Introduction:

I used Ubuntu 18.04 on a Virtual Machine on VMware Fusion on my Macbook Pro. I coded the project in Python 3.6.

I used the official documentation on pcapy from here:

- https://rawgit.com/CoreSecurity/pcapy/master/pcapy.html#idp1073152058096

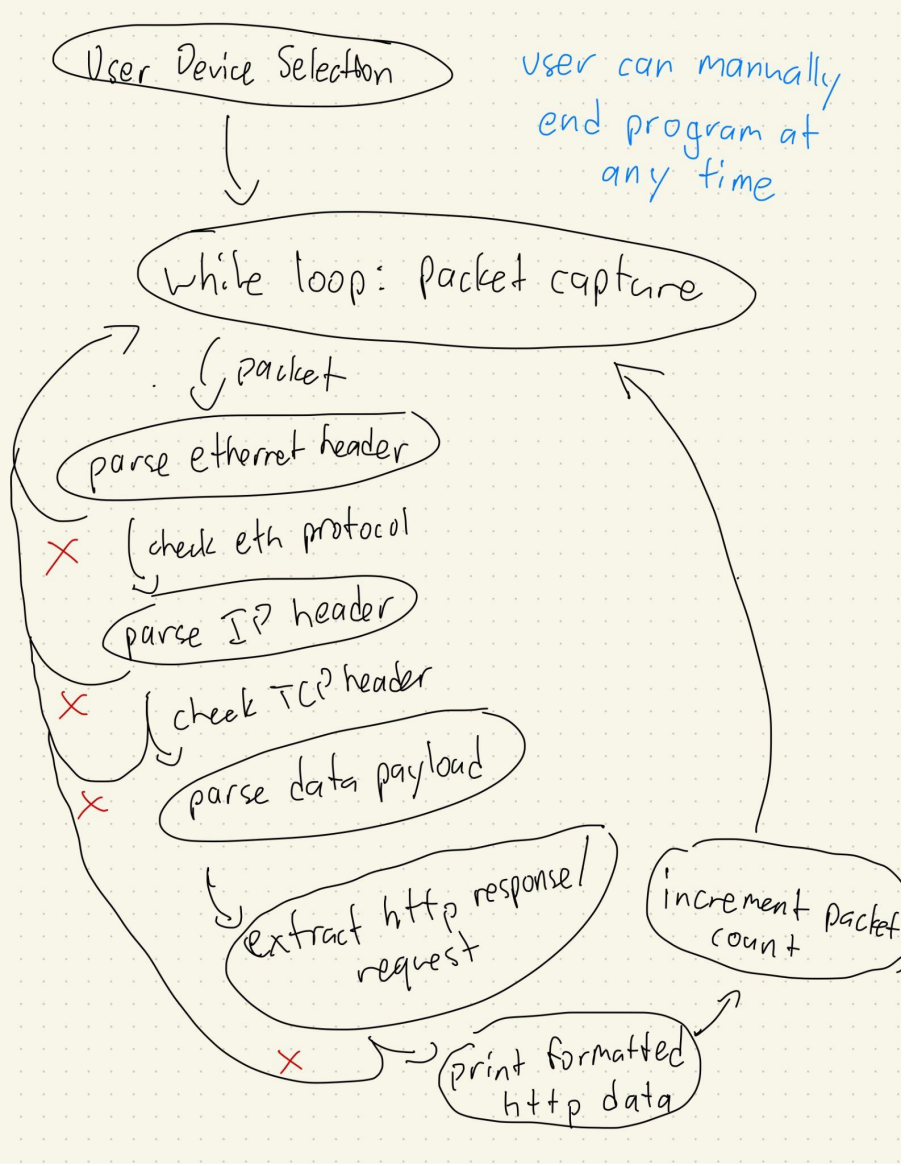
And I used the documentation about pcap from here:

- https://www.tcpdump.org/pcap.html

I used the code examples here on how to use the socket library to extract port and ip numbers:

- https://www.programcreek.com/python/example/9876/socket.ntohs

Flow Chart:



Logic Explanations:

Device Selection:

```
# DEVICE SELECTION ================================================================
    # Find all available devices
    devList = pcapy.findalldevs()

    # Construct prompt string
    input_string = "Enter the # for the device (Press Enter for 0 as default): \n"
    for number, option in enumerate(devList):
        input_string += "Device [" + str(number) + "]: " + option + "\n"

    # Prompt user for input
    deviceNum = int(input(input_string) or "0")
    device = devList[deviceNum]
    print("You have selected: " + str(deviceNum) + "\n")
    print("Press CTRL+C to exit the program at any time.")
    print("Currently sniffing network traffic on [" + device.upper() + "]...")

    # Open device
    #   Arg 1: Device
    #   Arg 2: Snaplen - max # of bytes to capture per packet
    #   Arg 3: Promiscious Mode - Set to True
    #   Arg 4: Timeout - In milliseconds
    capture = pcapy.open_live(device, 65536, True, 0)
# ================================================================================
```

- In this block I use pcapy to find all network devices, then display all device options to the user via command line and ask them to choose a device, with the default option being the first device.
- Once the user selects a device, I then begin capturing all network traffic on the device with pcapy.open_live

Packet Sniffing

```
# PACKET SNIFFING ================================================================
    # Set filter to reduce unwanted traffic
    capture.setfilter("tcp port 80")

    # Start packet sniffing
    packet_num = 1
    while True:
        # Capture the next packet header and packet data
        header, packet = capture.next()

        # Call the packet_helper function to get/print required data
        packet_num = packet_helper(packet, packet_num)
# ================================================================================
```

- In this block I set a filter on the capture device to reduce unwanted traffic.
- Then, after creating a counter for the HTTP response/requests, I capture the next packet and send it to the packet_helper helper function to extract all the required information from the packet.
- Until the user manually stops the program, it will continue looping in the while loop and capturing packets on the capture device.

Vars

```
# Helper function to break down data inside the packet and convert it to readable form
def packet_helper(packet, packet_num):
# VARS ============================================================================================
    # Ethernet
    eth_hdr_len = 14

    # IP
    ip_hdr_len = 20
    ip_src = ""
    ip_dest = ""

    # TCP
    tcp_hdr_len = -1
    tcp_src_port = ""
    tcp_dest_port = ""

    # Data Payload
    data_payload_len = -1
    data_payload = ""
# ================================================================================================
```

- This code block is all the variables I will be using in the final print, so I stored them in an easy to see place for me to reference back to.

Parse Ethernet Header

```
# PARSE ETHERNET HEADER ========================================
    # Unpack Ethernet header data using struct.unpack
    eth_hdr = packet[:eth_hdr_len]
    eth = struct.unpack('!6s6sH', eth_hdr)
    eth_dest_mac = ethernet_address(packet[0:6])
    eth_src_mac = ethernet_address(packet[6:12])

    # Grab ethernet protocol using socket library
    eth_proto = socket.ntohs(eth[2])
# ==============================================================
```

```
# Convert to readable ethernet address
def ethernet_address(raw_addr):
    addr = "%.2x:%.2x:%.2x:%.2x:%.2x:%.2x" % (raw_addr[0], raw_addr[1], raw_addr[2], raw_addr[3], raw_addr[4], raw_addr[5])
    return addr
```

- In this code block I unpack the Ethernet header data using the unpack function in the struct library, which performs conversions between Python values and C structs represented as Python bytes objects.
- I pass the extracted data through a helper function in the second photo to decode it into a readable format, and then save them in the variable fields from the previously explained code block.
- I use the socket library to extract the ethernet protocol from the Ethernet header data, and use this information in the next code block to determine if there is an IP header to parse

Parse IP Header

```
    if eth_proto == 8:
# PARSE IP HEADER ====================================================
        # Unpack IP Header data using struct.unpack
        ip_hdr = packet[eth_hdr_len:ip_hdr_len + eth_hdr_len]
        ip = struct.unpack('!BBHHHBBH4s4s', ip_hdr)

        # Grab IP Protocol to check later if TCP
        ip_proto = ip[6]

        # Grab source/destination IP addresses using socket library
        ip_src = socket.inet_ntoa(ip[8])
        ip_dest = socket.inet_ntoa(ip[9])
# ====================================================================
```

- In this code block, I determine whether the protocol points to there being an IP header, then I begin to unpack the IP header data using struct.unpack. I use the lengths of the Ethernet header and the IP header to determine which characters to extract data from
- I also grab the IP protocol to use in the next code block.
- Finally, I use the socket library again to extract the required IP addresses and save them in the variable fields from before.

Parse TCP Header

```
        if ip_proto == 6:
# PARSE TCP HEADER ============================================
            # Unpack TCP Header using struct.unpack
            tcp_hdr_offset = eth_hdr_len + ip_hdr_len
            tcp_hdr = packet[tcp_hdr_offset:tcp_hdr_offset + 20]
            tcp = struct.unpack('!HHLLBBHHH', tcp_hdr)

            # Get TCP Header length using bit shift
            tcp_hdr_len = tcp[4] >> 4

            # Grab source/destination TCP ports
            tcp_src_port = tcp[0]
            tcp_dest_port = tcp[1]
# ============================================================
```

- In this code block, I determine whether the protocol points to there being a TCP header, then I begin to unpack the TCP header using struct.unpack.
- Again, I use the lengths of the ethernet header and ip header to determine what range of characters to extract data from.
- I then extract the TCP header length using bit shift.
- Finally, I use the socket library again to extract the required TCP ports and save them in the variable fields from before.

Parse Data Payload

```
# PARSE DATA PAYLOAD ==========================================================
            # Calculate full header length
            header_len = eth_hdr_len + ip_hdr_len + tcp_hdr_len * 4

            # Extract packet data
            data_payload = packet[header_len:]
# =============================================================================
```

- In preparation for extracting the HTTP response/request I calculate the full header length and then extract all the data that comes after the full header.
- I save that data into data_payload and use it in the next code block

Extract HTTP Requests

```
# EXTRACT HTTP REQUESTS ================================================================================================
    # Ignore bit annotation to the string after conversion
    data = str(data_payload)[2:-1]

    # Check if data is a HTTP response of request
    if data and ("HTTP" or "GET" or "HEAD" or "POST" or "PUT" or "DELETE" or "CONNECT" or "OPTIONS" or "TRACE") in data:
        clean_print(ip_src, ip_dest, tcp_src_port, tcp_dest_port, data, packet_num)

        # If an HTTP response/request that sucessfully prints, increment by 1
        return 1 + packet_num
    # Else, don't increment by 1
    return 0 + packet_num
# ======================================================================================================================
```

- In this code block I trim the data after converting the data_payload to a string, to remove the bit annotation.
- Then, I check the string to see if it contains an HTTP method or “HTTP”. If it does, it sends all the required information to a printing helper function to properly format the output. If it doesn’t, nothing happens.
- After successfully printing, the function returns the packet number incremented by one to properly label the packets.

Clean Print

```
def clean_print(ip_src, ip_dest, tcp_src_port, tcp_dest_port, data, num):
    # Separate substrings by \r\n to get request/header lines
    sliced_data = data.split("\\r\\n")

    # Build the HTTP_line string from sliced data
    HTTP_line = str(num) + " " + \
                str(ip_src) + ":" + \
                str(tcp_src_port) + " " + \
                str(ip_dest) + ":" + \
                str(tcp_dest_port) + " HTTP "

    # Check if data is a HTTP Response or HTTP Request
    if ("GET" or "HEAD" or "POST" or "PUT" or "DELETE" or "CONNECT" or "OPTIONS" or "TRACE") in sliced_data[0]:
        HTTP_line += "Request"
    else:
        HTTP_line += "Response"
    print(HTTP_line)

    # Iterate through sliced data and print out header lines
    line = 0
    while sliced_data[line]:
        print(sliced_data[line])
        line += 1

    # Print newline
    print()
```

- In this code block, I split the string containing the HTTP response/request into headers by using the “\\r\\n” keys. I am left with a list of strings containing the individual headers.
- I then built the HTTP response/request line in the required format.
- I check to see if the data string is an HTTP response or request by checking it it contains an HTTP method or “HTTP”, then add the corresponding string to HTTP_line.
- Finally, I iterate through the rest of the headers in sliced_data and print them until I reach the end of the list or there is an empty string.