Jackson Tran
ID: 2021848238

Project 1: HTTP Header Sniffer

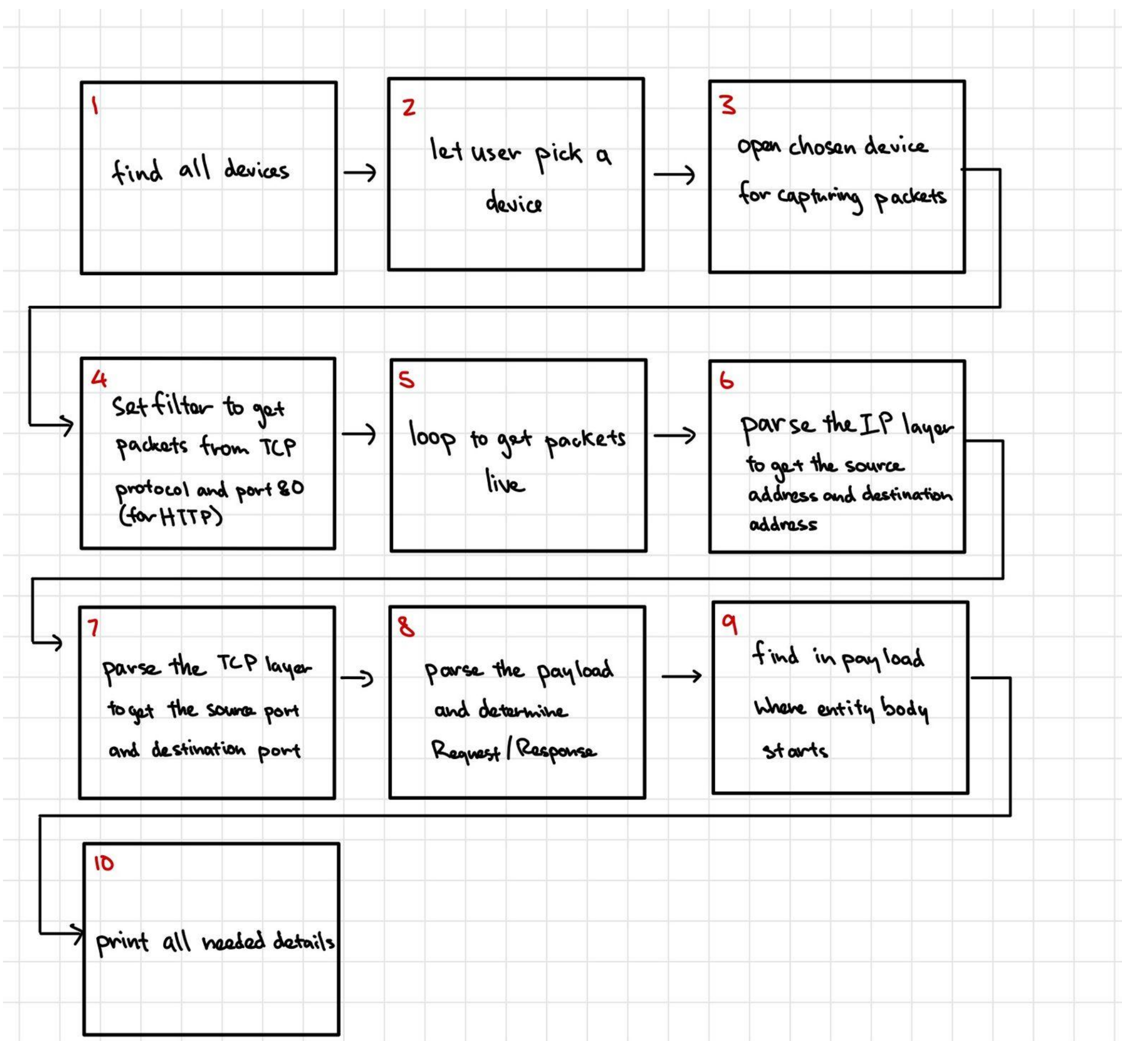
Introduction/Reference:

This program was written in text editor on Virtual Box and ran on Ubuntu 20.04.3. The program was also written in Python (Python 3.8.10) on Text Editor. References used

- Used tcpdump.com to understand the functions from the pcapy library.
- https://docs.python.org/3/library/stdtypes.html for converting bytes to int
- https://en.wikipedia.org/wiki/Ethernet_frame to look at ethernet frame
- https://www.w3.org/People/Frystyk/thesis/TcpIp.html to understand how to parse IP and TCP layer

This program is a simple HTTP header sniffer which prints the headers of Request and Response packets.

Flow chart or Diagram/Logical explanations:



Jackson Tran
ID: 2021848238

1. Find all devices

```
145 def main(argv):
146     # obtain the list of available network device
147     devices = pcapy.findalldevs()
148     if devices is None:
149         print("No devices are found")
150         sys.exit()
151
```

On line 147, from the pcapy library, the program uses findalldevs() which returns a list of devices available for capturing. Error check in case no devices are returned (line 148-150).

2. Let user pick a device

```
151
152     # prints list of devices
153     print("list of devices:")
154     for i in range(len(devices)):
155         print("{}. {}".format(i + 1, devices[i]))
156
157     # user selects the device
158     while True:
159         device = input("Select a device: ")
160         if device in devices:
161             break
162         else:
163             print(" '{}' is an invalid device. Select a valid device from the list".format(device))
164             continue
165     print("User selected device: {}".format(device))
```

Line 153-155 program prints all the returned devices as a result of findalldevs(). The program then loops on line 158 to get user input of the device they want to select. If a valid device is chosen (or the device is found in the list of devices), then we can break from the loop (line 160-161). If the user inputs an invalid device (or the device they input is not found in the list of devices) then the loop continues until the user chooses a valid device (line 162-164).

3. Open chosen device for capturing packets

```
167
168     # obtain packet capture handle from device
169     # maximum size of IP packet is 65,535
170     p = pcapy.open_live(device, 65535, 1, 1000)
171
```

Using the pcapy library, use pcapy.open_live() to open the chosen device (line 170). Snapshot length is set to 65535 because its maximum size of IP packet is 65,535. Set Promiscuous mode on because we want to be able to sniff the network. Set buffer timeout to 1000 ms as recommended by tcpdump to set a non-zero buffer timeout.

4. Set filter

```
171
172     # set the filter
173     if p.setfilter("tcp port 80") != 0:
174         pcapy.PcapError(p)
175
```

Use p.setfilter() with the expression “tcp port 80” to filter the packets we sniff (line 173). If an error is returned from the setting filter, prompt the error with an error message in p (line 174).

Jackson Tran
ID: 2021848238

5. Loop packets to get packet live

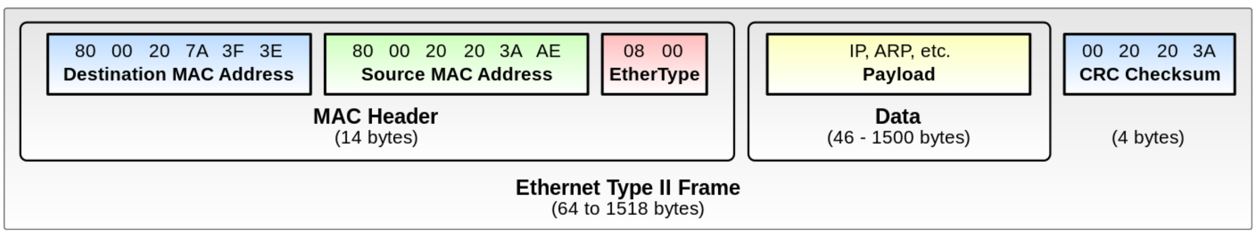
```
175
176     # starts the loop
177     p.loop(-1, packet_handling);
178
```

Uses p.loop() to loop *n* amount of packets using pack_handling() to deal with each packet (line 177). Set packet count to -1 to indicate infinity, so that the packets are processed until the program ends the condition.

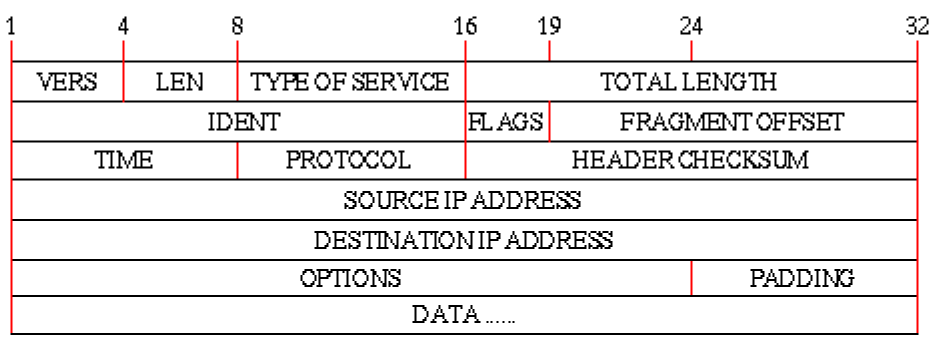
```
'''
Handles each packets that we receive. Parses through each layer
to get the source port/address and destination port/address. Exits (don't print) if the header is incomplete.
Prints the packet if we get a response/request through identify()
'''
def packet_handling(header, packet):
```

Loop() sends the live packet capture to packet_handling() where it processes the packet (where parsing addresses, port number, etc. happens).

6. Parse the IP layer to get the source address and destination address



The ethernet frame consists of 14 bytes (containing destination/source MAC and ether type) as displayed above from https://en.wikipedia.org/wiki/Ethernet_frame. We don't need to parse anything within the Ethernet frame for this project. However, we need the length of the ethernet layer in order to find the next index/start of the IP layer. We also know that the IP layer is at least 20 bytes long as shown in the image below from https://www.w3.org/People/Frystyk/thesis/TcpIp.html.



From version to destination IP address is 20 bytes long.

Jackson Tran
ID: 2021848238

```
22 def packet_handling(header, packet):
23 
24     # Parse ethernet frame (first 14 bytes)
25     ethernet_len = 14
26 
27     # unpacking for Ethernet (not needed)
28     #   destination MAC (6 bytes)
29     #   source MAC (6 bytes)
30     #   ether type (2 bytes)
31 
32     # get first 20 known bytes for ip header
33     ip = packet[ethernet_len: 20 + ethernet_len]
34 
35     # unpacking for IP
36     #   version/header length (1 byte)  -> ip[0]
37     #   type of service (1 byte)        -> ip[1]
38     #   total length (2 bytes)          -> ip[2:4]
39     #   identification (2 bytes)        -> ip[4:6]
40     #   fragment offset (2 bytes)       -> ip[6:8]
41     #   time to live (1 byte)           -> ip[8]
42     #   protocol (1 byte)               -> ip[9]
43     #   header checksum (2 bytes)       -> ip[10:12]
44     #   source address (4 bytes)        -> ip[12:16]
45     #   destination address (4 bytes)   -> ip[16:20]
46 
47 
48     # header length is last 4 bits, so mask only last 4
49     ip_HL = ip[0] & 0xF
50 
51     # length of IP header
52     ip_len = ip_HL * 4
53 
54     # in case the ip has invalid size, we exit
55     if ip_len < 20:
56         return
57 
58     # get the source and destination address
59     source_address = convert_address(ip[12:16])
60     destination_address = convert_address(ip[16:20])
```

Therefore, in line 33 the ip layer is held within at least 20 bytes after ethernet_len. Given the number of bytes of each detail in the IP frame, we can get the IP header length, source address and destination address. In order to get the IP header length, we first get the byte that it is located at (ip[0]). Since version and header length both share the same byte we need to do bit masking to get the bits we want. Header length is the last 4 bits of the byte so we can simply mask it with 0xF in hexadecimal or 00001111 in binary. We then need to multiply the results by 4 to get the actual size of the IP header. We want to find the length in case there is other info from IP Options which is after 20 bytes. We also want to exit in case the length is smaller than 20. To get source address and destination address, we can access them through indexing the last 8 bytes (line 59-60). Because they are still in bytes form, we need to convert them to the correct readable format. The program converts them through the convert_address() function shown below.

```

# takes in address in bytes form and convert to correct address format
def convert_address(address):
    # decodes each byte and join them by '.'
    converted = '.'.join(str(b) for b in address)
    return(converted)

```

The function essentially goes through each byte of an address and converts each byte to the corresponding string. Then the converted bytes are joined by ‘.’.

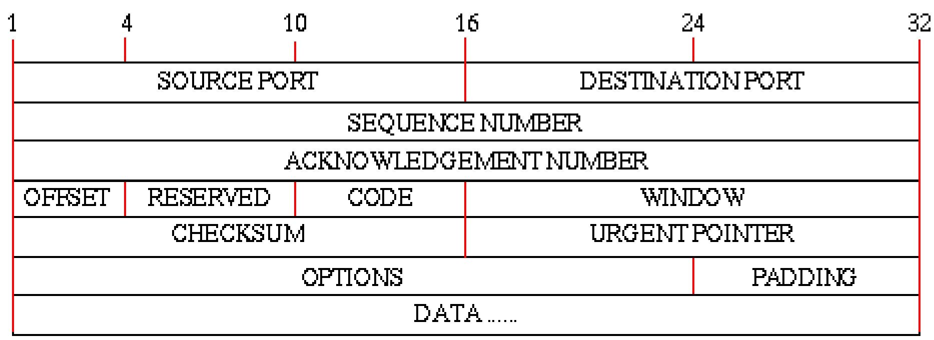
Jackson Tran
ID: 2021848238

7. Parse the TCP layer to get the source port and destination port

```
    # start index for tcp
    index_tcp = ethernet_len + ip_len
    tcp = packet[index_tcp:index_tcp + 20]

```

We found the ip header length earlier in the IP layer. We can then find where the TCP layer starts by adding ethernet_len and ip_len. Similar to the IP layer, we also know that the TCP layer is at least 20 bytes long as shown in the image below from https://www.w3.org/People/Frystyk/thesis/TcpIp.html.



From source port to urgent pointer is 20 bytes long. Therefore, we access 20 bytes after the IP layer.

```
65
66      # unpacking for TCP:
67      #   source port (2 bytes)                -> tcp[0:2]
68      #   destination port (2 bytes)           -> tcp[2:4]
69      #   sequence number (4 bytes)            -> tcp[4:8]
70      #   acknowledgement number (4 bytes)     -> tcp[8:12]
71      #   offset/reserved (1 byte)             -> tcp[12]
72      #   TCP Flags (1 byte)                   -> tcp[13]
73      #   Window (2 bytes)                     -> tcp[14:16]
74      #   checksum (2 bytes)                   -> tcp[16:18]
75      #   urgent pointer (2 bytes)             -> tcp[18:20]
76
77      # get source/destination port
78      source_port = int.from_bytes(tcp[0:2], "big")
79      destination_port = int.from_bytes(tcp[2:4], "big")
80
81      # offset is first 4 bits, we don't need reserved
82      tcp_offset = tcp[12] >> 4
83      tcp_len = tcp_offset * 4
84
85      # in case tcp has invalid size
86      if tcp_len < 20:
87          return
88
```

Given where and how many bytes is source port and destination port, we can simply access them from tcp[0:2] for source port and tcp[2:4] for destination port (line 78-79). To convert them into integers, the program uses int.from_bytes() where “big” indicates that the most significant bits are at the beginning of the byte. Again, we need the TCP offset inorder to determine the length of TCP header and also to determine where payload will start. The program can access offset from tcp[12] (line 82). Since offset and reserved both share the byte, we need to manipulate the bits in order to only get the offset. By doing, bit right shift 4 times, we can get the bits for only the offset. Similar to IP header length we multiply the offset by 4 to get the actual

Jackson Tran
ID: 2021848238

size of the TCP and the offset to the payload data. In case the TCP length is less than 20, we want to exit for an incomplete TCP header length (line 86).

8. Parse the payload and determine Request/Response

```
88
89      #start index for payload
90      index_payload = ethernet_len + ip_len + tcp_len
91
92      # get size of payload
93      payload_size = len(packet) - index_payload
94      payload = packet[index_payload:]
95
96      # we only want to print packets with payload
97      if payload_size > 0:
98          # identifies what kind of packet (response or request)
99          # also returns empty string for anything else
100         identification = identify(payload)
```

The payload starts after TCP, so we need to add ethernet_len, ip_len, and tcp_len to find the index of where the payload starts in the packet (line 90). We can simply calculate the payload size by subtracting the size of the packet by the length from ethernet to TCP (line 93). We can access the payload from the index of where payload starts in the packet till the end of the packet (line 94). In case the payload size is 0, we don't need the packet. To identify if the packet is a response or a request, it sends the payload to the function identify() (line 100).

```
116
117  #identifies the packet by checking the first line
118  # if the first 4 bytes is 'HTTP' then we have response
119  # if 'HTTP' appears anywhere in the first line thats not the first 4 bytes then we know its a request
120  # anything else means that the packet is not needed
121  def identify(payload):
122      # response if first 4 bytes is 'HTTP'
123      if bytes('HTTP','UTF-8') in payload[:4]:
124          return "Response"
125
126      for i in range(len(payload)):
127          # only checks the first line for 'HTTP'
128          if bytes("\r\n",'UTF-8') == payload[i:i + 2]:
129              # request if 'HTTP' appears in the first line but not first 4 bytes
130              if bytes('HTTP','UTF-8') in payload[:i]:
131                  return "Request"
132              else:
133                  break
134      # returns empty string to indicate useless header packets
135      return ""
136
```

If the first 4 bytes hold "HTTP" then it's a response (line 123). Otherwise, we have to go through the first line of the payload until it hits a new line (line 126 - 133). If "HTTP" appears anywhere on the first line that's not the first 4 bytes, then we know that it's a request (line 130). If "HTTP" does not appear in the first line, then it's not a request or a response. Then, we return an empty string in which it indicates that we don't need this packet (line 135).

9/10. Find in payload where entity body starts/print all needed details

```
13
14  # keep count of the number of request/response
15  count = 0
```

Count is to keep count of number of successful request/response packet prints

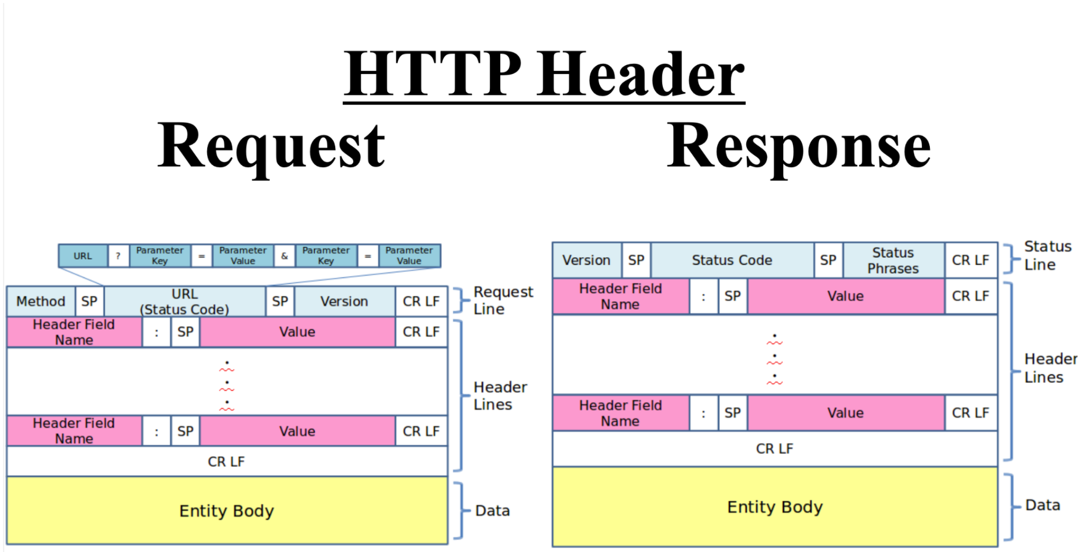
Jackson Tran
ID: 2021848238

```
101
102         # only print packets that are response or request
103         if identification != "":
104             # increase count of successful packet prints
105             global count
106             count += 1
107             print("{} {}:{} {}:{} HTTP {}\r\n{}\r\n\r\n".format(count, source_address, source_port, destination_address,
108             destination_port, identification, decode_print(payload)))
109     return
```

If the identification is empty as returned from identify(), then we know that we don't need the packet since we don't know if it's a request or response (line 103). The count is only incremented if we encounter a successful packet (line 105-106). The program determines when to stop printing payload before the entity body in decode_print() (line 108).

```
137
138 # takes in identified payload an sets up a string for printing
139 def decode_print(payload):
140     for i in range(len(payload)):
141         # decode anything thats before '\r\n\r\n' or the entity body
142         if "\r\n\r\n" == str(payload[i:i + 4], 'UTF-8'):
143             return str(payload[:i], 'UTF-8')
144
```

The entity body starts when two newlines appear after the header lines. We know this from how the request/response payload is structured, as shown below.



The function basically goes through the payload by each index until it hits "\r\n\r\n" which indicates where the entity body starts (line 140-142). At that point, the function returns the string form after decoding the request lines and header lines (line 143).

Lastly, in line 107, the program prints after getting all the necessary information: count, source address, source port, destination address, destination port, identification (request/response), decoded_print (request line & header lines).