information. The struct module provides a convenient api for converting between packed values, because there is usually a difference in endianness between the network and the local machine which requires translation.

Interactions with the socket module are mainly through the pack and unpack functions. For each of these functions it is necessary to provide a format specifier defining how to unpack/pack the bytes/values. In Listing [[echosend]](#echosend), 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 in Listing [[echorecv]](#echorecv), which uses struct.unpack to unpack value from the received icmp packet before printing the fields out to the terminal. Listing [[echosend]](#echosend) makes use of an IP checksum function which I wrote (see Listing [[checksum]](#checksum) below). In Figure [8](#echodissect), you can see the output when I run the command ping 127.0.0.1 which the code in Listing [[echorecv]](#echorecv) is listening for packets.

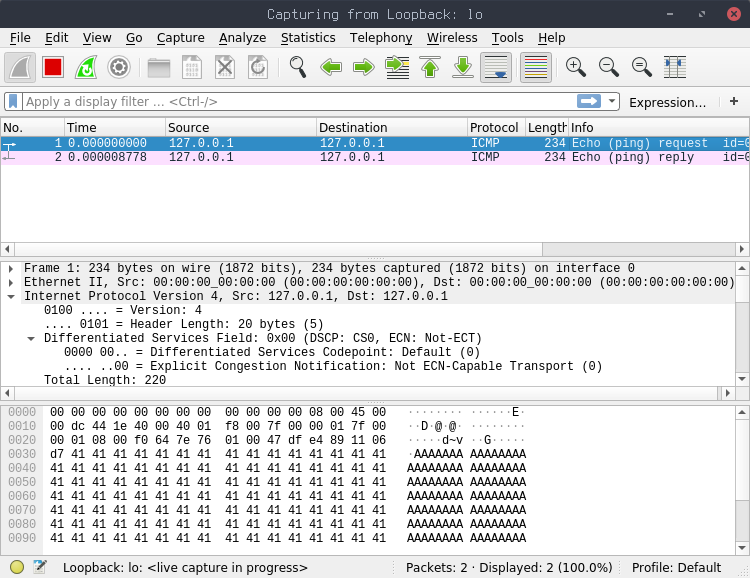
#!/usr/bin/Python3.7  
import socket  
import struct  
import os  
import time  
import array  
  
from os import getcwd, getpid  
import sys  
sys.path.append("../modules/")  
  
import ip\_utils  
  
  
ICMP\_ECHO\_REQUEST = 8  
  
# opens a raw socket for the ICMP protocol  
ping\_sock = socket.socket(socket.AF\_INET, 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  
  
# the two zeros are the code and the dummy checksum, the one is the sequence number  
dummy\_header = struct.pack("bbHHh", ICMP\_ECHO\_REQUEST, 0, 0, ID, 1)  
  
data = struct.pack("d", time.time()) + bytes((192 - struct.calcsize("d")) \* "A", "ascii")  
  
checksum = ip\_utils.ip\_checksum(dummy\_header+data)  
  
header = struct.pack("bbHHh", ICMP\_ECHO\_REQUEST, 0, checksum, ID, 1)  
  
packet = header + data  
  
ping\_sock.sendto(packet, ("127.0.0.1", 1))

#!/usr/bin/Python3.7  
  
import socket  
import struct  
import time  
from typing import List  
  
# socket object using an IPV4 address, using only raw socket access, set ICMP protocol   
ping\_sock = socket.socket(socket.AF\_INET, socket.SOCK\_RAW, socket.IPPROTO\_ICMP)  
  
packets: List[bytes] = []  
  
while len(packets) < 1:  
 recPacket, addr = ping\_sock.recvfrom(1024)  
 ip\_header = recPacket[:20]  
 icmp\_header = recPacket[20:28]  
  
 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)  
  
 hl\_v = f"{ip\_hp\_ip\_v:08b}"  
 ip\_v = int(hl\_v[:4], 2)  
 ip\_hl = int(hl\_v[4:], 2)  
 dscp\_ecn = f"{ip\_dscp\_ip\_ecn:08b}"  
 ip\_dscp = int(dscp\_ecn[:6], 2)  
 ip\_ecn = int(dscp\_ecn[6:], 2)  
 flgs\_off = f"{ip\_flgs\_ip\_off:016b}"  
 ip\_flgs = int(flgs\_off[:3],2)  
 ip\_off = int(flgs\_off[3:], 2)  
 src\_addr = socket.inet\_ntoa(struct.pack('!I', ip\_src))  
 dst\_addr = socket.inet\_ntoa(struct.pack('!I', ip\_dst))  
  
 print("IP header:")  
 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")  
  
 msg\_type, code, checksum, p\_id, sequence = struct.unpack('!bbHHh', icmp\_header)  
 print("ICMP header:")  
 print(f"Type: [{msg\_type}]\nCode: [{code}]\nChecksum: [{checksum:04x}]\nProcess ID: [{p\_id:04x}]\nSequence: [{sequence}]"  
 packets.append(recPacket)  
open("current\_packet", "w").write("\n".join(" ".join(map(lambda x: "{x:02x}", map(int, i))) for i in packets))

def ip\_checksum(packet: bytes) -> int:  
 """  
 ip\_checksum function takes in a packet  
 and returns the checksum.  
 """  
 if len(packet) % 2 == 1:  
 # if the length of the packet is odd, add a NULL byte  
 # to the end as padding to make it even in length  
 packet += b"\0"  
  
 total = 0  
 for first, second in (  
 packet[i:i+2]  
 for i in range(0, len(packet), 2)  
 ):  
 total += (first << 8) + second  
  
 # calculate the number of times a  
 # carry bit was added and add it back on  
 carried = (total - (total & 0xFFFF)) >> 16  
 total &= 0xFFFF  
 total += carried  
  
 if total > 0xFFFF:  
 # adding the carries generated a carry  
 total &= 0xFFFF  
 total += 1  
  
 # invert the checksum and take the last 16 bits  
 return (~total & 0xFFFF)



*Dissecting an icmp echo request packet.*



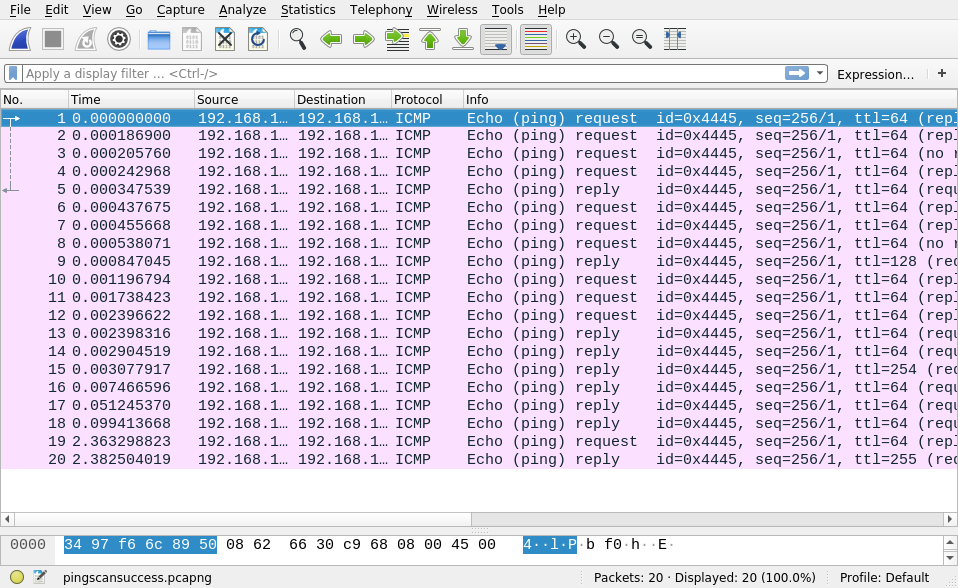
*Screenshot of Wireshark showing a successful send of an icmp echo request packet.*



*Screenshot showing me first successfully dissecting an icmp echo request packet.*

Having written a prototype program which was capable of sending an icmp echo request and another prototype which was capable of receiving and unpacking it. Having done these prototypes I identified that it would be best to abstract the code for dissecting all the headers i.e. icmp, tcp and ip into classes where I can just pass the received packet into the class and have the class dissect it for me. This will also give me access to some of the benefits of classes, such as the \_\_repr\_\_ method, which is called when you print classes out and allows control over what is printed out. Before embarking on the final program, I decided to write a prototype ping scanner, as this would allow me to get a feel for making a scanner, and to further exploring low level protocol interactions.

#!/usr/bin/Python3.7  
from os import getcwd, getpid  
import sys  
sys.path.append("../modules/")  
  
import ip\_utils  
  
import socket  
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), 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)))



*Screenshot of Wireshark showing a successful ping scan.*

$ sudo ./ping\_scan.py  
host: [192.168.1.1] responded to an ICMP ECHO REQUEST in 0.00037 seconds, ttl: [64]  
host: [192.168.1.35] responded to an ICMP ECHO REQUEST in 0.00042 seconds, ttl: [128]  
host: [192.168.1.37] responded to an ICMP ECHO REQUEST in 0.002 seconds, ttl: [64]  
host: [192.168.1.117] responded to an ICMP ECHO REQUEST in 0.0017 seconds, ttl: [64]  
host: [192.168.1.176] responded to an ICMP ECHO REQUEST in 0.0014 seconds, ttl: [254]  
host: [192.168.1.14] responded to an ICMP ECHO REQUEST in 0.0072 seconds, ttl: [64]  
host: [192.168.1.246] responded to an ICMP ECHO REQUEST in 0.049 seconds, ttl: [64]  
host: [192.168.1.8] responded to an ICMP ECHO REQUEST in 0.099 seconds, ttl: [64]

Completion of these prototypes has given me an understanding of 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 starts 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.

## Acceptable Limitations

My original concept included dedicated operating system detection as an option. However, if I find that time is short having implemented version detection, it will be an acceptable limitation to omit operating system detection. This is because version detection can provide some indication of the operating system in use, by virtue of the fact that some services are restricted to certain operating systems. For example, if the scanner detected that the service ActiveSync was in use, this would indicate that the system being scanned was a Windows system, which is reflected in the match directive and attached CPE information for ActiveSync:

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:ac  
tivesync/ cpe:/o:microsoft:windows/a

## Test Strategy

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

1. Unit testing
2. Wireshark

I will employ two separate testing strategies because they are good at different things, both of which I need in order to show that my project works. First, I will use unit testing to test some general purpose functions, which are pure functions (are independent of the current state of the machine).

I will use Wireshark to test those parts of the program which involve the use of impure functions and low level networking. Wireshark makes this easy by allowing capture of all the pkts going over the wire. As well as this, it has a vast array of pkt decoders (2231 in my install), which it can use to dissect almost any pkt that is on the network. Wireshark will allow me to see my scanners sending packets, and to check whether the parsers I have written for the various protocol are working. I can also check that the csums in each of the various protocols are valid as Wireshark is capable of performing csum verification for a wide variety of protocols.

I will be running these tests on my laptop, which is a Thinkpad T480 running Arch Linux with kernel version 5.0.7. I have installed the following versions of each of the programs I will be using to test my code: Wireshark 3.1.0, Python 3.7.2 and PyTest 4.3.1. I am also using pyenv version 1.2.9 to manage the version of Python in my Python environment. I plan not to use any modules outside of the Python standard library, so that my program is as portable as possible and its functionality is as reproducible as possible.

# Design

## 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 socket.connect it and sees whether it can connect or not. 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 pkts to. It can then differentiate between open, open|filtered, filtered and closed ports, based on whether it receives a pkt back and what flags are set in the received pkt. Flags are parts of tcp pkts that constitute a one byte long section, which store “flags” where each bit in the byte represents a different flag.

## Design of User Interface

I am designing 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 has the advantage that anything learnt using either tool is applicable to both, which benefits everyone.

Based on this perception, I plan to use the same option flags as nmap, as well as similar help messages and an almost identical call signature (how the program is used on the command line).  
Running ./netscan.py <options> <target specification> should be almost identical to nmap <options> <target specification> in terms of which scan types will be run, which hosts will be scanned and which ports are scanned. Below, you can see a concept help message, for my program with all the arguments I plan to implement.

usage: netscan.py <options> <target specification>  
  
required arguments: target specification  
  
optional arguments: -h, --help -Pn, -sL, -sn, -sS,  
 -sT, -sU, -sV, -p, --ports, -O  
 --exclude\_ports

The above shows clearly which are required arguments, and which are optional ones. It also shows that some some arguments can be called with either a short format e.g. -p or with a more 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 should easier to recall the function of the more verbose flags. If the user enters erroneous data, they should be greeted by a ValueError, which will explain exactly what the issue was with their input, and will print out the argument that caused the error.

## System Algorithms

## Input data validation

I plan to perform data validation in all of the functions in the fundamental modules which will hold the basic functionality for my project, such as the scanning functions. This is because my project will revolve heavily around these functions and they will need to be as error free as possible. Adding input validation to these core functions will enable me to find errors in my code earlier. For example, passing a function a list of strings instead of just a string might work in some cases, but the function will have a completely different result. These types of programming errors can be quite hard to debug, as although they may not generate errors very often but on occasions they will still break the application. Although data validation helps when programming, it will mainly be there to guide the user by showing them where in their arguments the problem is. This is far more useful than some programs, which simply exit with no information, beyond the fact that error occurred.

An example for a Python ValueError could be trying to turn the string  
"tacos" into an integer. This will result in the following error message:  
ValueError: invalid literal for int() with base 10: `tacos'  
This informs you that you have tried to turn "tacos" into an integer with base 10, which is invalid. This is a clear and helpful error message, because it tells you what you tried to do that went wrong, and which argument was the one that caused the error.

## Algorithm for complex structures

[ip\_range]

# Technical Solution

I have placed all of my code in Appendix [6](#code). I will be going through each of the items in this appendix and explaining what they do.

Appendix [6.1](#app:icmpping) contains all the code which I wrote while in an early experimentation phase where I was testing out how I was planning to make and structure the project.

Appendix [6.2](#app:pingscanner) contains all the code which I wrote while writing the initial prototype of my ping scanner which uses icmp echo request messages to detect hosts which are online on a given subnet. This is used to meet success criterion [[ping]](#ping).

Appendix [6.3](#app:subnettoaddresses) contains all the code which I wrote while writing a tool to translate a cidr-specified subnet into the list of ip addresses for that subnet. It uses logic to exclude the broadcast address and host addresses for each subnet. This is used to meet success criterion [[cidr]](#cidr).

Appendix [6.4](#app:tcpscan) contains all of the prototypes for tcp-based scanning, which are contained in the sub-appendices [6.4.1](#app:connectscan) and [6.4.2](#app:synscan). Appendix [6.4.1](#app:connectscan) contains all of the code which I created whilst prototyping connect scanning. It satisfies success criteria [[tcpopen]](#tcpopen) and [[tcpclosed]](#tcpclosed). Appendix [6.4.2](#app:synscan) contains all of the code I wrote while prototyping tcp SYN scanning. It satisfies success criteria [[tcpopen]](#tcpopen), [[tcpclosed]](#tcpclosed) and [[tcpfiltered]](#tcpfiltered).

Appendix [6.5](#app:udpscan) contains all of the code I wrote while prototyping udp scanning. It satisfies success criteria [[udpopen]](#udpopen), [[udpclosed]](#udpclosed) and [[udpfiltered]](#udpfiltered).

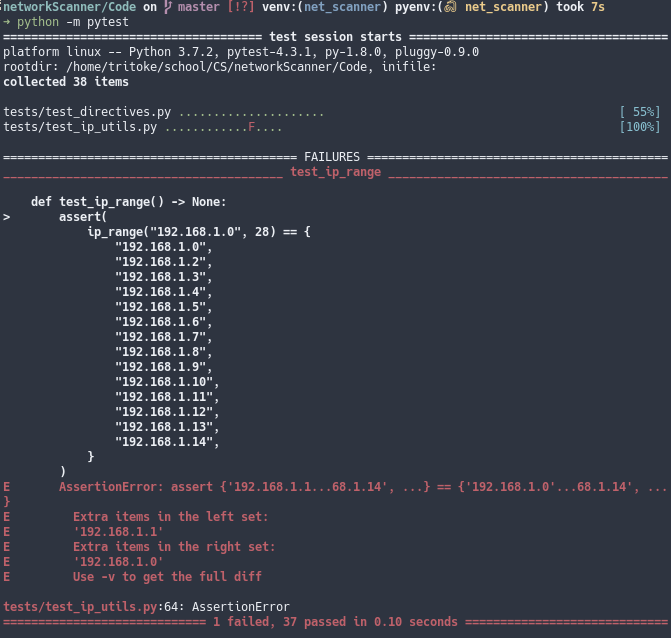
Appendix [6.6](#app:versiondetection) contains all of the code I wrote while prototyping version detection scanning. It satisfies success criteria [[servicedetect]](#servicedetect) and [[versiondetect]](#versiondetect).

Appendix [6.7](#app:modules) contains all of the modules I wrote to help with creating my main application, described later. These modules mainly contain code which I reuse often, such as code to calculate an ip checksum, or validate an ipaddr.

Appendix [6.8](#app:examples) contains a script I wrote which will run each of the prototype applications I made. This doesn’t satisfy any of the success criteria, but was very useful for solving issues I had with importing Python modules, where owing to the directory structure everything has to be started from the root of the directory structure, otherwise errors occur. My solution for eliminating the errors was to run everything at the root of the directory structure, as this can call the main() function defined in each of the modules, and can also import all of the modules in the modules directory.

Appendix [6.9](#app:netscan) contains the code for my final application. This satisfies all of the success criteria apart from [[osdetect]](#osdetect), which as previously discussed, is alternatively completed via version detection scanning.

Appendix [6.10](#app:tests) contains all of the code for my unit tests which I run using  
Python -m pytest. It automatically runs each function, and delivers verbose information on each one. I have deliberately named all of the test functions in a very verbose way and I only test one thing in each function. This means that it is much easier for me to read from the name of a failed test exactly what went wrong with what function and what argument caused it. An example of this can be seen in Figure [12](#testing), where I have changed one of the tests so that it fails. You can see that the output of the test shows me a clear difference between what was expected on one side of the assertion statement and then what actually happened on the other side. In this case, it shows that in the left set there is an extra element “192.168.1.1” and in the right an extra element “192.168.1.0”. This is very helpful for preventing regressions in the code. Until I wrote these unit tests, I found that I would write a new feature and accidentally break another piece of functionality as a consequence.



*A screenshot of running pytest with a deliberately broken test.*

# Testing

## Test Plan

I will be testing my application using a combination of unit tests, and Wireshark where applicable. Unit tests are more suitable for doing tests on specific functions to make sure that regressions don’t occur while developing the application. A regression is a when a feature or change that was implemented into the program has accidental consequences that cause the application to break. I will use Wireshark to show the scanning portion of my code, and when external connections are made/custom packets created. Section [4.2](#wiresharktests) contains the tests using Wireshark; the unit tests are in Appendix [6.10](#app:tests).

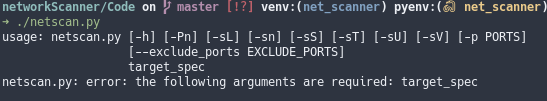
The Wireshark testing will require Wireshark and iptables. I will need to set up Wireshark in listen mode on the right interface so that it captures the packets that my program is sending. From there I will be able to inspect the sent packets and determine whether they fit what was expected in the test description or whether they don’t match at all. For filtered packet tests I will need to run the command iptables -I INPUT -s 127.0.0.1 -j DROP and scan the localhost address, and after the test I will need to run the command iptables -F to flush all the iptables rules to prevent any confusion in future caused by an firewall rule that shouldn’t be there.

To running the unit tests will require Python 3.7.2 and pytest 4.3.1. To run the tests I will need to run python -m pytest inside the Code directory. This will call pytest, which will find the tests inside the tests directory and run them. It will then display the number of tests that passed along with lots of information on the tests that failed, such as what the arguments were etc. Pytest does this via introspection of the comparison and assert commands. This means that it uses its own versions of those commands which allow it to get more information out about what went wrong, such as which element in a list was the one that caused the comparison to return false etc.

## Testing Evidence

### Printing a usage message when run without parameters

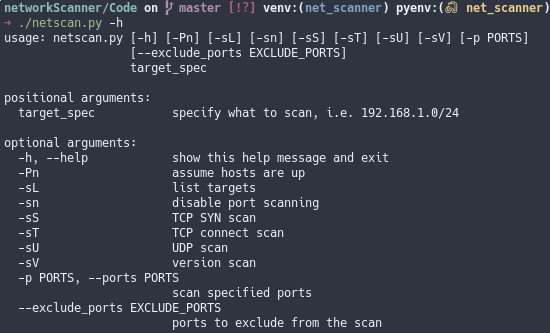
To show this I will run my program passing it no parameters. This should print out a message of the form: USAGE: ./<program> <required> <parameters> where everything in angle brackets should be replaced by what is necessary for my program. In Figure [13](#noparametertest) you can see me run ./netscan.py with no parameters and it prints out the required usage message telling me that I am missing the target\_spec parameter, this shows that it passed this test. This shows success criterion [[usage]](#usage).



*Screenshot showing my program being run without parameters.*

### Printing a help message when passed -h

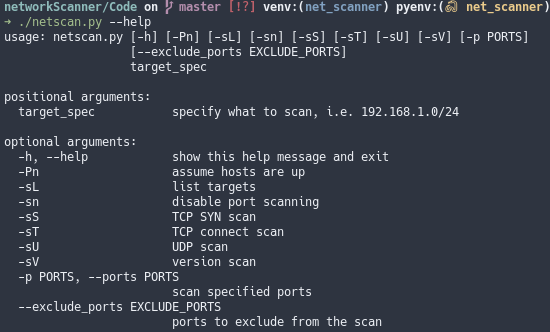
To show this I will run my program with the -h flag. This should print out a message showing each of the options as well as what each of them do. It should also print out whether they are positional arguments or optional arguments and if an argument can have two forms then it should print out both forms of the flag, i.e. -p --ports. In Figure [14](#hflagtest) you can see me run my program with the -h flag and it proceeds to print of a help message with messages with what each option is for as well as short and long form of arguments, this shows my program passed this test. This shows success criterion [[usage]](#usage).



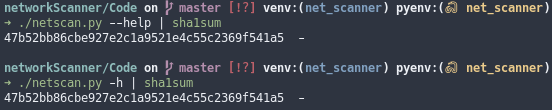
*Screenshot showing my program being run with the -h flag.*

### Printing a help message when passed -help

To show this I will run my program with the --help flag. This should produce the same output as with -h. This shows the same message as in the test of -h. To prove this, if I take the sha1sum of the output for both flags, we can see that the hashes are identical and therefore the originals were also identical; this is shown in Figure [16](#messagehash). This shows success criterion [[usage]](#usage).



*Screenshot showing my program being run with the help flag.*



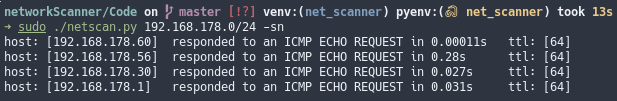
*Screenshot showing the hashes of the two help messages.*

### Scanning a subnet with ICMP echo request messages

To show this I will run my program with the -sn flag and specify the subnet of my local network 192.168.178.0/24. This should produce a list of all the hosts which are up on the network. In Figure [17](#lanscantest) you can see you can see my program’s output showing that the hosts:

* 192.168.178.60
* 192.168.178.56
* 192.168.178.30
* 192.168.178.1

all responded with icmp echo reply messages. This is reflected in a packet capture I took while performing the scan. A section of this scan is shown in Figure [18](#lanscanWireshark), where you can see some of icmp echo request messages my program sent, along with some of the requests to hosts that don’t exist. Note the different addresses in the source and destination fields, and the Echo (ping) request vs reply in the info column. This meets success criteria [[blackbox]](#blackbox) and [[ping]](#ping).



*Screenshot showing the output of a scan of my local network.*

*Screenshot showing a selection of the packets being sent by this scan.*

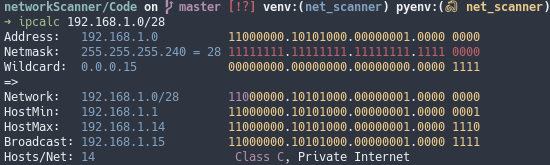
*Screenshot showing a selection of the packets being sent by this scan.*

### Translating a CIDR-specified subnet into a list of IP addresses

To show this I will run my program with the -sL flag and I will specify a small subnet of 192.168.1.0/28 (I have chosen such a small subnet such that it will fit on my terminal and therefore in a screenshot). I expect the list of addresses to be 192.168.1.1 - 192.168.1.14. To prove that my program works, I will screenshot the output when run with the stated parameters and I will use a website to translate the same subnet and show that it displays the same addresses as my program. In Figure [19](#cidrtest), you can see that the output from my program matches the expected list of IP addresses from 192.168.1.1 to 192.168.1.14 which is also shown by the screen shot of the same subnet translated by the ipcalc utility on Linux. This proves my program works and covers success criterion [[cidr]](#cidr).



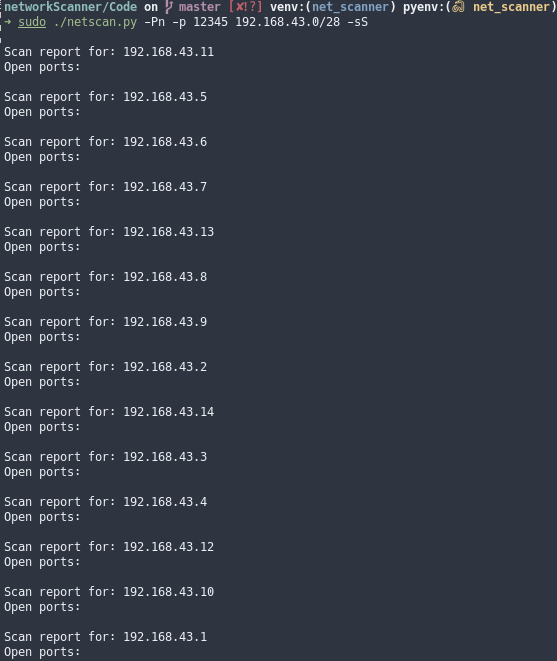
*Screenshot showing the output of my program when asked to translate the subnet 192.168.1.0/28.*



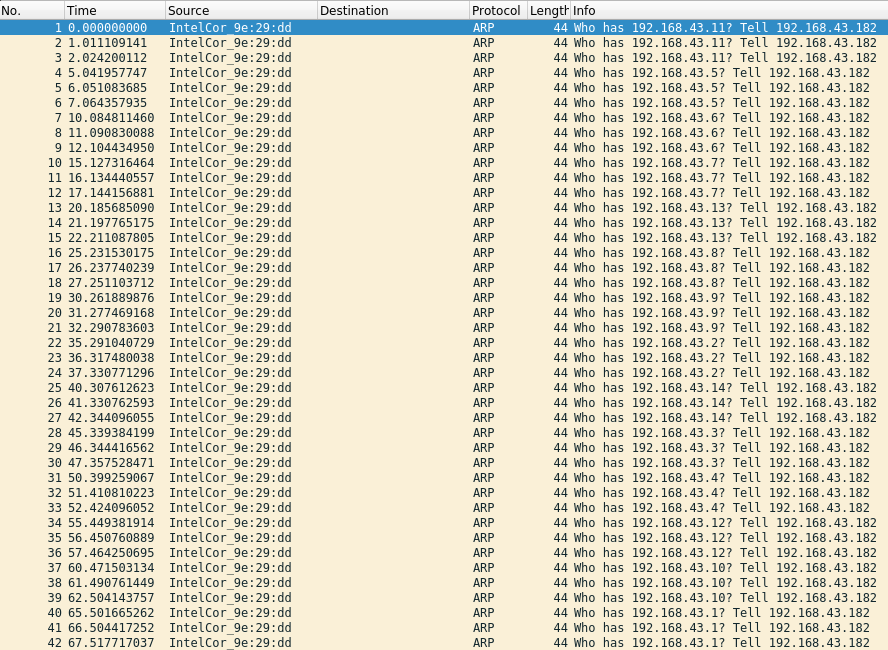
*Screenshot showing the range displayed by the ipcalc utility when asked to calculate the same subnet.*

### Scanning without first checking whether hosts are up.

To show this I will perform a TCP scan on a small subnet where I know there are no hosts and show that the scan continues despite there actually being no host on the other end. To do this I will pass the -Pn flag and I will specify the subnet 192.168.43.0/28 which I know has no has no hosts on it. I will also specify -p 12345 to only scan port 12345 so that there are fewer requests in the packet capture. Finally I will specify -sS to do tcp SYN SCANNING.I expect to see a multiple of 14 arp messages. This is because I don’t know how many times my nic will retry at getting the destination mac address. It needs the destination mac address to send the packet to its destination as we are scanning a private IP range of my router. In Figure [21](#nocheckoutput) you can see the output of my program when run with the specified flags, you can see that as expected it showed that there were no open ports on those machines as they don’t exist. In Figure [22](#nocheckWireshark) you can see the packet capture of the packets my code sent, however, there are only ARP messages, this is because we are scanning in the private IP range of my router which was the only way I could guarantee that there was no machine at the other end. However, this is as expected, as well as this we can see 42 arp requests, which is arp requests, which would indicate each scan made three arp requests before giving up. This shows my program can perform scans without first checking if the host is up, showing success criterion [[nocheck]](#nocheck).



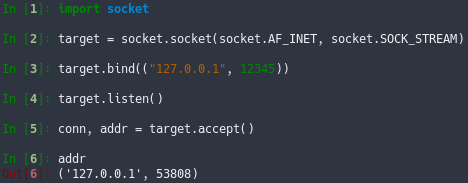
*Screenshot showing the output from my code when asked to port scan a subnet with no machines behind the addresses.*



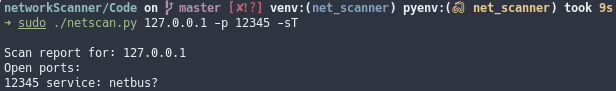
*Screenshot showing the ARP requests my NIC sent to attempt to determine where to send the attempted connection packets.*

### Detecting whether a TCP port is open

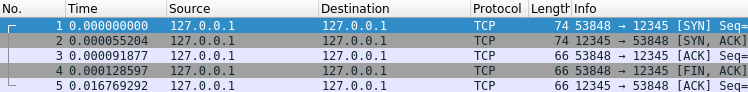
To show this I will perform a tcp Connect() scan on my local machine while running a script which will listen on port 12345 for any connections and send back a message. I will pass my program the flags -sT and -p 12345 as well as specifying localhost to scan (127.0.0.1). I expect to see a tcp SYN-ACK handshake between my program and the script and then my program to output that the port is open. In Figure [25](#tcpopenpcap) you can see the expected tcp SYN-ACK handshake performed by my program and the script in Figure [23](#tcpopenscript). You can see the output of my program in Figure [24](#tcpopenoutput); as expected it outputs that port 12345 is open. This shows success criteria [[blackbox]](#blackbox) and [[tcpopen]](#tcpopen).



*Screenshot showing the script I ran to accept a connection on localhost port 12345.*



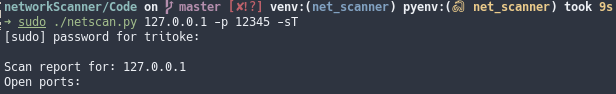
*Screenshot showing the output of my script when run with the specified flags and while the script in Figure*[*23*](#tcpopenscript) *was running.*



*Screenshot showing the packet capture of the TCP SYN-ACK handshake performed by the scan in Figure*[*24*](#tcpopenoutput) *with the script in*[*23*](#tcpopenscript)*.*

### Detecting whether a TCP port is closed

To show this, I will perform a tcp Connect() scan on my local machine, except that instead of running a script to catch the request, I will just let it try to connect to the closed port. I expect to see a tcp SYN packet sent to the port, and then a RST ACK packet sent back; my program should output no open ports. I will pass my program the same options as in the test for a tcp open port. In Figure [27](#tcpclosedpcap), you can see the attempted connection to 127.0.0.1 port 12345, along with the RST ACK packet afterwards, indicating the port is closed. This is reflected in Figure [26](#tcpclosedoutput) with no open ports, showing success criteria [[blackbox]](#blackbox) and [[tcpclosed]](#tcpclosed).



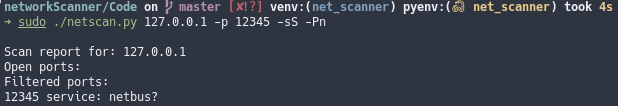
*Screenshot showing the output of my program when run with the specified options.*

 Screenshot showing the packet capture of the TCP SYN-RST closed port indication caused by the scan in Figure 26. 

*Screenshot showing the packet capture of the TCP SYN-RST closed port indication caused by the scan in Figure*[*26*](#tcpclosedoutput)*.*

### Detecting whether a TCP port is filtered

To show this I will perform a tcp SYN scan on localhost port 12345, except that I will also introduce a firewall rule to drop all requests to localhost. I expect this to produce no response to the initial SYN packet sent by my program, and my program to output that port as filtered. To test this, I will run my program with the flags -sS,-p 12345,-Pn. This will instruct it to perform a TCP SYN scan on port 12345 and not to check that the host is up before beginning the scan. I will also introduce a firewall rule using the Linux iptables utility to drop all requests to localhost as so: iptables -I INPUT -s 127.0.0.1 -j DROP. The output of my program is shown in Figure [28](#tcpfilteredoutput). It can be seen that port 12345 is displayed as filtered, and in the packet capture shown in Figure [29](#tcpfilteredpcap), you can see that there is no response to our initial packet, which corresponds to what I thought would happen with an iptables rule in place to drop packets. This shows success criteria [[blackbox]](#blackbox) and [[tcpfiltered]](#tcpfiltered).



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

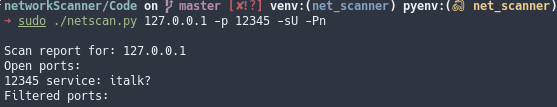
 Screenshot showing the packet capture of the scan in Figure 28 

*Screenshot showing the packet capture of the scan in Figure*[*28*](#tcpfilteredoutput)

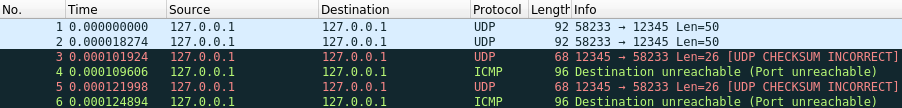
### Detecting whether a UDP port is open

To show this I will perform a udp scan on a script I had already written while developing udp scanning. This can be seen in Listing [[udpopenscript]](#udpopenscript). I expect to see my program output port 12345 as open, and in the packet capture, I expect to see two udp packets followed by two response udp packets from my listener program. I will test this using the following flags: -Pn,-p 12345,-sU. These translate to scanning port 12345 over UDP and not checking the host is up beforehand. In Figure [30](#udpopenoutput), you can see the output of my program when run as specified, and you can see that it correctly detects port 12345 as being open. In Figure [31](#udpopenpcap), you can see the packet capture of my program being run. However, the output was not precisely as expected, as I did not foresee the icmp destination unreachable messages, which were sent by the kernel in response to the UDP probe. However, apart from those messages, the capture shows the program detecting the udp port was open, as expected. This meets success criteria [[blackbox]](#blackbox) and [[udpopen]](#udpopen).

import socket  
from contextlib import closing  
  
with closing(  
 socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_DGRAM  
 )  
) as s:  
 s.bind(("127.0.0.1", 12345))  
 print("opened port 12345 on localhost")  
 while True:  
 data, addr = s.recvfrom(1024)  
 s.sendto(bytes("Well hello there good sir.", "utf-8"), addr)



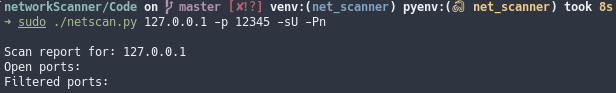
*Screenshot showing the output of my program when run with the options specified above, and the script in Listing*[*[udpopenscript]*](#udpopenscript) *is running.*



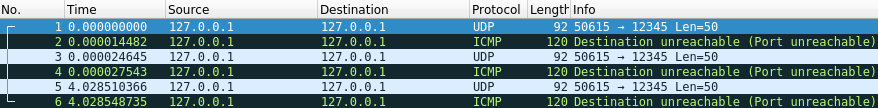
*screenshot showing the packet capture of the scan in Figure*[*30*](#udpopenoutput)

### Detecting whether a UDP port is closed

To show this I will perform a udp scan on a port which has no service listening behind it. I expect my program to print out no filtered ports and no open ports, showing that the port was closed. In the packet capture, I expect to see three udp packets and three response icmp packets. To test this, I will use my program with the following flags: -p 12345,-Pn,-sU which perform a udp port scan without first checking if the host is up. In Figure [32](#udpclosedoutput), you can see the output of my program when run with the options specified above. There are no ports displayed as either open or filtered. This shows that my program successfully identified the port as closed. This shows success criteria [[blackbox]](#blackbox) and [[udpclosed]](#udpclosed).



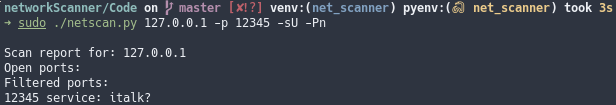
*screenshot showing the output of my program when scanning with the options specified above.*



*screenshot showing the packet capture of the scan in Figure*[*32*](#udpclosedoutput)

### Detecting whether a UDP port is filtered

To show this I will use my program to perform a udp scan on my local machine with a firewall rule to drop any ports sent to the localhost address. I expect to see my program to output the port as filtered and in the packet capture I expect to see three udp packets with no response to any of them. In Figure [34](#udpfilteredoutput) you can see my program correctly identifies the port as being filtered, and in Figure [35](#udpfilteredpcap) you can see the packet capture of the scan which also as expected shows the three udp packets with no reply packets. This shows the program meeting success criteria [[blackbox]](#blackbox) and [[udpfiltered]](#udpfiltered).



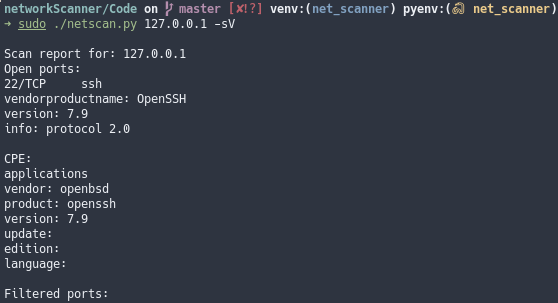
*screenshot showing the output of my program when scanning with the options specified above.*

 screenshot showing the packet capture of the scan in Figure 34 

*screenshot showing the packet capture of the scan in Figure*[*34*](#udpfilteredoutput)

### Detecting the operating system of another machine

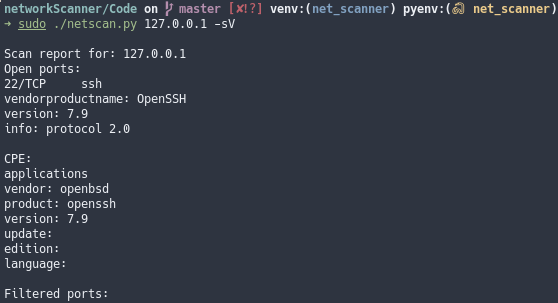
I haven’t directly added this as a feature to my project, largely because I discovered on implementing version scanning that this had the substantially the same effect. I found that in most instances it was possible to detect an operating system-dependent service and thereby deduce which operating system that machine was running. For example, if a machine is open on tcp port 22, and ssh is detected to be running behind that port, then the machine is most likely running Linux, as rdp is more commonly used for this purpose on Windows machines. It would become even more likely that the target machine runs Linux if the scan reveals some further information such as the cpe. In Figure [36](#sshversiondetect) you can see a scan of my machine where I have ssh running. My program reveals that the version is 7.9 and the vendor is openbsd, which is a Unix-like operating system. This shows that my ssh version is Unix-based and therefore that my machine is likely running Linux, which is the case. This partially completes success criterion [[osdetect]](#osdetect).



*screenshot showing a version scan of my local machine.*

### Detecting the service and its version running behind a port

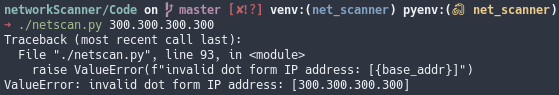
To show this I will use my program to perform a version detection scan on my local machine while I am running ssh. I expect to see my program identify that ssh is running on tcp port 22 and that it detects it as OpenSSH version 7.9. To test this I will run my program with the -sV flag to indicate version detection and I will run it against the localhost address. In Figure [[versiondetect]](#versiondetect) you can see that my program successfully identified ssh as running on tcp port 22 as well as the expected identification of OpenSSH version 7.9 operating on protocol version 2. It also identified some CPE information such as OpenSSH coming from the openbsd distribution. This meets success criteria [[blackbox]](#blackbox), [[servicedetect]](#servicedetect) and [[versiondetect]](#versiondetect).



*screenshot showing a version scan of my local machine running ssh.*

### User enters invalid ip address

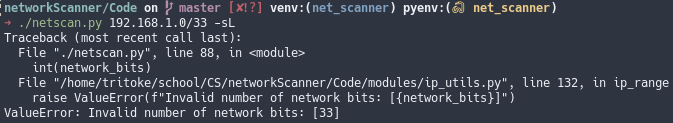
To show this I will run my program with the target\_spec option being  
300.300.300.300 which is an invalid IPv4 address, because each of the octets is not between 0 and 255. I expect to see my program raise a Python ValueError saying that this is an invalid dot form ipaddr, and displaying  
300.300.300.300 as the invalid ipaddr. In Figure [38](#invalidip) you can see my program’s output for this invalid IP address. This shows a successful pass as it correctly identifies the invalid IP and displays the error and the argument that caused the error to the user.



*Screenshot showing the output from an invalid IP address being used.*

### User enters invalid number of network bits

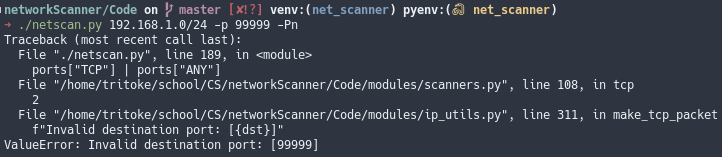
To show this I will run my program and ask it to list the ip addresses specified by the subnet 192.168.1.0/33. ip addresses are only 32 bits, long so specifying 33 network bits has no meaning, and thus is invalid data. I expect my program to raise a ValueError and print out that it was an invalid number of network bits that caused the error, along with 33 being the network bits. In Figure [39](#invalidnetworkbits) you can see that my program successfully identified the invalid number of network bits, raised the expected error, and printed the expected information.



*Screenshot showing the output of my program when passed an invalid number of network bits.*

### User enters an invalid port number to scan

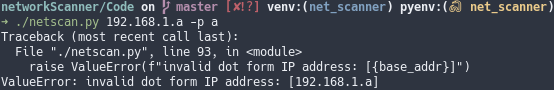
To show this I will run my program with the argument -p 99999. Because port numbers can only go up to 65535, this is erroneous data, and as such should generate an error message specifying that you have tried to scan an invalid destination port. In Figure [40](#invalidportnum) you can see that my program successfully identified 99999 as an invalid destination port and printed the correct error message accordingly.



*Screenshot of my program showing the output from an invalid port number.*

### User enters an invalid number of network bits and a bad IP address

To show this I will run my program with both an invalid ipaddr and an invalid port number. I expect this to raise a ValueError for the invalid ipaddr. This is because the ipaddrs are checked first, and thus an invalid ipaddr would be caught before an error could be raised for an invalid port number. To show this I will pass my program the following arguments:  
192.168.1.a -p a. In Figure [41](#invalidipandport) you can see that my program catches the invalid ipaddr and raises the correct ValueError as expected, thus it passed the test.



*screenshot of my program showing the output from an invalid ip and an invalid port number.*

## Test Table

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test No. | Test Data | Expectation | Result | Fig | Success Criteria |
|  |  | usage message | Pass | [13](#noparametertest) | [[usage]](#usage) |
|  | -h | help message | Pass | [14](#hflagtest) | [[usage]](#usage) |
|  | --help | help message | Pass | [15](#helpflagtest) | [[usage]](#usage) |
|  | -sL | print addresses | Pass | [19](#cidrtest) | [[cidr]](#cidr) |
|  | -sn | ping scan | Pass | [17](#lanscantest) | [[ping]](#ping) |
|  | -Pn | assume host up | Pass | [21](#nocheckoutput) | [[nocheck]](#nocheck) |
|  | -sS|sT | TCP port open | Pass | [24](#tcpopenoutput) | [[tcpopen]](#tcpopen) |
|  | -sS|sT | TCP port closed | Pass | [26](#tcpclosedoutput) | [[tcpclosed]](#tcpclosed) |
|  | -sS | TCP port filtered | Pass | [28](#tcpfilteredoutput) | [[tcpfiltered]](#tcpfiltered) |
|  | -sU | UDP port open | Pass | [30](#udpopenoutput) | [[udpopen]](#udpopen) |
|  | -sU | UDP port closed | Pass | [32](#udpclosedoutput) | [[udpclosed]](#udpclosed) |
|  | -sU | UDP port filtered | Pass | [34](#udpfilteredoutput) | [[udpfiltered]](#udpfiltered) |
|  | -sV | OS detection | Partial | [36](#sshversiondetect) | [[osdetect]](#osdetect) |
|  | -sV | service detection | Pass | [36](#sshversiondetect) | [[servicedetect]](#servicedetect) |
|  | -sV | version detection | Pass | [36](#sshversiondetect) | [[versiondetect]](#versiondetect) |
|  |  | invalid IP | Pass | [38](#invalidip) |  |
|  |  | invalid subnet | Pass | [39](#invalidnetworkbits) |  |
|  |  | invalid port & IP | Pass | [40](#invalidportnum) |  |

# Evaluation

## Reflection on final outcome

Overall, I am very happy with how my program has turned out. At the beginning of this project, I did not believe I could do anywhere near as much as I have done. My rationale for choosing this project was to learn more about low level networking, and networking in general, because these were areas where I felt my knowledge was deficient. I greatly enjoyed learning about various aspects of networking structure and protocols. For instance, I found it fascinating to discover that the order that bytes are packed into the packet matters enormously, and to learn how users are automatically assigned an IP address on joining a network.

There were many areas where I encountered difficulty while developing this program. The first problem was understanding what a packet is, and how I send one that I have made myself using Python. There seemed to be very little in the way of documentation, so I ended up reading many peoples’ answers to questions on the Stack Overflow website, where they had encountered similar problems. Once I discovered that I could simply open a socket in raw mode and use the socket.sendto method to send any arbitrary bytes to an address and Python would handle the IP header, it became easier to make progress. The next major problem was getting the checksum correct for tcp packets, because they use a pseudo-header, which is sort of a header made from fields in the ip header that are not in the tcp header. They are used to calculate the checksum for the tcp header. Figuring out how to pack these properly, and in what endianness, turned out to be one of the biggest difficulties in getting tcp SYN scanning working. I had a completely different problem when it came to implementing udp scanning. This is because when a udp packet arrives and is destined for a closed port, an icmp destination unreachable message is sent back, and on Linux systems these are time-limited to a maximum of one per second. Thus, I had to come up with a strategy that could deal with problem. In the end, I added a maximum wait time for each packet sent, and instructed the program to listen for that length of time to determine if an icmp destination unreachable message was received. Implementing version scanning came with its own difficulties. Parsing the file which defines the probes and all the directives which are needed to interpret the service’s response was challenging, as was matching the returned data using a regular expression that was parsed from a file.

## Evaluation against objectives, end user feedback

When I set out on this project I intended to make a program which was able to scan other computers on a network from a bbox perspective in a way similar to nmap. Before commencing, I set out 14 pre-specified criteria against which the performance of my program could be measured. As has been demonstrated above, the program fully meets 13 of the 14 criteria, and partially meets the remaining criterion. I believe therefore that my program has substantially met all the objectives set out at the beginning.

## Potential improvements

I believe that I could improve my program if it were to have a dedicated option for operating system detection. It is clear that as a single individual, it would not be feasible for me to collect the required number of operating system signatures myself. Therefore, in order to implement this feature, I would need use another of the files from nmap: nmao-os-db, which contains a database of tcp/ip stack fingerprints that show how the tcp and ip protocols are implemented in virtually all currently-used operating systems, as well as in different versions of Linux and different service packs and versions of Windows.

# Technical Solution

## icmp\_ping

#!/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  
  
 # the two zeros are the code and the dummy checksum, the one is the  
 # sequence number  
 dummy\_header = struct.pack("bbHHh", ICMP\_ECHO\_REQUEST, 0, 0, ID, 1)  
  
 data = struct.pack(  
 "d", time.time()  
 ) + bytes(  
 (192 - struct.calcsize("d")) \* "A",  
 "ascii"  
 )  
 # the data to send in the packet  
 checksum = socket.htons(ip\_checksum(dummy\_header + data))  
 # calculates the checksum for the packet and psuedo header  
 header = struct.pack("bbHHh", ICMP\_ECHO\_REQUEST, 0, checksum, ID, 1)  
 # packs the packet header  
 packet = header + data  
 # concatonates the header and the data to form the final packet.  
 ping\_sock.sendto(packet, ("127.0.0.1", 1))  
 # sends the packet to localhost

#!/usr/bin/env python  
from modules import headers  
import socket  
from typing import List  
  
  
def main() -> None:  
 # socket object using an IPV4 address, using only raw socket access, set  
 # ICMP protocol  
 ping\_sock = socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_RAW,  
 socket.IPPROTO\_ICMP  
 )  
  
 packets: List[bytes] = []  
  
 while len(packets) < 1:  
 recPacket, addr = ping\_sock.recvfrom(1024)  
 ip = headers.ip(recPacket[:20])  
 icmp = headers.icmp(recPacket[20:28])  
  
 print(ip)  
 print()  
 print(icmp)  
 print("\n")  
  
 packets.append(recPacket)

## ping\_scanner

#!/usr/bin/env python  
from modules import headers  
from modules import ip\_utils  
import socket  
import struct  
import time  
from contextlib import closing  
from itertools import repeat  
from math import log10, floor  
from multiprocessing import Pool  
from os import getpid  
from typing import Set, Tuple  
  
  
def sig\_figs(x: float, n: int) -> float:  
 """  
 rounds x to n significant figures.  
 sig\_figs(1234, 2) = 1200.0  
 """  
 return round(x, n - (1 + int(floor(log10(abs(x))))))  
  
  
def ping\_listener(  
 ID: int,  
 timeout: float  
) -> Set[Tuple[str, float, headers.ip]]:  
 """  
 Takes in a process id and a timeout and returns  
 a 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  
 )  
 # opens a raw socket for sending ICMP protocol packets  
 time\_remaining = timeout  
 addresses = set()  
 while True:  
 time\_waiting = ip\_utils.wait\_for\_socket(ping\_sock, time\_remaining)  
 # time\_waiting stores the time the socket took to become readable  
 # or returns minus one if it ran out of time  
  
 if time\_waiting == -1:  
 break  
 time\_recieved = time.time()  
 # store the time the packet was recieved  
 recPacket, addr = ping\_sock.recvfrom(1024)  
 # recieve the packet  
 ip = headers.ip(recPacket[:20])  
 # unpack the IP header into its respective components  
 icmp = headers.icmp(recPacket[20:28])  
 # unpack the time from the packet.  
 time\_sent = struct.unpack(  
 "d",  
 recPacket[28:28 + struct.calcsize("d")]  
 )[0]  
 # unpack the value for when the packet was sent  
 time\_taken: float = time\_recieved - time\_sent  
 # calculate the round trip time taken for the packet  
 if icmp.id == ID:  
 # if the ping was sent from this machine then add it to the list of  
 # responses  
 ip\_address, port = addr  
 addresses.add((ip\_address, time\_taken, ip))  
 elif time\_remaining <= 0:  
 break  
 else:  
 continue  
 # return a list of all the addesses that replied to our ICMP echo request.  
 return addresses  
  
  
def main() -> None:  
 with closing(  
 socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_RAW,  
 socket.IPPROTO\_ICMP  
 )  
 ) as ping\_sock:  
 ip\_addresses = ["127.0.0.1"] # ip\_utils.ip\_range("192.168.43.0", 24)  
 # generate the range of IP addresses to scan.  
 # get the local ip address  
 addresses = [  
 ip  
 for ip in ip\_addresses  
 if (  
 not ip.endswith(".0")  
 and not ip.endswith(".255")  
 )  
 ]  
  
 # initialise a process pool  
 p = Pool(1)  
 # get the local process id for use in creating packets.  
 ID = getpid() & 0xFFFF  
 # run the listeners.ping function asynchronously  
 replied = p.apply\_async(ping\_listener, (ID, 5))  
 time.sleep(0.01)  
 for address in zip(addresses, repeat(1)):  
 try:  
 packet = ip\_utils.make\_icmp\_packet(ID)  
 ping\_sock.sendto(packet, address)  
 except PermissionError:  
 ip\_utils.eprint("raw sockets require root priveleges, exiting")  
 exit()  
 p.close()  
 p.join()  
 # close and join the process pool to so that all the values  
 # have been returned and the pool closed  
 hosts\_up = replied.get()  
 # get the list of addresses that replied to the echo request from the  
 # listener function  
 print("\n".join(  
 f"host: [{host}]\t" +  
 "responded to an ICMP ECHO REQUEST in " +  
 f"{str(sig\_figs(taken, 2))+'s':<10s} " +  
 f"ttl: [{ip\_head.time\_to\_live}]"  
 for host, taken, ip\_head in hosts\_up  
 ))

## subnet\_to\_addresses

#!/usr/bin/env python  
import re  
from modules.ip\_utils import ip\_range, dot\_to\_long  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 from argparse import ArgumentParser  
 parser = ArgumentParser()  
 parser.add\_argument(  
 "ip\_subnet",  
 help="The CIDR form ip/subnet that you wish to print" +  
 "the IP addresses specified by."  
 )  
 args = parser.parse\_args()  
 CIDR\_regex = re.compile(r"(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}/\d+)")  
 search = CIDR\_regex.search(args.ip\_subnet)  
 if search:  
 ip, network\_bits = search.group(1).split("/")  
 print("\n".join(  
 sorted(  
 ip\_range(ip, int(network\_bits)),  
 key=dot\_to\_long  
 )  
 ))

## tcp\_scan

### connect\_scan

#!/usr/bin/python3  
from contextlib import closing  
import socket  
LOCAL\_IP = "192.168.1.159"  
PORT = 22  
  
address = ("127.0.0.1", 22)  
  
with closing(  
 socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_STREAM  
 )  
) as s:  
 try:  
 s.connect(address)  
 print(f"connection on port {PORT} succedded")  
 except ConnectionRefusedError:  
 print(f"port {PORT} is closed")

#!/usr/bin/python3  
  
from typing import List, Set  
  
  
def connect\_scan(address: str, ports: Set[int]) -> List[int]:  
 import socket  
 from contextlib import closing  
 open\_ports: List[int] = []  
 for port in ports:  
 # loop through each port in the list of ports to scan  
 try:  
 with closing(  
 socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_STREAM  
 )  
 ) as s:  
 # open an IPV4 TCP socket  
 s.connect((address, port))  
 # attempt to connect the newly created socket to the target  
 # address and port  
 open\_ports.append(port)  
 # if the connection was successful then add the port to the  
 # list of open ports  
 except ConnectionRefusedError:  
 pass  
 return open\_ports  
  
  
def main() -> None:  
 open\_ports = connect\_scan("192.168.43.225", set(range(65535)))  
 print("\n".join(map(lambda x: f"port: [{x}]\tis open", open\_ports)))

### syn\_scan

#!/usr/bin/python3.7  
from contextlib import closing  
import socket  
import ip\_utils  
  
dest\_port = 22  
src\_port = ip\_utils.get\_free\_port()  
local\_ip = ip\_utils.get\_local\_ip()  
dest\_ip = "192.168.1.159"  
local\_ip = dest\_ip = "127.0.0.1"  
loc\_long = ip\_utils.dot\_to\_long(local\_ip)  
  
SYN = 2  
RST = 4  
  
  
  
with closing(  
 socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_RAW,  
 socket.IPPROTO\_TCP  
 )  
) as s:  
 tcp\_packet = ip\_utils.make\_tcp\_packet(  
 src\_port,  
 dest\_port,  
 local\_ip,  
 dest\_ip,  
 SYN  
 )  
 if tcp\_packet is not None:  
 s.sendto(tcp\_packet, (dest\_ip, dest\_port))  
 else:  
 print(f"Couldn't make TCP packet with supplied arguments:",  
 f"source port: [{src\_port}]",  
 f"destination port: [{dest\_port}]",  
 f"local ip: [{local\_ip}]",  
 f"destination ip: [{dest\_ip}]",  
 f"SYN flag: [{SYN}]",  
 sep="\n")

#!/usr/bin/python3.7  
from modules import headers  
from modules import ip\_utils  
import socket  
from contextlib import closing  
from multiprocessing import Pool  
from typing import List, Set, Tuple  
  
  
def syn\_listener(address: Tuple[str, int], timeout: float) -> List[int]:  
 """  
 This function is run asynchronously and listens for  
 TCP ACK responses to the sent TCP SYN msg.  
 """  
 print(f"address: [{address}]\ntimeout: [{timeout}]")  
 open\_ports: List[int] = []  
 with closing(  
 socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_RAW,  
 socket.IPPROTO\_TCP  
 )) as s:  
 s.bind(address)  
 # bind the raw socket to the listening address  
 time\_remaining = timeout  
 print("started listening")  
 while True:  
 time\_taken = ip\_utils.wait\_for\_socket(s, time\_remaining)  
 # wait for the socket to become readable  
 if time\_taken == -1:  
 break  
 else:  
 time\_remaining -= time\_taken  
 packet = s.recv(1024)  
 # recieve the packet data  
 tcp = headers.tcp(packet[20:40])  
 if tcp.flags == 0b00010010: # syn ack  
 print(tcp)  
 open\_ports.append(tcp.source)  
 # check that the header contained the TCP ACK flag and if it  
 # did append it  
 else:  
 continue  
 print("finished listening")  
 return open\_ports  
  
  
def syn\_scan(dest\_ip: str, portlist: Set[int]) -> List[int]:  
 src\_port = ip\_utils.get\_free\_port()  
 # request a local port to connect from  
 local\_ip = ip\_utils.get\_local\_ip()  
 p = Pool(1)  
 listener = p.apply\_async(syn\_listener, ((local\_ip, src\_port), 5))  
 # start the TCP ACK listener in the background  
 print("starting scan")  
 for port in portlist:  
 packet = ip\_utils.make\_tcp\_packet(src\_port, port, local\_ip, dest\_ip, 2)  
 # create a TCP packet with the syn flag  
 with closing(  
 socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_RAW,  
 socket.IPPROTO\_TCP  
 )  
 ) as s:  
 s.sendto(packet, (dest\_ip, port))  
 # send the packet to its destination  
  
 print("finished scan")  
 p.close()  
 p.join()  
 open\_ports = listener.get()  
 # collect the list of ports that responded to the TCP SYN message  
 print(open\_ports)  
 return open\_ports  
  
  
def main() -> None:  
 dest\_ip = "127.0.0.1"  
 syn\_scan(dest\_ip, set(range(2\*\*16)))

## udp\_scan

#!/usr/bin/ python  
from contextlib import closing  
import ip\_utils  
import socket  
  
dest\_ip = "192.168.1.1"  
dest\_port = 68  
local\_ip = ip\_utils.get\_local\_ip()  
local\_port = ip\_utils.get\_free\_port()  
  
local\_ip = dest\_ip = "127.0.0.1"  
  
address = (dest\_ip, dest\_port)  
  
with closing(  
 socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_RAW,  
 socket.IPPROTO\_UDP  
 )) as s:  
 try:  
 pkt = ip\_utils.make\_udp\_packet(  
 local\_port,  
 dest\_port,  
 local\_ip,  
 dest\_ip  
 )  
 if pkt is not None:  
 packet = bytes(pkt)  
 s.sendto(packet, address)  
 else:  
 print(  
 "Error making packet.",  
 f"local port: [{local\_port}]",  
 f"destination port: [{dest\_port}]",  
 f"local ip: [{local\_ip}]",  
 f"destination ip: [{dest\_ip}]",  
 sep="\n"  
 )  
 except socket.error:  
 raise

#!/usr/bin/env python  
from modules import headers  
from modules import ip\_utils  
import socket  
import time  
from collections import defaultdict  
from contextlib import closing  
from multiprocessing import Pool  
from typing import Set, DefaultDict  
  
  
def udp\_listener(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])  
 # unpack the UDP header  
 if dest\_ip == ip.source and ip.protocol == 17:  
 ports.add(udp.src)  
  
 return ports  
  
  
def icmp\_listener(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:  
 continue  
 ping\_sock.close()  
 return code  
  
  
def udp\_scan(  
 dest\_ip: str,  
 ports\_to\_scan: Set[int]  
) -> DefaultDict[str, Set[int]]:  
 """  
 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: DefaultDict[str, Set[int]] = defaultdict(set)  
 ports["REMAINING"] = ports\_to\_scan  
 p = Pool(1)  
 udp\_listen = p.apply\_async(udp\_listener, (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"]:  
 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(icmp\_listener, (dest\_ip,))  
 # start the ICMP listener  
 time.sleep(1)  
 s.sendto(packet, (dest\_ip, dest\_port))  
 # send packet  
 p.close()  
 p.join()  
 icmp\_code = icmp\_listen.get()  
 # recieve ICMP code from the ICMP listener  
 if icmp\_code in {0, 1, 2, 9, 10, 13}:  
 ports["FILTERED"].add(dest\_port)  
 elif icmp\_code == 3:  
 ports["CLOSED"].add(dest\_port)  
 except socket.error:  
 packet\_bytes = " ".join(map("{:02x}".format, packet))  
 ip\_utils.eprint(  
 "The socket modules sendto method with the following",  
 "argument resulting in a socket error.",  
 f"\npacket: [{packet\_bytes}]\n",  
 "address: [{dest\_ip, dest\_port}])"  
 )  
 # this creates a new set which contains all the elements that  
 # are in the list of ports to be scanned but have not yet  
 # been classified  
 ports["OPEN|FILTERED"] = (  
 ports["REMAINING"]  
 - ports["OPEN"]  
 - ports["FILTERED"]  
 - ports["CLOSED"]  
 )  
 # set comprehension to update the list of open filtered ports  
 return ports  
  
  
def main() -> None:  
 ports = udp\_scan("127.0.0.1", {22, 68, 53, 6969})  
 print(f"Open ports: {ports['OPEN']}")  
 print(f"Open or filtered ports: {ports['OPEN|FILTERED']}")  
 print(f"Filtered ports: {ports['FILTERED']}")  
 print(f"Closed ports: {ports['CLOSED']}")

#!/usr/bin/env python  
  
import socket  
from contextlib import closing  
  
with closing(  
 socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_DGRAM  
 )  
) as s:  
 s.bind(("127.0.0.1", 12345))  
 print("opened port 12345 on localhost")  
 while True:  
 data, addr = s.recvfrom(1024)  
 s.sendto(bytes("Well hello there good sir.", "utf-8"), addr)

## version\_detection

#!/usr/bin/env python  
from typing import Dict, Set, Pattern, Tuple, DefaultDict  
from functools import reduce  
from collections import defaultdict  
from modules import directives  
import re  
import operator  
  
# type annotaion for the container which  
# holds the probes. I have abstracted it from  
# the function definition because multiple functions  
# depend on it and they weren't all getting updated  
# if I needed to change the function signature.  
PROBE\_CONTAINER = DefaultDict[str, Dict[str, directives.Probe]]  
  
  
def parse\_ports(portstring: str) -> DefaultDict[str, Set[int]]:  
 """  
 This function takes in a port directive  
 and returns a set of the ports specified.  
 A set is used because it is O(1) for contains  
 operations as opposed for O(N) for lists.  
 """  
 # matches both the num-num port range format  
 # and the plain num port specification  
 # num-num form must come first otherwise it breaks.  
 proto\_regex = re.compile(r"([ TU]):?([0-9,-]+)")  
 # THE SPACE IS IMPORTANT!!!  
 # it allows ports specified before TCP/UDP ports  
 # to be specified globally as in for all protocols.  
  
 pair\_regex = re.compile(r"(\d+)-(\d+)")  
 single\_regex = re.compile(r"(\d+)")  
 ports: DefaultDict[str, Set[int]] = defaultdict(set)  
 # searches contains the result of trying the pair\_regex  
 # search against all of the command seperated  
 # port strings  
  
 for protocol, portstring in proto\_regex.findall(portstring):  
 pairs = pair\_regex.findall(portstring)  
 # for each pair of numbers in the pairs list  
 # seperate each number and cast them to int  
 # then generate the range of numbers from x[0]  
 # to x[1]+1 then cast this range to a list  
 # and "reduce" the list of lists by joining them  
 # with operator.ior (inclusive or) and then let  
 # ports be the set of all the ports in that list.  
 proto\_map = {  
 " ": "ANY",  
 "U": "UDP",  
 "T": "TCP"  
 }  
 if pairs:  
 def pair\_to\_ports(pair: Tuple[int, int]) -> Set[int]:  
 """  
 a function to go from a port pair i.e. (80-85)  
 to the set of specified ports: {80,81,82,83,84,85}  
 """  
 start, end = pair  
 return set(range(start, end+1))  
 # ports contains the set of all ANY/TCP/UDP specified ports  
 ports[proto\_map[protocol]] = set(reduce(  
 operator.ior,  
 map(pair\_to\_ports, pairs)  
 ))  
  
 singles = single\_regex.findall(portstring)  
 # for each of the ports that are specified on their own  
 # cast them to int and update the set of all ports with  
 # that list.  
 ports[proto\_map[protocol]].update(map(int, singles))  
  
 return ports  
  
  
def parse\_probes(probe\_file: str) -> PROBE\_CONTAINER:  
 """  
 Extracts all of the probe directives from the  
 file pointed to by probe\_file.  
 """  
 # lines contains each line of the file which doesn't  
 # start with a # and is not empty.  
 lines = [  
 line  
 for line in open(probe\_file).read().splitlines()  
 if line and not line.startswith("#")  
 ]  
  
 # list holding each of the probe directives.  
 probes: PROBE\_CONTAINER = defaultdict(dict)  
  
 regexes: Dict[str, Pattern] = {  
 "probe": re.compile(r"Probe (TCP|UDP) (\S+) q\|(.\*)\|"),  
 "match": re.compile(" ".join([  
 r"(?P<type>softmatch|match)",  
 r"(?P<service>\S+)",  
 r"m([@/%=|])(?P<regex>.+?)\3(?P<flags>[si]\*)"  
 ])),  
 "rarity": re.compile(r"rarity (\d+)"),  
 "totalwaitms": re.compile(r"totalwaitms (\d+)"),  
 "tcpwrappedms": re.compile(r"tcpwrappedms (\d+)"),  
 "fallback": re.compile(r"fallback (\S+)"),  
 "ports": re.compile(r"ports (\S+)"),  
 "exclude": re.compile(r"Exclude T:(\S+)")  
 }  
  
 # parse the probes out from the file  
 for line in lines:  
 # add any ports to be excluded to the base probe class  
 if line.startswith("Exclude"):  
 search = regexes["exclude"].search(line)  
 if search:  
 # parse the ports from the grouped output of  
 # a search with the regex defined above.  
 for protocol, ports in parse\_ports(search.group(1)).items():  
 directives.Probe.exclude[protocol].update(ports)  
 else:  
 print(line)  
 input()  
  
 # new probe directive  
 if line.startswith("Probe"):  
 # parse line into probe protocol, name and probestring  
 search = regexes["probe"].search(line)  
 if search:  
 try:  
 proto, name, string = search.groups()  
 except ValueError:  
 print(line)  
 raise  
 probes[name][proto] = directives.Probe(proto, name, string)  
 # assign current\_probe to the most recently added probe  
 current\_probe = probes[name][proto]  
 else:  
 print(line)  
 input()  
  
 # new match directive  
 elif line.startswith("match") or line.startswith("softmatch"):  
 search = regexes["match"].search(line)  
 if search:  
 # the remainder of the string after the match  
 version\_info = line[search.end()+1:]  
 # escape the curly braces so the regex engine doesn't  
 # consider them to be special characters  
 pattern = bytes(search.group("regex"), "utf-8")  
 # these replace the literal \n, \r and \t  
 # strings with their actual characters  
 # i.e. \n -> newline character  
 pattern = pattern.replace(b"\\n", b"\n")  
 pattern = pattern.replace(b"\\r", b"\r")  
 pattern = pattern.replace(b"\\t", b"\t")  
 matcher = directives.Match(  
 search.group("service"),  
 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  
  
  
def version\_detect\_scan(  
 target: directives.Target,  
 probes: PROBE\_CONTAINER  
) -> directives.Target:  
 for probe\_dict in probes.values():  
 for proto in probe\_dict:  
 target = probe\_dict[proto].scan(target)  
 return target  
  
  
def main() -> None:  
 print("reached here")  
 probes = parse\_probes("./version\_detection/nmap-service-probes")  
 open\_ports: DefaultDict[str, Set[int]] = defaultdict(set)  
 open\_filtered\_ports: DefaultDict[str, Set[int]] = defaultdict(set)  
 open\_filtered\_ports["TCP"].add(22)  
 open\_ports["TCP"].update([1, 2, 3, 4, 5, 6, 8, 65,  
 20, 21, 23, 24, 25])  
  
 target = directives.Target(  
 "127.0.0.1",  
 open\_ports,  
 open\_filtered\_ports  
 )  
 target.open\_ports["TCP"].update([1, 2, 3])  
 print("BEFORE")  
 print(target)  
 scanned = version\_detect\_scan(target, probes)  
 print("AFTER")  
 print(scanned)

## modules

#!/usr/bin/env python  
from collections import defaultdict  
from contextlib import closing  
from dataclasses import dataclass, field  
from functools import reduce  
from string import whitespace, printable  
from typing import (  
 DefaultDict,  
 Dict,  
 Set,  
 List,  
 Pattern,  
 Match as RE\_Match,  
 Tuple  
)  
from . import ip\_utils  
import operator  
import re  
import socket  
import struct  
  
  
class Match:  
 """  
 This is a class for both Matches and  
 Softmatches as they are actually the same  
 thing except that softmatches have less information.  
 """  
 options\_to\_flags = {  
 "i": re.IGNORECASE,  
 "s": re.DOTALL  
 }  
 letter\_to\_name = {  
 "p": "vendorproductname",  
 "v": "version",  
 "i": "info",  
 "h": "hostname",  
 "o": "operatingsystem",  
 "d": "devicetype"  
 }  
 cpe\_part\_map: Dict[str, str] = {  
 "a": "applications",  
 "h": "hardware platforms",  
 "o": "operating systems"  
 }  
 # look into match.expand when looking at the substring version info things.  
  
 def \_\_init\_\_(  
 self,  
 service: str,  
 pattern: bytes,  
 pattern\_options: str,  
 version\_info: str  
 ):  
 self.version\_info: Dict[str, str] = dict()  
 self.cpes: Dict[str, Dict[str, str]] = dict()  
 self.service: str = service  
 # bitwise or is used to combine flags  
 # pattern options will never be anything but a  
 # combination of s and i.  
 # the default value of re.V1 is so that  
 # re uses the newer matching engine.  
 flags = reduce(  
 operator.ior,  
 [  
 self.options\_to\_flags[opt]  
 for opt in pattern\_options  
 ],  
 0  
 )  
 try:  
 self.pattern: Pattern = re.compile(  
 pattern,  
 flags=flags  
 )  
 except Exception as e:  
 print("Regex failed to compile:")  
 print(e)  
 print(pattern)  
 input()  
  
 vinfo\_regex = re.compile(r"([pvihod]|cpe:)([/|])(.+?)\2([a]\*)")  
 cpe\_regex = re.compile(  
 ":?".join((  
 "(?P<part>[aho])",  
 "(?P<vendor>[^:]\*)",  
 "(?P<product>[^:]\*)",  
 "(?P<version>[^:]\*)",  
 "(?P<update>[^:]\*)",  
 "(?P<edition>[^:]\*)",  
 "(?P<language>[^:]\*)"  
 ))  
 )  
  
 for fieldname, \_, val, opts in vinfo\_regex.findall(version\_info):  
 if fieldname == "cpe:":  
 search = cpe\_regex.search(val)  
 if search:  
 part = search.group("part")  
 # this next bit is so that the bytes produced by the regex  
 # are turned to strings  
 self.cpes[Match.cpe\_part\_map[part]] = {  
 key: value  
 for key, value  
 in search.groupdict().items()  
 }  
 else:  
 self.version\_info[  
 Match.letter\_to\_name[fieldname]  
 ] = val  
  
 def \_\_repr\_\_(self) -> str:  
 return "Match(" + ", ".join((  
 f"service={self.service}",  
 f"pattern={self.pattern}",  
 f"version\_info={self.version\_info}",  
 f"cpes={self.cpes}"  
 )) + ")"  
  
 def matches(self, string: bytes) -> bool:  
 def replace\_groups(  
 string: str,  
 original\_match: RE\_Match  
 ) -> str:  
 """  
 This function takes in a string and the original  
 regex search performed on the data recieved and  
 replaces all of the $i, $SUBST, $I, $P occurances  
 with the relavant formatted text that they produce.  
 """  
 def remove\_unprintable(  
 group: int,  
 original\_match: RE\_Match  
 ) -> bytes:  
 """  
 Mirrors the P function from nmap which  
 is used to print only printable characters.  
 i.e. W\0O\0R\0K\0G\0R\0O\0U\0P -> WORKGROUP  
 """  
 return b"".join(  
 i for i in original\_match.group(group)  
 if ord(i) in (  
 set(printable)  
 - set(whitespace)  
 | {" "}  
 )  
 )  
 # if i in the set of all printable characters,  
 # excluding those of which that are whitespace characters  
 # but including space.  
  
 def substitute(  
 group: int,  
 before: bytes,  
 after: bytes,  
 original\_match: RE\_Match  
 ) -> 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):  
 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:  
 """  
 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]}" + "}"  
 store\_results.append(key\_result)  
 # format the final result  
 result = "{" + ", ".join(store\_results) + "}"  
 return result  
  
 open\_ports = collapse(self.open\_ports)  
 open\_filtered\_ports = collapse(self.open\_filtered\_ports)  
 return ", ".join((  
 f"Target(address=[{self.address}]",  
 f"open\_ports=[{open\_ports}]",  
 f"open\_filtered\_ports=[{open\_filtered\_ports}]",  
 f"services={self.services})"  
 ))  
  
  
class Probe:  
 """  
 This class represents the Probe directive of the nmap-service-probes file.  
 It holds information such as the protocol to use, the string to send,  
 the ports to scan, the time to wait for a null TCP to return a banner,  
 the rarity of the probe (how often it will return a response) and the  
 probes to try if this one fails.  
 """  
  
 # a default dict is one which takes in a  
 # "default factory" which is called when  
 # a new key is introduced to the dict  
 # in this case the default factory is  
 # the set function meaning that when I  
 # do exclude[protocol].update(ports)  
 # but exclude[protocol] has not yet been defined  
 # it will be defined as an empty set  
 # allowing me to update it with ports.  
 exclude: DefaultDict[str, Set[int]] = defaultdict(set)  
 proto\_to\_socket\_type: Dict[str, int] = {  
 "TCP": socket.SOCK\_STREAM,  
 "UDP": socket.SOCK\_DGRAM  
 }  
  
 def \_\_init\_\_(self, protocol: str, probename: str, probe: str):  
 """  
 This is the initial function that is called by the  
 constructor of the Probe class, it is used to define  
 the variables that are specific to each instance of  
 the class.  
 """  
 if protocol in {"TCP", "UDP"}:  
 self.protocol = protocol  
 else:  
 raise ValueError(  
 f"Probe object must have protocol TCP or UDP not {protocol}.")  
 self.name: str = probename  
 self.string: str = probe  
 self.payload: bytes = bytes(probe, "utf-8")  
 self.matches: Set[Match] = set()  
 self.softmatches: Set[Match] = set()  
 self.ports: DefaultDict[str, Set[int]] = defaultdict(set)  
 self.totalwaitms: int = 6000  
 self.tcpwrappedms: int = 3000  
 self.rarity: int = -1  
 self.fallback: Set[str] = set()  
  
 def \_\_repr\_\_(self) -> str:  
 """  
 This is the function that is called when something  
 tries to print an instance of this class.  
 It is used to reveal information internal  
 to the class.  
 """  
 return ", ".join([  
 f"Probe({self.protocol}",  
 f"{self.name}",  
 f"\"{self.string}\"",  
 f"{len(self.matches)} matches",  
 f"{len(self.softmatches)} softmatches",  
 f"ports: {self.ports}",  
 f"rarity: {self.rarity}",  
 f"fallbacks: {self.fallback})"  
 ])  
  
 def scan(self, target: Target) -> Target:  
 """  
 scan takes in an object of class Target to  
 probe and attempts to detect the version of  
 any services running on the machine.  
 """  
 # this constructs the set of all ports,  
 # that are either open or open\_filtered,  
 # and are in the set of ports to scan for  
 # this particular probe, this means that,  
 # we are only connecting to ports that we  
 # know are not closed and are not to be excluded.  
  
 ports\_to\_scan: Set[int] = (  
 (  
 target.open\_filtered\_ports[self.protocol]  
 | target.open\_ports[self.protocol]  
 )  
 ) - Probe.exclude[self.protocol] - Probe.exclude["ANY"]  
 # if the probe defines a set of ports to scan  
 # then don't scan any that aren't defined for it  
 if self.ports[self.protocol] != set():  
 ports\_to\_scan &= self.ports[self.protocol]  
 for port in ports\_to\_scan:  
 # open a self closing IPV4 socket  
 # for the correct protocol for this probe.  
 with closing(  
 socket.socket(  
 socket.AF\_INET,  
 self.proto\_to\_socket\_type[self.protocol]  
 )  
 ) as sock:  
 # setup the connection to the target  
 try:  
 sock.connect((target.address, port))  
 # if the connection fails then continue scanning  
 # the next ports, this shouldn't really happen.  
 except ConnectionError:  
 continue  
 # send the payload to the target  
 sock.send(self.payload)  
 # wait for the target to send a response  
 time\_taken = ip\_utils.wait\_for\_socket(  
 sock,  
 self.totalwaitms/1000  
 )  
 # if the response didn't time out  
 if time\_taken != -1:  
 # if the port was in open\_filtered move it to open  
 if port in target.open\_filtered\_ports[self.protocol]:  
 target.open\_filtered\_ports[  
 self.protocol  
 ].remove(port)  
 target.open\_ports[self.protocol].add(port)  
  
 # recieve the data and decode it to a string  
 data\_recieved = sock.recv(4096)  
 # print("Recieved", data\_recieved)  
 service = ""  
 # try and softmatch the service first  
 for softmatch in self.softmatches:  
 if softmatch.matches(data\_recieved):  
 service = softmatch.service  
 target.services[port] = softmatch  
 break  
 # try and get a full match for the service  
 for match in self.matches:  
 if service in match.service.lower():  
 if match.matches(data\_recieved):  
 target.services[port] = match  
 break  
 return target  
  
  
PROBE\_CONTAINER = DefaultDict[str, Dict[str, Probe]]  
  
  
def parse\_ports(portstring: str) -> DefaultDict[str, Set[int]]:  
 """  
 This function takes in a port directive  
 and returns a set of the ports specified.  
 A set is used because it is O(1) for contains  
 operations as opposed for O(N) for lists.  
 """  
 # matches both the num-num port range format  
 # and the plain num port specification  
 # num-num form must come first otherwise it breaks.  
 proto\_regex = re.compile(r"([ TU]?):?([0-9,-]+)")  
 # THE SPACE IS IMPORTANT!!!  
 # it allows ports specified before TCP/UDP ports  
 # to be specified globally as in for all protocols.  
  
 pair\_regex = re.compile(r"(\d+)-(\d+)")  
 single\_regex = re.compile(r"(\d+)")  
 ports: DefaultDict[str, Set[int]] = defaultdict(set)  
 # searches contains the result of trying the pair\_regex  
 # search against all of the command seperated  
 # port strings  
  
 for protocol, portstring in proto\_regex.findall(portstring):  
 pairs = pair\_regex.findall(portstring)  
 # for each pair of numbers in the pairs list  
 # seperate each number and cast them to int  
 # then generate the range of numbers from x[0]  
 # to x[1]+1 then cast this range to a list  
 # and "reduce" the list of lists by joining them  
 # with operator.ior (inclusive or) and then let  
 # ports be the set of all the ports in that list.  
 proto\_map = {  
 "": "ANY",  
 " ": "ANY",  
 "U": "UDP",  
 "T": "TCP"  
 }  
 if pairs:  
 def pair\_to\_ports(pair: Tuple[str, str]) -> Set[int]:  
 """  
 a function to go from a port pair i.e. (80-85)  
 to the set of specified ports: {80,81,82,83,84,85}  
 """  
 start, end = pair  
 return set(range(  
 int(start),  
 int(end)+1  
 ))  
 # ports contains the set of all ANY/TCP/UDP specified ports  
 ports[proto\_map[protocol]] = set(reduce(  
 operator.ior,  
 map(pair\_to\_ports, pairs)  
 ))  
  
 singles = single\_regex.findall(portstring)  
 # for each of the ports that are specified on their own  
 # cast them to int and update the set of all ports with  
 # that list.  
 ports[proto\_map[protocol]].update(map(int, singles))  
  
 return ports  
  
  
def parse\_probes(probe\_file: str) -> PROBE\_CONTAINER:  
 """  
 Extracts all of the probe directives from the  
 file pointed to by probe\_file.  
 """  
 # lines contains each line of the file which doesn't  
 # start with a # and is not empty.  
 lines = [  
 line  
 for line in open(probe\_file).read().splitlines()  
 if line and not line.startswith("#")  
 ]  
  
 # list holding each of the probe directives.  
 probes: PROBE\_CONTAINER = defaultdict(dict)  
  
 regexes: Dict[str, Pattern] = {  
 "probe": re.compile(r"Probe (TCP|UDP) (\S+) q\|(.\*)\|"),  
 "match": re.compile(" ".join([  
 r"(?P<type>softmatch|match)",  
 r"(?P<service>\S+)",  
 r"m([@/%=|])(?P<regex>.+?)\3(?P<flags>[si]\*)"  
 ])),  
 "rarity": re.compile(r"rarity (\d+)"),  
 "totalwaitms": re.compile(r"totalwaitms (\d+)"),  
 "tcpwrappedms": re.compile(r"tcpwrappedms (\d+)"),  
 "fallback": re.compile(r"fallback (\S+)"),  
 "ports": re.compile(r"ports (\S+)"),  
 "exclude": re.compile(r"Exclude T:(\S+)")  
 }  
  
 # parse the probes out from the file  
 for line in lines:  
 # add any ports to be excluded to the base probe class  
 if line.startswith("Exclude"):  
 search = regexes["exclude"].search(line)  
 if search:  
 # parse the ports from the grouped output of  
 # a search with the regex defined above.  
 for protocol, ports in parse\_ports(search.group(1)).items():  
 Probe.exclude[protocol].update(ports)  
 else:  
 print(line)  
 input()  
  
 # new probe directive  
 if line.startswith("Probe"):  
 # parse line into probe protocol, name and probestring  
 search = regexes["probe"].search(line)  
 if search:  
 try:  
 proto, name, string = search.groups()  
 except ValueError:  
 print(line)  
 raise  
 probes[name][proto] = Probe(proto, name, string)  
 # assign current\_probe to the most recently added probe  
 current\_probe = probes[name][proto]  
 else:  
 print(line)  
 input()  
  
 # new match directive  
 elif line.startswith("match") or line.startswith("softmatch"):  
 search = regexes["match"].search(line)  
 if search:  
 # the remainder of the string after the match  
 version\_info = line[search.end()+1:]  
 # escape the curly braces so the regex engine doesn't  
 # consider them to be special characters  
 pattern = bytes(search.group("regex"), "utf-8")  
 # these replace the literal \n, \r and \t  
 # strings with their actual characters  
 # i.e. \n -> newline character  
 pattern = pattern.replace(b"\\n", b"\n")  
 pattern = pattern.replace(b"\\r", b"\r")  
 pattern = pattern.replace(b"\\t", b"\t")  
 matcher = Match(  
 search.group("service"),  
 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

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  
 self.header\_length: int = ip\_hl  
 self.dscp: int = ip\_dscp  
 self.ecn: int = ip\_ecn  
 self.len: int = ip\_len  
 self.id: int = ip\_id  
 self.flags: int = ip\_flgs  
 self.data\_offset: int = ip\_off  
 self.time\_to\_live: int = ip\_ttl  
 self.protocol: int = ip\_p  
 self.checksum: int = ip\_sum  
 self.source: str = src\_addr  
 self.destination: str = dst\_addr  
  
 def \_\_repr\_\_(self) -> str:  
 return "\n\t".join((  
 "IP header:",  
 f"Version: [{self.version}]",  
 f"Internet Header Length: [{self.header\_length}]",  
 f"Differentiated Services Point Code: [{self.dscp}]",  
 f"Explicit Congestion Notification: [{self.ecn}]",  
 f"Total Length: [{self.len}]",  
 f"Identification: [{self.id:04x}]",  
 f"Flags: [{self.flags:03b}]",  
 f"Fragment Offset: [{self.data\_offset}]",  
 f"Time To Live: [{self.time\_to\_live}]",  
 f"Protocol: [{self.protocol}]",  
 f"Header Checksum: [{self.checksum:04x}]",  
 f"Source Address: [{self.source}]",  
 f"Destination Address: [{self.destination}]"  
 ))  
  
  
class icmp:  
 """  
 A class for parsing, storing and displaying  
 data from an IP header.  
 """  
 # relates the type and code to the message  
 messages: Dict[int, Dict[int, str]] = {  
 0: {  
 0: "Echo reply."  
 },  
 3: {  
 0: "Destination network unreachable.",  
 1: "Destination host unreachable",  
 2: "Destination protocol unreachable",  
 3: "Destination port unreachable",  
 4: "Fragmentation required, and DF flag set.",  
 5: "Source route failed.",  
 6: "Destination network unknown.",  
 7: "Destination host unknown.",  
 8: "Source host isolated.",  
 9: "Network administratively prohibited.",  
 10: "Host administratively prohibited.",  
 11: "Network unreachable for ToS.",  
 12: "Host unreachable for ToS.",  
 13: "Communication administratively prohibited.",  
 14: "Host precedence violation.",  
 15: "Precedence cutoff in effect."  
 },  
 4: {  
 0: "Source quench."  
 },  
 5: {  
 0: "Redirect datagram for the network",  
 1: "Redirect datagram for the host.",  
 2: "Redirect datagram for the ToS & network.",  
 3: "Redirect datagram for the ToS & host."  
 },  
 8: {  
 0: "Echo request."  
 },  
 9: {  
 0: "Router advertisment"  
 },  
 10: {  
 0: "Router discovery/selection/solicitation."  
 },  
 11: {  
 0: "TTL expired in transit",  
 1: "Fragment reassembly time exceeded."  
 },  
 12: {  
 0: "Bad IP header: pointer indicates error.",  
 1: "Bad IP header: missing a required option.",  
 2: "Bad IP header: Bad length."  
 },  
 13: {  
 0: "Timestamp"  
 },  
 14: {  
 0: "Timestamp reply"  
 },  
 15: {  
 0: "Information request."  
 },  
 16: {  
 0: "Information reply."  
 },  
 17: {  
 0: "Address mask request."  
 },  
 18: {  
 0: "Address mask reply."  
 }  
 }  
  
 def \_\_init\_\_(self, header: bytes):  
 (  
 ICMP\_type,  
 code,  
 csum,  
 remainder  
 ) = struct.unpack('!bbHI', header)  
  
 self.type: int = ICMP\_type  
 self.code: int = code  
 self.checksum: int = csum  
  
 self.message: str  
 try:  
 self.message = icmp.messages[self.type][self.code]  
 except KeyError:  
 # if we can't assign a message then just set a description  
 # as to what caused the failure.  
 self.message = f"Failed to assign message: ({self.type/self.code})"  
  
 self.id: int  
 self.sequence: int  
 if self.type in {0, 8}:  
 self.id = socket.htons(remainder >> 16)  
 self.sequence = socket.htons(remainder & 0xFFFF)  
 else:  
 self.id = -1  
 self.sequence = -1  
  
 def \_\_repr\_\_(self) -> str:  
 return "\n\t".join((  
 "ICMP header:",  
 f"Message: [{self.message}]",  
 f"Type: [{self.type}]",  
 f"Code: [{self.code}]",  
 f"Checksum: [{self.checksum:04x}]",  
 f"ID: [{self.id}]",  
 f"Sequence: [{self.sequence}]"  
 ))  
  
  
class tcp:  
 def \_\_init\_\_(self, header: bytes):  
 (  
 src\_prt,  
 dst\_prt,  
 seq,  
 ack,  
 data\_offset,  
 flags,  
 window\_size,  
 checksum,  
 urg  
 ) = struct.unpack("!HHIIBBHHH", header)  
  
 self.source: int = src\_prt  
 self.destination: int = dst\_prt  
 self.seq: int = seq  
 self.ack: int = ack  
 self.data\_offset: int = data\_offset >> 4  
 self.flags: int = flags + ((data\_offset & 0x01) << 8)  
 self.window\_size: int = window\_size  
 self.checksum: int = checksum  
 self.urg: int = urg  
  
 def \_\_repr\_\_(self) -> str:  
 return "\n\t".join((  
 "TCP header:",  
 f"Source port: [{self.source}]",  
 f"Destination port: [{self.destination}]",  
 f"Sequence number: [{self.seq}]",  
 f"Acknowledgement number: [{self.ack}]",  
 f"Data offset: [{self.data\_offset}]",  
 f"Flags: [{self.flags:08b}]",  
 f"Window size: [{self.window\_size}]",  
 f"Checksum: [{self.checksum:04x}]",  
 f"Urgent: [{self.urg}]"  
 ))  
  
  
class udp:  
 def \_\_init\_\_(self, header: bytes):  
 # parse udp header  
 (  
 src\_port,  
 dest\_port,  
 length,  
 checksum  
 ) = struct.unpack("!HHHH", header)  
  
 self.src: int = src\_port  
 self.dest: int = dest\_port  
 self.length: int = length  
 self.checksum: int = checksum  
  
 def \_\_repr\_\_(self) -> str:  
 return "\n\t".join((  
 "UDP header:",  
 f"Source port: {self.src}",  
 f"Destination port: {self.dest}",  
 f"Length: {self.length}",  
 f"Checksum: {self.checksum:04x}"  
 ))

import array  
import socket  
import struct  
import select  
import time  
  
from contextlib import closing  
from functools import singledispatch  
from itertools import islice, cycle  
from sys import stderr  
from typing import Set, Union  
  
  
def eprint(\*args: str, \*\*kwargs: str) -> None:  
 """  
 Mirrors print exactly but prints to stderr  
 instead of stdout.  
 """  
 print(\*args, file=stderr, \*\*kwargs) # type: ignore  
  
  
def long\_to\_dot(long: int) -> str:  
 """  
 Take in an IP address in packed 32 bit int form  
 and return that address in dot notation.  
 i.e. long\_to\_dot(0x7F000001) = 127.0.0.1  
 """  
 # these are long form values for 0.0.0.0  
 # and 255.255.255.255  
 if not 0 <= long <= 0xFFFFFFFF:  
 raise ValueError(f"Invalid long form IP address: [{long:08x}]")  
 else:  
 # shift the long form IP along 0, 8, 16, 24 bits  
 # take only the first 8 bits of the newly shifted number  
 # 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}]")  
  
 if len(parts) != 4:  
 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  
 # the is\_valid\_ip function.  
 try:  
 # try to turn the long form ip address  
 # to a dot form one, if it fails,  
 # then return False, else return True  
 long\_to\_dot(ip)  
 return True  
 except ValueError:  
 return False  
  
  
# the type ignore comment is required to stop  
# mypy exploding over the fact I have defined `\_` twice.  
@is\_valid\_ip.register # type: ignore  
def \_(ip: str):  
 # this is the string overload variant  
 # of the is\_valid\_ip function.  
 try:  
 # try to turn the dot form ip address  
 # to a long form one, if it fails,  
 # then return False, else return True  
 dot\_to\_long(ip)  
 return True  
 except ValueError:  
 return False  
  
  
def is\_valid\_port\_number(port\_num: int) -> bool:  
 """  
 Checks whether the given port number is valid i.e. between 0 and 65536.  
 """  
 # port numbers must be between 0 and 65535(2^16 - 1)  
 if 0 <= port\_num < 2\*\*16:  
 return True  
 else:  
 return False  
  
  
def ip\_range(ip: str, network\_bits: int) -> Set[str]:  
 """  
 Takes a Classless Inter Domain Routing(CIDR) address subnet  
 specification and returns the list of addresses specified  
 by the IP/network bits format.  
 If the number of network bits is not between 0 and 32 it raises an error.  
 If the IP address is invalid according to is\_valid\_ip it raises an error.  
 """  
  
 if not 0 <= network\_bits <= 32:  
 raise ValueError(f"Invalid number of network bits: [{network\_bits}]")  
  
 if not is\_valid\_ip(ip):  
 raise ValueError(f"Invalid IP address: [{ip}]")  
 # get the ip as long form which is useful  
 # later on for using bitwise operators  
 # to isolate only the constant(network) bits  
 ip\_long = dot\_to\_long(ip)  
  
 # generate the bit mask which specifies  
 # which bits to keep and which to discard  
 mask = int(  
 f"{'1'\*network\_bits:0<32s}",  
 base=2  
 )  
 lower\_bound = ip\_long & mask  
 upper\_bound = ip\_long | (mask ^ 0xFFFFFFFF)  
  
 # turn all the long form IP addresses between  
 # the lower and upper bound into dot form  
 if network\_bits <= 30:  
 return set(  
 long\_to\_dot(long\_ip)  
 for long\_ip in  
 range(lower\_bound+1, upper\_bound)  
 )  
 else:  
 return set(  
 long\_to\_dot(long\_ip)  
 for long\_ip in  
 range(lower\_bound, upper\_bound+1)  
 )  
  
  
  
def get\_local\_ip() -> str:  
 """  
 Connects to the google.com with UDP and gets  
 the IP address used to connect(the local address).  
 """  
 with closing(  
 socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_DGRAM  
 )  
 ) as s:  
 try:  
 s.connect(("google.com", 80))  
 ip, \_ = s.getsockname()  
 except:  
 ip = "127.0.0.1"  
 return ip  
  
  
def get\_free\_port() -> int:  
 """  
 Attempts to bind to port 0 which assigns a free port number to the socket,  
 the socket is then closed and the port number assigned is returned.  
 """  
  
 with closing(  
 socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_STREAM  
 )  
 ) as s:  
 s.bind(('', 0))  
 \_, port = s.getsockname()  
 return port  
  
  
def ip\_checksum(packet: bytes) -> int:  
 """  
 ip\_checksum function takes in a packet  
 and returns the checksum.  
 """  
 if len(packet) % 2 == 1:  
 # if the length of the packet is odd, add a NULL byte  
 # to the end as padding  
 packet += b"\0"  
  
 total = 0  
 for first, second in (  
 packet[i:i+2]  
 for i in range(0, len(packet), 2)  
 ):  
 total += (first << 8) + second  
  
 # calculate the number of times a  
 # carry bit was added and add it back on  
 carried = (total - (total & 0xFFFF)) >> 16  
 total &= 0xFFFF  
 total += carried  
  
 if total > 0xFFFF:  
 # adding the carries generated a carry  
 total &= 0xFFFF  
 total += 1  
  
 # invert the checksum and take the last 16 bits.  
 return (~total & 0xFFFF)  
  
  
def make\_icmp\_packet(ID: int) -> bytes:  
 """  
 Takes an argument of the process ID of the calling process.  
 Returns an ICMP ECHO REQUEST packet created with this ID  
 """  
  
 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:  
 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  
 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  
) -> 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\_port\_number(src):  
 raise ValueError(  
 f"Invalid source port: [{src}]"  
 )  
 if not is\_valid\_port\_number(dst):  
 raise ValueError(  
 f"Invalid destination port: [{dst}]"  
 )  
 data = b"Most services don't respond to an empty data field"  
 # pack the data  
 # and return the packed bytes  
 # UDP checksum is optional over IPv4  
 return struct.pack(  
 "!HHHH",  
 src,  
 dst,  
 8+len(data),  
 0  
 ) + data  
  
  
def wait\_for\_socket(sock: socket.socket, wait\_time: float) -> float:  
 """  
 Wait for wait\_time seconds or until the socket is readable.  
 If the socket is readable return a tuple of the socket and the time taken  
 otherwise return None.  
 """  
  
 start = time.time()  
 is\_socket\_readable = select.select([sock], [], [], wait\_time)  
 taken = time.time() - start  
 if is\_socket\_readable[0] == []:  
 return float(-1)  
 else:  
 return taken

from modules import headers  
from modules import ip\_utils  
import socket  
import struct  
import time  
from collections import defaultdict  
from contextlib import closing  
from typing import Tuple, Set, DefaultDict  
  
  
PORTS = DefaultDict[str, Set[int]]  
  
  
def ping(  
 ID: int,  
 timeout: float  
) -> Set[Tuple[str, float, headers.ip]]:  
 """  
 Takes in a process id and a timeout and returns  
 a 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)  
 # opens a raw socket for sending ICMP protocol packets  
 time\_remaining = timeout  
 addresses = set()  
 recieved\_from = set()  
 while True:  
 time\_waiting = ip\_utils.wait\_for\_socket(ping\_sock, time\_remaining)  
 # time\_waiting stores the time the socket took to become readable  
 # or returns minus one if it ran out of time  
  
 if time\_waiting == -1:  
 break  
 time\_recieved = time.time()  
 # store the time the packet was recieved  
 recPacket, addr = ping\_sock.recvfrom(1024)  
 # recieve the packet  
 ip = headers.ip(recPacket[:20])  
 # unpack the IP header into its respective components  
 icmp = headers.icmp(recPacket[20:28])  
 # unpack the time from the packet.  
 time\_sent = struct.unpack(  
 "d",  
 recPacket[28:28 + struct.calcsize("d")]  
 )[0]  
 # unpack the value for when the packet was sent  
 time\_taken: float = time\_recieved - time\_sent  
 # calculate the round trip time taken for the packet  
 if icmp.id == ID:  
 # if the ping was sent from this machine then add it to the list of  
 # responses  
 ip\_address, port = addr  
 # this is to prevent a bug where IPs were being added twice  
 if ip\_address not in recieved\_from:  
 addresses.add((ip\_address, time\_taken, ip))  
 recieved\_from.add(ip\_address)  
 elif time\_remaining <= 0:  
 break  
 else:  
 continue  
 # return a list of all the addesses that replied to our ICMP echo request.  
 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])  
 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:  
 continue  
 ping\_sock.close()  
 return code  
  
  
def tcp(address: Tuple[str, int], timeout: float) -> PORTS:  
 """  
 This function is run asynchronously and listens for  
 TCP ACK responses to the sent TCP SYN msg.  
 """  
 ports: DefaultDict[str, Set[int]] = defaultdict(set)  
 with closing(  
 socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_RAW,  
 socket.IPPROTO\_TCP  
 )) as s:  
 s.bind(address)  
 # bind the raw socket to the listening address  
 time\_remaining = timeout  
 while True:  
 time\_taken = ip\_utils.wait\_for\_socket(s, time\_remaining)  
 # wait for the socket to become readable  
 if time\_taken == -1:  
 break  
 else:  
 time\_remaining -= time\_taken  
 packet = s.recv(1024)  
 # recieve the packet data  
 tcp = headers.tcp(packet[20:40])  
 if tcp.flags & 2: # syn flags set  
 ports["OPEN"].add(tcp.source)  
 elif tcp.flags & 4:  
 ports["CLOSED"].add(tcp.source)  
 else:  
 continue  
 return ports

import socket  
import time  
from modules import directives  
from modules import headers  
from modules import ip\_utils  
from modules import listeners  
from collections import defaultdict  
from contextlib import closing  
from itertools import repeat  
from multiprocessing import Pool  
from os import getpid  
from typing import Set, Tuple  
  
  
def ping(addresses: Set[str]) -> Set[Tuple[str, float, headers.ip]]:  
 """  
 Send an ICMP ECHO REQUEST to each address  
 in the set addresses. Then return a set which  
 contains all the addresses which replied and  
 which have the correct ID.  
 """  
 with closing(  
 socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_RAW,  
 socket.IPPROTO\_ICMP  
 )  
 ) as ping\_sock:  
 # get the local ip address  
 addresses = {  
 ip  
 for ip in addresses  
 if (  
 not ip.endswith(".0")  
 and not ip.endswith(".255")  
 )  
 }  
  
 # initialise a process pool  
 p = Pool(1)  
 # get the local process id for use in creating packets.  
 ID = getpid() & 0xFFFF  
 # run the listeners.ping function asynchronously  
 replied = p.apply\_async(listeners.ping, (ID, 5))  
 time.sleep(0.01)  
 for address in zip(addresses, repeat(1)):  
 try:  
 packet = ip\_utils.make\_icmp\_packet(ID)  
 ping\_sock.sendto(packet, address)  
 except PermissionError:  
 ip\_utils.eprint("raw sockets require root priveleges, exiting")  
 exit()  
 p.close()  
 p.join()  
 # close and join the process pool to so that all the values  
 # have been returned and the pool closed  
 return replied.get()  
  
  
def connect(address: str, ports: Set[int]) -> Set[int]:  
 """  
 This is the most basic kind of scan  
 it simply connects to every specififed port  
 and identifies whether they are open.  
 """  
 import socket  
 from contextlib import closing  
 open\_ports: Set[int] = set()  
 for port in ports:  
 # loop through each port in the list of ports to scan  
 try:  
 with closing(  
 socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_STREAM  
 )  
 ) as s:  
 # open an IPV4 TCP socket  
 s.connect((address, port))  
 # attempt to connect the newly created socket to the target  
 # address and port  
 open\_ports.add(port)  
 # if the connection was successful then add the port to the  
 # list of open ports  
 except (ConnectionRefusedError, OSError) as e:  
 pass  
 return open\_ports  
  
  
def tcp(dest\_ip: str, portlist: Set[int]) -> listeners.PORTS:  
 src\_port = ip\_utils.get\_free\_port()  
 # request a local port to connect from  
 if "127.0.0.1" == dest\_ip:  
 local\_ip = "127.0.0.1"  
 else:  
 local\_ip = ip\_utils.get\_local\_ip()  
 p = Pool(1)  
 listener = p.apply\_async(listeners.tcp, ((local\_ip, src\_port), 5))  
 time.sleep(0.01)  
 # start the TCP ACK listener in the background  
 for port in portlist:  
 # flag = 2 for syn scan  
 packet = ip\_utils.make\_tcp\_packet(  
 src\_port,  
 port,  
 local\_ip,  
 dest\_ip,  
 2  
 )  
 with closing(  
 socket.socket(  
 socket.AF\_INET,  
 socket.SOCK\_RAW,  
 socket.IPPROTO\_TCP  
 )  
 ) as s:  
 s.sendto(packet, (dest\_ip, port))  
 # send the packet to its destination  
 p.close()  
 p.join()  
 ports = listener.get()  
 ports["FILTERED"] = portlist - ports["OPEN"] - ports["CLOSED"]  
 if local\_ip == "127.0.0.1":  
 ports["OPEN"] -= set([src\_port])  
  
 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\_port = ip\_utils.get\_free\_port()  
 # get 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))  
 time.sleep(0.01)  
 # 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"]:  
 try:  
 packet = ip\_utils.make\_udp\_packet(  
 local\_port,  
 dest\_port  
 )  
 # 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())  
 # if we are on localhost remove the scanning port  
 if dest\_ip == "127.0.0.1":  
 ports["OPEN"] -= set([local\_port])  
 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  
 )  
 # make a new UDP packet  
 p = Pool(1)  
 icmp\_listen = p.apply\_async(  
 listeners.icmp\_unreachable,  
 (dest\_ip,),  
 )  
 # start the ICMP listener  
 time.sleep(0.01)  
 s.sendto(packet, (dest\_ip, dest\_port))  
 # send packet  
 p.close()  
 p.join()  
 icmp\_code = icmp\_listen.get()  
 # receive ICMP code from the ICMP listener  
 if icmp\_code in {0, 1, 2, 9, 10, 13}:  
 ports["FILTERED"].add(dest\_port)  
 elif icmp\_code == 3:  
 ports["CLOSED"].add(dest\_port)  
 except socket.error:  
 packet\_bytes = " ".join(map("{:02x}".format, packet))  
 ip\_utils.eprint(  
 "The socket modules sendto method with the following",  
 "argument resulting in a socket error.",  
 f"\npacket: [{packet\_bytes}]\n",  
 "address: [{dest\_ip, dest\_port}])"  
 )  
 # this creates a new set which contains all the elements that  
 # are in the list of ports to be scanned but have not yet  
 # been classified  
 ports["OPEN|FILTERED"] = (  
 ports["REMAINING"]  
 - ports["OPEN"]  
 - ports["FILTERED"]  
 - ports["CLOSED"]  
 )  
 del(ports["REMAINING"])  
 # set comprehension to update the list of open filtered ports  
 return ports  
  
  
def version\_detect\_scan(  
 target: directives.Target,  
 probes: directives.PROBE\_CONTAINER  
) -> directives.Target:  
 for probe\_dict in probes.values():  
 for proto in probe\_dict:  
 target = probe\_dict[proto].scan(target)  
 return target

## examples

#!/usr/bin/env python  
from icmp\_ping import icmp\_echo\_recv, icmp\_echo\_send  
from ping\_scanner import ping\_scan  
from tcp\_scan.connect\_scan import scan\_port\_list as connect\_scan\_list  
from tcp\_scan.syn\_scan import scan\_port\_list as syn\_scan\_list  
from udp\_scan import scan\_port\_list as udp\_scan\_list  
from version\_detection import version\_detection  
  
examples = {  
 "icmp\_echo\_recv": icmp\_echo\_recv.main,  
 "icmp\_echo\_send": icmp\_echo\_send.main,  
 "ping\_scanner": ping\_scan.main,  
 "connect\_scan": connect\_scan\_list.main,  
 "syn\_scan": syn\_scan\_list.main,  
 "udp\_scan": udp\_scan\_list.main,  
 "version\_detection": version\_detection.main,  
}  
  
print("\n\t".join(("Programs:", \*examples)))  
  
while True:  
 print()  
 program = input("Enter the name of the example program to run: ")  
 if program.lower() in {"quit", "q", "end", "exit"}:  
 break  
 found = False  
 for name in examples:  
 if name.startswith(program.lower()):  
 program = name  
 print(f"Running: {program}")  
 examples[program]()  
 found = True  
 if not found:  
 print(  
 "The program name must exactly match one of the following examples"  
 )  
 print("\n".join(examples))

## netscan

#!/usr/bin/env python  
import re  
from argparse import ArgumentParser  
from collections import defaultdict  
from math import floor, log10  
from modules import (  
 scanners,  
 ip\_utils,  
 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"  
)  
parser.add\_argument(  
 "-sV",  
 help="version scan",  
 action="store\_true"  
)  
parser.add\_argument(  
 "-p",  
 "--ports",  
 help="scan specified ports",  
 required=False,  
 default=top\_ports  
)  
parser.add\_argument(  
 "--exclude\_ports",  
 help="ports to exclude from the scan",  
 required=False,  
 default=""  
)  
  
args = parser.parse\_args()  
  
# check whether the address spec is in CIDR form  
CIDR\_regex = re.compile(r"(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})/(\d{1,2})")  
search = CIDR\_regex.search(args.target\_spec)  
if search:  
 base\_addr, network\_bits = search.groups()  
 addresses = ip\_utils.ip\_range(  
 base\_addr,  
 int(network\_bits)  
 )  
else:  
 base\_addr = args.target\_spec  
 if not ip\_utils.is\_valid\_ip(base\_addr):  
 raise ValueError(f"invalid dot form IP address: [{base\_addr}]")  
 addresses = {base\_addr}  
  
  
def error\_exit(error\_type: str, scan\_type: str, scanning: str) -> bool:  
 messages = {  
 "permission": "\n".join((  
 "You have insufficient permissions to run this type of scan",  
 "EXITING!"  
 ))  
 }  
 print(f"You tried to scan {scanning} using scan type: {scan\_type}")  
 try:  
 print(messages[error\_type])  
 except KeyError:  
 print(f"ERROR MESSAGE NOT FOUND: {error\_type}")  
 exit(-1)  
  
  
if args.sL:  
 print("Targets:")  
 print("\n".join(sorted(addresses, key=ip\_utils.dot\_to\_long)))  
else:  
 if args.sn:  
 def sig\_figs(x: float, n: int) -> float:  
 """  
 rounds x to n significant figures.  
 sig\_figs(1234, 2) = 1200.0  
 """  
 return round(x, n - (1 + int(floor(log10(abs(x))))))  
  
 try:  
 print("\n".join(  
 f"host: [{host}]\t" +  
 "responded to an ICMP ECHO REQUEST in " +  
 f"{str(sig\_figs(taken, 2))+'s':<10s} " +  
 f"ttl: [{ip\_head.time\_to\_live}]"  
 for host, taken, ip\_head in scanners.ping(addresses)  
 ))  
 except PermissionError:  
 error\_exit("permission", "ping scan", str(addresses))  
  
 else:  
 if args.Pn:  
 targets = [  
 directives.Target(  
 addr,  
 defaultdict(set),  
 defaultdict(set)  
 )  
 for addr in addresses  
 ]  
 else:  
 try:  
 targets = [  
 directives.Target(  
 addr,  
 defaultdict(set),  
 defaultdict(set),  
 )  
 for addr, \_, \_ in scanners.ping(addresses)  
 ]  
 except PermissionError:  
 error\_exit("permission", "ping\_scan", str(addresses))  
 # define the ports to scan  
 if args.ports == "-":  
 # case they have specified all ports  
 ports = {  
 "UDP": set(range(1, 65536)),  
 "TCP": set(range(1, 65536)),  
 }  
 elif isinstance(args.ports, str):  
 # case they have specifed ports  
 ports = directives.parse\_ports(args.ports)  
 else:  
 # default  
 ports = args.ports  
  
 # exclude all the ports speified to be excluded  
 to\_exclude = directives.parse\_ports(args.exclude\_ports)  
 ports["TCP"] -= to\_exclude["TCP"]  
 ports["TCP"] -= to\_exclude["ANY"]  
 ports["UDP"] -= to\_exclude["UDP"]  
 ports["UDP"] -= to\_exclude["ANY"]  
  
 # if version scanning is desired  
 if args.sV:  
 probes = directives.parse\_probes(  
 "./version\_detection/nmap-service-probes"  
 )  
  
 for target in targets:  
 if not args.sU and not args.sT or args.sS:  
 try:  
 tcp\_ports = scanners.tcp(  
 target.address,  
 ports["TCP"] | ports["ANY"]  
 )  
 except PermissionError:  
 error\_exit("permission", "tcp\_scan", target.address)  
 target.open\_ports["TCP"].update(tcp\_ports["OPEN"])  
 target.open\_filtered\_ports["TCP"].update(  
 tcp\_ports["FILTERED"]  
 )  
 if args.sT:  
 target.open\_ports["TCP"].update(  
 scanners.connect(  
 target.address,  
 ports["TCP"] | ports["ANY"]  
 )  
 )  
 if args.sU:  
 try:  
 udp\_ports = scanners.udp(  
 target.address,  
 ports["UDP"] | ports["ANY"]  
 )  
 except PermissionError:  
 error\_exit("permission", "udp\_scan", target.address)  
  
 target.open\_ports["UDP"].update(  
 udp\_ports["OPEN"]  
 )  
 target.open\_filtered\_ports["UDP"].update(  
 udp\_ports["FILTERED"]  
 )  
 target.open\_filtered\_ports["UDP"].update(  
 udp\_ports["OPEN|FILTERED"]  
 )  
 if args.sV:  
 target = scanners.version\_detect\_scan(target, probes)  
 # display scan info  
 print()  
 print(f"Scan report for: {target.address}")  
 # print(target)  
 print("Open ports:")  
 for proto, open\_ports in target.open\_ports.items():  
 for port in open\_ports:  
 try:  
 service\_name = services[proto][port]  
 except KeyError:  
 service\_name = "unknown"  
 if port in target.services:  
 exact\_match = target.services[port]  
 print(  
 f"{port}/{proto}{exact\_match.service:>8s}"  
 )  
 # print version information  
 for key, val in exact\_match.version\_info.items():  
 print(f"{key}: {val}")  
 if exact\_match.cpes:  
 print()  
 print("CPE:")  
 for cpe\_type, cpe\_vals in exact\_match.cpes.items():  
 print(cpe\_type)  
 try:  
 del(cpe\_vals["part"])  
 except KeyError:  
 pass  
 for key, val in cpe\_vals.items():  
 print(f"{key}: {val}")  
 print()  
 else:  
 print(f"{port} service: {service\_name}?")  
  
 print("Filtered ports:")  
 for proto, filtered\_ports in target.open\_filtered\_ports.items():  
 for port in filtered\_ports:  
 try:  
 service\_name = services[proto][port]  
 except KeyError:  
 service\_name = "unknown"  
 print(f"{port} service: {service\_name}?")

## tests

from modules.ip\_utils import (  
 dot\_to\_long,  
 long\_to\_dot,  
 ip\_range,  
 is\_valid\_ip,  
 is\_valid\_port\_number,  
 ip\_checksum,  
 make\_tcp\_packet,  
 make\_udp\_packet,  
 make\_icmp\_packet,  
)  
from binascii import unhexlify  
  
  
def test\_dot\_to\_long\_private\_ip() -> None:  
 assert(dot\_to\_long("192.168.1.0") == 0xC0A80100)  
  
  
def test\_long\_to\_dot\_private\_ip() -> None:  
 assert(long\_to\_dot(0xC0A80100) == "192.168.1.0")  
  
  
def test\_dot\_to\_long\_localhost() -> None:  
 assert(dot\_to\_long("127.0.0.1") == 0x7F000001)  
  
  
def test\_long\_to\_dot\_localhost() -> None:  
 assert(long\_to\_dot(0x7F000001) == "127.0.0.1")  
  
  
def test\_is\_valid\_ip\_localhost\_long() -> None:  
 assert is\_valid\_ip(0x7F000001)  
  
  
def test\_is\_valid\_ip\_localhost() -> None:  
 assert is\_valid\_ip("127.0.0.1")  
  
  
def test\_is\_not\_valid\_ip\_5\_zeros\_dotted() -> None:  
 assert not is\_valid\_ip("0.0.0.0.0")  
  
  
def test\_is\_not\_valid\_ip\_5\_255s\_long() -> None:  
 assert not is\_valid\_ip(0xFF\_FF\_FF\_FF\_FF)  
  
  
def test\_is\_valid\_port\_number\_0() -> None:  
 assert is\_valid\_port\_number(0)  
  
  
def test\_is\_valid\_port\_number\_65535() -> None:  
 assert is\_valid\_port\_number(65535)  
  
  
def test\_is\_not\_valid\_port\_number\_negative\_one() -> None:  
 assert not is\_valid\_port\_number(-1)  
  
  
def test\_is\_not\_valid\_port\_number\_65536() -> None:  
 assert not is\_valid\_port\_number(65536)  
  
  
def test\_ip\_range() -> None:  
 assert(  
 ip\_range("192.168.1.0", 28) == {  
 "192.168.1.1",  
 "192.168.1.2",  
 "192.168.1.3",  
 "192.168.1.4",  
 "192.168.1.5",  
 "192.168.1.6",  
 "192.168.1.7",  
 "192.168.1.8",  
 "192.168.1.9",  
 "192.168.1.10",  
 "192.168.1.11",  
 "192.168.1.12",  
 "192.168.1.13",  
 "192.168.1.14",  
 }  
 )  
  
  
def test\_ip\_checksum\_verify() -> None:  
 packet = unhexlify(  
 "45000073000040004011b861c0a80001c0a800c7"  
 )  
 assert ip\_checksum(packet) == 0  
  
  
def test\_ip\_checksum\_generate() -> None:  
 packet = unhexlify(  
 "450000730000400040110000c0a80001c0a800c7"  
 )  
 assert ip\_checksum(packet) == 0xB861  
  
  
def test\_make\_tcp\_packet() -> None:  
 correct = unhexlify(  
 "e54700500000000000000000600204002af50000020405b4"  
 )  
 info = 58695, 80, "192.168.1.45", "192.168.1.28", 2  
 assert correct == make\_tcp\_packet(\*info)  
  
  
def test\_make\_udp\_packet() -> None:  
 correct = unhexlify(  
 "e5470050003a0000"  
 )  
 info = 58695, 80  
 # clipping the packet at 8 simply removes the data section  
 assert correct == make\_udp\_packet(\*info)[:8]

from modules.directives import (  
 parse\_ports  
)  
from collections import defaultdict  
from typing import DefaultDict  
  
  
def test\_parse\_probes\_single() -> None:  
 portstring = "12345"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["ANY"] = set([12345])  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_range() -> None:  
 portstring = "10-20"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["ANY"] = set(range(10, 21))  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_single\_and\_range() -> None:  
 portstring = "1,2,3,10-20,6,7,8"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["ANY"] = set([1, 2, 3, \*range(10, 21), 6, 7, 8])  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_tcp\_single() -> None:  
 portstring = "T:12345"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["TCP"] = set([12345])  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_tcp\_range() -> None:  
 portstring = "T:10-20"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["TCP"] = set(range(10, 21))  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_tcp\_single\_and\_range() -> None:  
 portstring = "T:1,2,3,10-20,6,7,8"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["TCP"] = set([1, 2, 3, \*range(10, 21), 6, 7, 8])  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_udp\_single() -> None:  
 portstring = "U:12345"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["UDP"] = set([12345])  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_udp\_range() -> None:  
 portstring = "U:10-20"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["UDP"] = set(range(10, 21))  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_udp\_single\_and\_range() -> None:  
 portstring = "U:1,2,3,10-20,6,7,8"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["UDP"] = set([1, 2, 3, \*range(10, 21), 6, 7, 8])  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_any\_and\_tcp\_single() -> None:  
 portstring = "12345 T:12345"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["TCP"] = set([12345])  
 expected["ANY"] = set([12345])  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_any\_and\_tcp\_range() -> None:  
 portstring = "10-20 T:10-20"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["TCP"] = set(range(10, 21))  
 expected["ANY"] = set(range(10, 21))  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_any\_and\_tcp\_single\_and\_range() -> None:  
 portstring = "1,2,3,10-20,6,7,8 T:1,2,3,10-20,6,7,8"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["TCP"] = set([1, 2, 3, \*range(10, 21), 6, 7, 8])  
 expected["ANY"] = set([1, 2, 3, \*range(10, 21), 6, 7, 8])  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_any\_and\_udp\_single() -> None:  
 portstring = "12345 U:12345"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["UDP"] = set([12345])  
 expected["ANY"] = set([12345])  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_any\_and\_udp\_range() -> None:  
 portstring = "10-20 U:10-20"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["UDP"] = set(range(10, 21))  
 expected["ANY"] = set(range(10, 21))  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_any\_and\_udp\_single\_and\_range() -> None:  
 portstring = "1,2,3,10-20,6,7,8 U:1,2,3,10-20,6,7,8"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["UDP"] = set([1, 2, 3, \*range(10, 21), 6, 7, 8])  
 expected["ANY"] = set([1, 2, 3, \*range(10, 21), 6, 7, 8])  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_udp\_and\_tcp\_single() -> None:  
 portstring = "U:12345 T:12345"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["TCP"] = set([12345])  
 expected["UDP"] = set([12345])  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_udp\_and\_tcp\_range() -> None:  
 portstring = "U:10-20 T:10-20"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["TCP"] = set(range(10, 21))  
 expected["UDP"] = set(range(10, 21))  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_udp\_and\_tcp\_single\_and\_range() -> None:  
 portstring = "U:1,2,3,10-20,6,7,8 T:1,2,3,10-20,6,7,8"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["TCP"] = set([1, 2, 3, \*range(10, 21), 6, 7, 8])  
 expected["UDP"] = set([1, 2, 3, \*range(10, 21), 6, 7, 8])  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_all\_single() -> None:  
 portstring = "12345 U:12345 T:12345"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["TCP"] = set([12345])  
 expected["UDP"] = set([12345])  
 expected["ANY"] = set([12345])  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_all\_range() -> None:  
 portstring = "10-20 U:10-20 T:10-20"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["TCP"] = set(range(10, 21))  
 expected["UDP"] = set(range(10, 21))  
 expected["ANY"] = set(range(10, 21))  
 assert expected == parse\_ports(portstring)  
  
  
def test\_parse\_probes\_all\_single\_and\_range() -> None:  
 portstring = "1,2,3,10-20,6,7,8 U:1,2,3,10-20,6,7,8 T:1,2,3,10-20,6,7,8"  
 expected: DefaultDict[str, set] = defaultdict(set)  
 expected["TCP"] = set([1, 2, 3, \*range(10, 21), 6, 7, 8])  
 expected["UDP"] = set([1, 2, 3, \*range(10, 21), 6, 7, 8])  
 expected["ANY"] = set([1, 2, 3, \*range(10, 21), 6, 7, 8])  
 assert expected == parse\_ports(portstring)