I did the interactive tutorial

First steps

- Apparently scapy has its own CLI and allows me to create a packet object and modify its contents

```
>>> a = IP(ttl=10)
>>> a
<IP  ttl=10 |>
>>> a.src
'127.0.0.1' # the default packet source is lo
>>> a.dst="192.168.1.1"
>>> a.dst
'192.168.1.1' # i can modify each field of a packet
>>> a.ttl
10
>>> a.ttl = 64
>>> a.ttl
64
>>> del(a.ttl) # i can also delete a field of a packet
>>> a
<IP  dst=192.168.1.1 |>
```

Stacking Layers

I can stack layers using an overriden `/` operator.

Let's create a HTTP packet. The raw layer can just be a string, simple as that.

```
>>> IP()/TCP()/"GET / HTTP/1.0\r\n\r\n"
<IP  frag=0 proto=tcp |<TCP  |<Raw  load=b'GET / HTTP/1.0\r\n\r\n' |>>>
```

Let's create a HTTP packet and send it to google.com

```
>>> send(IP(dst="google.com")/TCP()/"GET / HTTP/1.0\r\n\r\n")
.
Sent 1 packets.
```

Proto parameter

The `proto` parameter is an integer value, that represents the protocol used in the packet. For example:

- TCP is 6 (or just `proto=tcp`)
- UDP is 17

```
>>> IP(dst="google.com", proto=6)/TCP()/"GET / HTTP/1.0\r\n\r\n"
```

Reading PCAP files

Wireshark is way better for this. Even tshark is better for this.

Dump of a packet

```
>>> a = IP()/TCP()/"GET / HTTP/1.0\r\n\r\n"
>>> a.show()
###[ IP ]###
  version   = 4
  ihl       = None
  tos       = 0x0
  len       = None
  id        = 1
  flags     =
  frag      = 0
  ttl       = 64
  proto     = tcp
  chksum    = None
  src       = 127.0.0.1
  dst       = 127.0.0.1
  \options   \
###[ TCP ]###
     sport     = ftp_data
     dport     = www_http
     seq       = 0
     ack       = 0
     dataofs   = None
     reserved  = 0
     flags     = S
     window    = 8192
     chksum    = None
```

```
      urgptr    = 0
      options   = []
###[ Raw ]###
         load      = b'GET / HTTP/1.0\r\n\r\n'
```

Multicast on layer 3

Multicast is a way to send a packet to multiple hosts at once.

- Multicast is done using the `IP` layer

Sending and receiving packets

- `sr` in scapy is a function that sends a packet and waits for a response
- `sr1` is a function that sends a packet and waits for a single response
- `srp` is a function that sends a packet and waits for a response, but it uses the link layer instead of the IP layer.

```
>>> sr(IP(dst="google.com")/TCP()/"GET / HTTP/1.0\r\n\r\n")
Begin emission
.
Finished sending 1 packets
*
Received 2 packets, got 1 answers, remaining 0 packets
(<Results: TCP:1 UDP:0 ICMP:0 Other:0>, <Unanswered: TCP:0 UDP:0 ICMP:0
Other:0>)

>>> a=sr1(IP(dst="onet.pl")/TCP()/"GET / HTTP/1.0\r\n\r\n")
Begin emission

Finished sending 1 packets
*
Received 1 packets, got 1 answers, remaining 0 packets
<IP  version=4 ihl=5 tos=0x0 len=44 id=0 flags=DF frag=0 ttl=248 proto=tcp
chksum=0xbc70 src=13.227.146.66 dst=10.26.28.28 |<TCP  sport=www_http
dport=ftp_data seq=3489292617 ack=1 dataofs=6 reserved=0 flags=SA
window=65535 chksum=0xa026 urgptr=0 options=[('MSS', 1440)] |<Padding
load=b'\x00\x00'
>>> a.show()
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = 44
  id        = 0
  flags     = DF
  frag      = 0
  ttl       = 248
  proto     = tcp
  chksum    = 0xbc70
  src       = 13.227.146.66
```

```
  dst       = 10.26.28.28
  \options   \
###[ TCP ]###
     sport     = www_http
     dport     = ftp_data
     seq       = 3489292617
     ack       = 1
     dataofs   = 6
     reserved  = 0
     flags     = SA
     window    = 65535
     chksum    = 0xa026
     urgptr    = 0
     options   = [('MSS', 1440)]
###[ Padding ]###
        load      = b'\x00\x00'
```

SYNScan

- SYNScan is a way to scan for open ports on a host.
- It works by sending a SYN packet to the host and waiting for a response.
- If the host responds with a SYN-ACK packet, it means that the port is open.
- If the host responds with a RST packet, it means that the port is closed.

```
>>> a = sr1(IP(dst="google.com")/TCP(dport=80,flags="S"))
Begin emission

Finished sending 1 packets
*
Received 1 packets, got 1 answers, remaining 0 packets
<IP  version=4 ihl=5 tos=0x0 len=44 id=0 flags=DF frag=0 ttl=122 proto=tcp
chksum=0x15ef src=172.217.23.206 dst=10.26.28.28 |<TCP  sport=www_http
dport=ftp_data seq=1514712537 ack=1 dataofs=6 reserved=0 flags=SA
window=65535 chksum=0xa4e2 urgptr=0 options=[('MSS', 1412)] |<Padding
load=b'\x00\x00' |
>>> a.show()
```

We can try to make a bad nmap using scapy.

```
from scapy.all import *
from scapy.layers.inet import IP, TCP

def syn_scan(target, ports):
    open_ports = []
    for port in ports:
        # Create a SYN packet
        syn_packet = IP(dst=target)/TCP(dport=port, flags="S")
        # Send the packet and wait for a response
        response = sr1(syn_packet, timeout=1, verbose=0)
```

```
        if response and response.haslayer(TCP):
            if response[TCP].flags == 0x12:  # SYN-ACK
                open_ports.append(port)
                # Send RST to close the connection
                rst_packet = IP(dst=target)/TCP(dport=port, flags="R")
                send(rst_packet, verbose=0)
    return open_ports

if __name__ == "__main__":
    print(syn_scan("google.com", [22, 80, 443, 8080]))
```

ans/unans

- ans is the list of answered packets
- unans is the list of unanswered packets

```
>>> ans, unans = sr(IP(dst="google.com")/TCP()/"GET / HTTP/1.0\r\n\r\n")
Begin emission
.
Finished sending 1 packets
.*
Received 3 packets, got 1 answers, remaining 0 packets
>>> ans.summary( lambda s,r: r.sprintf("%TCP.sport% \t %TCP.flags%") )
www_http         SA
```

A simple traceroute

```
from scapy.all import IP, ICMP, sr1
import argparse
import socket
import time

ICMP_TIME_EXCEEDED = 11
ICMP_ECHO_REPLY = 0

def traceroute(destination, max_hops=30, timeout=1, verbose=False):
    """
    Perform a traceroute to a destination using ICMP or UDP packets.
    """

    try:
        dest_ip = socket.gethostbyname(destination)
    except socket.gaierror:
        print(f"Could not resolve {destination}")
        return

    print(f"Traceroute to {destination} ({dest_ip}), {max_hops} hops max")

    for ttl in range(1, max_hops + 1):
```

```
        packet = IP(dst=dest_ip, ttl=ttl) / ICMP()
        start_time = time.time()
        reply = sr1(packet, verbose=0, timeout=timeout)
        rtt = (time.time() - start_time) * 1000

        if reply is None:
            print(f"{ttl}\t*")
            if verbose:
                print(f"Timeout waiting for TTL {ttl}")
        elif reply.type == ICMP_TIME_EXCEEDED:
            print(f"{ttl}\t{reply.src}\t{rtt:.2f} ms")
            if verbose:
                print(f"Received ICMP Time Exceeded from {reply.src}")
        elif reply.type == ICMP_ECHO_REPLY:
            print(f"{ttl}\t{reply.src}\t{rtt:.2f} ms")
            print("Destination reached!")
            break
        else:
            print(f"{ttl}\t{reply.src}\t{rtt:.2f} ms")
            if verbose:
                print(f"Received ICMP type {reply.type} from {reply.src}")

            if reply.src == dest_ip:
                print("Destination reached!")
                break

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Simple traceroute
implementation using Scapy")
    parser.add_argument("destination", help="Destination hostname or IP
address")
    parser.add_argument("-m", "--max-hops", type=int, default=30,
                        help="Maximum number of hops (default: 30)")
    parser.add_argument("-t", "--timeout", type=int, default=2,
                        help="Timeout in seconds for each reply (default:
2)")
    parser.add_argument("-v", "--verbose", action="store_true",
                        help="Show verbose output")

    args = parser.parse_args()

    traceroute(
        args.destination,
        max_hops=args.max_hops,
        timeout=args.timeout,
        verbose=args.verbose
    )
```

Packet sniffing

- `sniff` is a function that captures packets from the network (from a specific interface)

```
>>> sniff(iface="enp0s31f6", count=10)
<Sniffed: TCP:8 UDP:0 ICMP:0 Other:2>
>>> sniff(iface="enp0s31f6", filter="tcp and port 443", count=10)
<Sniffed: TCP:10 UDP:0 ICMP:0 Other:0>
>>> sniff(iface="enp0s31f6", filter="tcp and port 443", count=10,
prn=lambda x: x.show())
[...]
###[ Ethernet ]###
  dst       = 6c:3c:8c:53:0a:76
  src       = b4:0c:25:e0:40:10
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 91
     id        = 27177
     flags     =
     frag      = 0
     ttl       = 122
     proto     = tcp
     chksum    = 0xbd4a
     src       = 34.120.208.123
     dst       = 10.26.28.28
     \options   \
###[ TCP ]###
        sport     = https
        dport     = 46374
        seq       = 285179649
        ack       = 1585944215
        dataofs   = 8
        reserved  = 0
        flags     = PA
        window    = 1004
        chksum    = 0xdbc0
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp',
(3903348397, 2328815939))]
###[ Raw ]###
           load      =
b'\x17\x03\x03\x00"\x92[\x8b\x9a\xdcv#\x93\x0f\xddu2:\x87\x91\x0e\xf3\x9cn\
r\xfb\xed\x00M0\x9d\xe9\xba\xa9\x97\xefy\xcbU'

<Sniffed: TCP:10 UDP:0 ICMP:0 Other:0>
```

Simple Wireshark

We account on a few things:

- Program must run in cli with root privileges (interface sniffing)
- Program must present a summary of the packets captured

- We must be able to quit the program with `Ctrl+C` (SIGINT)

```
from scapy.all import Ether, IP, TCP, UDP, ICMP, sniff
import argparse
import signal
import sys
from datetime import datetime

packet_count = 0
start_time = None

def signal_handler(sig, frame):
    print("\n--- Packet capture summary ---")
    if start_time:
        duration = datetime.now() - start_time
        print(f"Capture duration: {duration}")
    print(f"Total packets captured: {packet_count}")
    sys.exit(0)

def packet_callback(packet, display_filter=None, verbose=False,
output_file=None):
    global packet_count

    packet_count += 1
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f")[:-3]

    if display_filter and not display_filter(packet):
        return

    output = f"[{timestamp}]#{packet_count}:{len(packet)} bytes"

    if Ether in packet:
        output += f"\n  Ethernet: {packet[Ether].src} ->
{packet[Ether].dst}"

    if IP in packet:
        ip = packet[IP]
        output += f"\n  IP: {ip.src}:{ip.sport if hasattr(ip, 'sport') else
'?'} -> " \
                  f"{ip.dst}:{ip.dport if hasattr(ip, 'dport') else '?'} " \
                  f"(Proto: {ip.proto}, TTL: {ip.ttl})"

    if TCP in packet:
        tcp = packet[TCP]
        output += f"\n  TCP: Flags: {tcp.flags}, Seq: {tcp.seq}, Ack:
{tcp.ack}"

    elif UDP in packet:
        udp = packet[UDP]
        output += f"\n  UDP: Length: {udp.len}, Checksum: {udp.chksum}"

    elif ICMP in packet:
        icmp = packet[ICMP]
```

```
            output += f"\n  ICMP: Type: {icmp.type}, Code: {icmp.code}"

        if verbose:
            output += f"\n  Full packet:\n{packet.show(dump=True)}"

        print(output)

        if output_file:
            with open(output_file, 'a') as f:
                f.write(output + "\n")

    def create_display_filter(filter_str):
        """Create a display filter function from a string"""
        if not filter_str:
            return None

        def filter_func(packet):
            try:
                return packet.haslayer(filter_str)
            except:
                return filter_str.lower() in str(packet).lower()

        return filter_func

    def main():
        global start_time

        parser = argparse.ArgumentParser(description="Packet sniffer using
    Scapy")
        parser.add_argument("-i", "--interface", default=None, help="Network
    interface to sniff on (default: auto-detect)")
        parser.add_argument("-f", "--filter", default="", help="Filter to apply
    (e.g., 'tcp port 80')")
        parser.add_argument("-d", "--display-filter", default="", help="Display
    filter to show only matching packets")
        parser.add_argument("-c", "--count", type=int, default=0, help="Number
    of packets to capture (0 for unlimited)")
        parser.add_argument("-v", "--verbose", action="store_true", help="Show
    verbose packet output")
        parser.add_argument("-o", "--output", default=None, help="Output file
    to save captured packets")
        parser.add_argument("-t", "--timeout", type=int, default=10,
    help="Capture timeout in seconds (10 default)")

        args = parser.parse_args()

        signal.signal(signal.SIGINT, signal_handler)

        print(f"Starting packet capture on interface {args.interface or
    'default'}...")
        print(f"Filter: {args.filter or 'none'}")
        if args.display_filter:
            print(f"Display filter: {args.display_filter}")
        if args.count > 0:
```

```
        print(f"Capturing {args.count} packets")
    if args.timeout > 0:
        print(f"Timeout: {args.timeout} seconds")

    start_time = datetime.now()

    display_filter = create_display_filter(args.display_filter)

    sniff(
        iface=args.interface,
        prn=lambda p: packet_callback(p, display_filter, args.verbose,
args.output),
        filter=args.filter,
        count=args.count,
        timeout=args.timeout
    )

    signal_handler(None, None)

if __name__ == "__main__":
    main()
```