PROJECT TITLE : PACKET SNIFFER

DONE BY

GROUP 9

BATCHU.TARUN –AP18110010517
MEKALA VENU GOPAL – AP18110010494
K. REVANTH KUMAR – AP18110010077
SHOBITH GUPTHA – AP18110010520

DESCRIPTION:

Packet sniffer is an application or program that performs packet sniffing and Packet sniffing is a process or technique of monitoring and capturing all data packets passing through a software, network, hardware devices. Many system administrator or network administrator use it for monitoring and troubleshooting network traffic. Packet sniffers are useful for both wired and wireless networks or we can say that, It works on both switched environment and non-switch environment. It works by intercepting traffic data as it passes over the wired or wireless network and copying it to a file. It is called as packet capture.

In other words, Sniffing allows you to see all sorts of traffic, both protected and unprotected. In the right conditions and with the right protocols in place, an attacking party may be able to gather the information that can be used for further attacks or to cause other issues for the network or system owner.

What can be sniffed?

One can sniff the following sensitive information from a network −

- Email traffic
- FTP passwords
- Web traffics
- Telnet passwords
- Router configuration
- Chat sessions
- DNS traffic

Types of Sniffing:

Sniffing can be either Active or Passive. We will now learn about the different types of sniffing.

Passive Sniffing:

In passive sniffing, the traffic is locked but it is not altered in any way. Passive sniffing allows listening only. It works with the Hub devices. On a hub device, the traffic is sent to all the ports. In a network that uses hubs to connect systems, all hosts on the network can see the traffic. Therefore, an attacker can easily capture traffic going through.

The good news is that hubs have almost become obsolete in recent times. Most modern networks use switches. Hence, passive sniffing is no more effective.

Active Sniffing:

Inactive sniffing, the traffic is not only locked and monitored but it may also be altered in some way as determined by the attack. Active sniffing is used to sniff a switch-based network. It involves injecting address resolution packets (ARP) into a target network to flood on the switch content addressable memory (CAM) table. CAM keeps track of which host is connected to which port.

Following are the Active Sniffing Techniques −

- MAC Flooding
- DHCP Attacks
- DNS Poisoning
- Spoofing Attacks
- ARP Poisoning

The Sniffing Effects on Protocols:

Protocols such as the tried and true TCP/IP were never designed with security in mind. Such protocols do not offer much resistance to potential intruders. Following are the different protocols that lend themselves to easy sniffing −

HTTP

It is used to send information in clear text without any encryption and thus a real target.

SMTP (Simple Mail Transfer Protocol)

SMTP is utilized in the transfer of emails. This protocol is efficient, but it does not include any protection against sniffing.

NNTP (Network News Transfer Protocol)

It is used for all types of communication. A major drawback of this is that data and even passwords are sent over the network as clear text.

POP (Post Office Protocol)

POP is strictly used to receive emails from the servers. This protocol does not include protection against sniffing because it can be trapped.

FTP (File Transfer Protocol)

FTP is used to send and receive files, but it does not offer any security features. All the data is sent as clear text that can be easily sniffed.

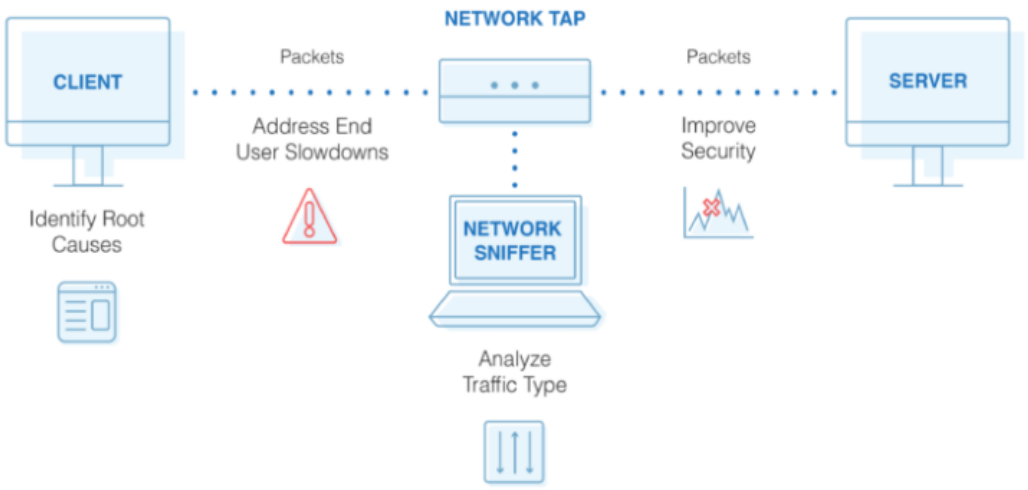
IMAP (Internet Message Access Protocol)

IMAP is the same as SMTP in its functions, but it is highly vulnerable to sniffing.

Telnet

Telnet sends everything (usernames, passwords, keystrokes) over the network as clear text and hence, it can be easily sniffed.

Sniffers are not the dumb utilities that allow you to view only live traffic. If you want to analyze each packet, save the capture and review it whenever time allows.



The proposed project is implemented in python programming language, and using this application or program admin of the system can capture network packets and analyze data received/sent from/to the network.

Developed as a program, packet sniffer facilitates web-based monitoring of network packets that are traveling over the system network. The primary data captured by this program is the packets source and destination address data and many other can be captured.

Approach:

Our main aim was to see the data flow in a network but in a network the data will be in machine readable so initially by using a function we will make it into human readable code and our program will keep on running and capturing all the packets in a particular network flowing while the program was in running state and when a packet is caught first it will unpack that packet in a particular format and return the ethernet prototype, destination mac address, source mac address and data and if the ethernet port number is 8 it means it belongs to ipv4 and then that ipv4 will return the version, header, TTL, prototype, source address, destination address and in that if the prototype is numbered in below specified protocols it will go to their specified functions to return their required information like ICMP returns ICMP type and header length, Time to live, prototype, source address same UDP returns source port, destination port, size, data similar for all the protocols.

Protocol		
IP Protocol ID INCLUDING (but not limited to):		
1 ICMP	17 UDP	57 SKIP
2. IGMP	47 GRE	88 EIGRP
6 TCP	50 ESP	89 OSPF
9 IGRP	51 AH	115 L2TP

Algorithm:

Step 1: Establish a socket connection and initialize it to connection

Step 2: See for packets and keep on capturing them

Step 3: Store the data into raw_data, and address into addr

Step 4: Pass the raw_data into the function named ethernet_frame

Step 5: the function ethernet_frame function will unpack the raw_data and divide the 1st six characters and assign it to dest_mac and 2nd 6 characters to src_mac and ethernet type as H

And then return dest_mac passing into get_mac_addr function and src_mac passing into get_mac_addr function and proto to converting to man read format and data from 14th character to last

Step 6: the function named get_mac_addr will take the mac address of source and destination which in is byte format and returns the mac address which man can read

Step 7: Now the destination address and source address will be printed

Step 8: if the eth_proto is equal to 8 then go to step 9 else go to step 16

Step 9: pass the data to the ipv4_packets function

Step 10: initialize the 1st character of the data to version_header_length and initialize the version to the 4 shifts to version_header_length and header_length to 4 times of the & operator of version_heder_length and 15

Step 11: unpack the data in format 8x B B 2X 4s 4s bypassing the 1st 20 characters of the data and return version header_length, TTL, proto, src, pass src to ipv4 function and target to step 9 and data up to header_length

Step 12: if proto equal to 1 then go to step 13, if proto equal to 6 then go to the step 14, if proto equal to 17 then go to step 15 else go to step 16

Step 13: pass the data into the icmp_packet function

Step 13.1: icmp_packet function will take the data up to the 4th character and unpack them in format B B H and initialize them to icmp_type, code, checksum and return those

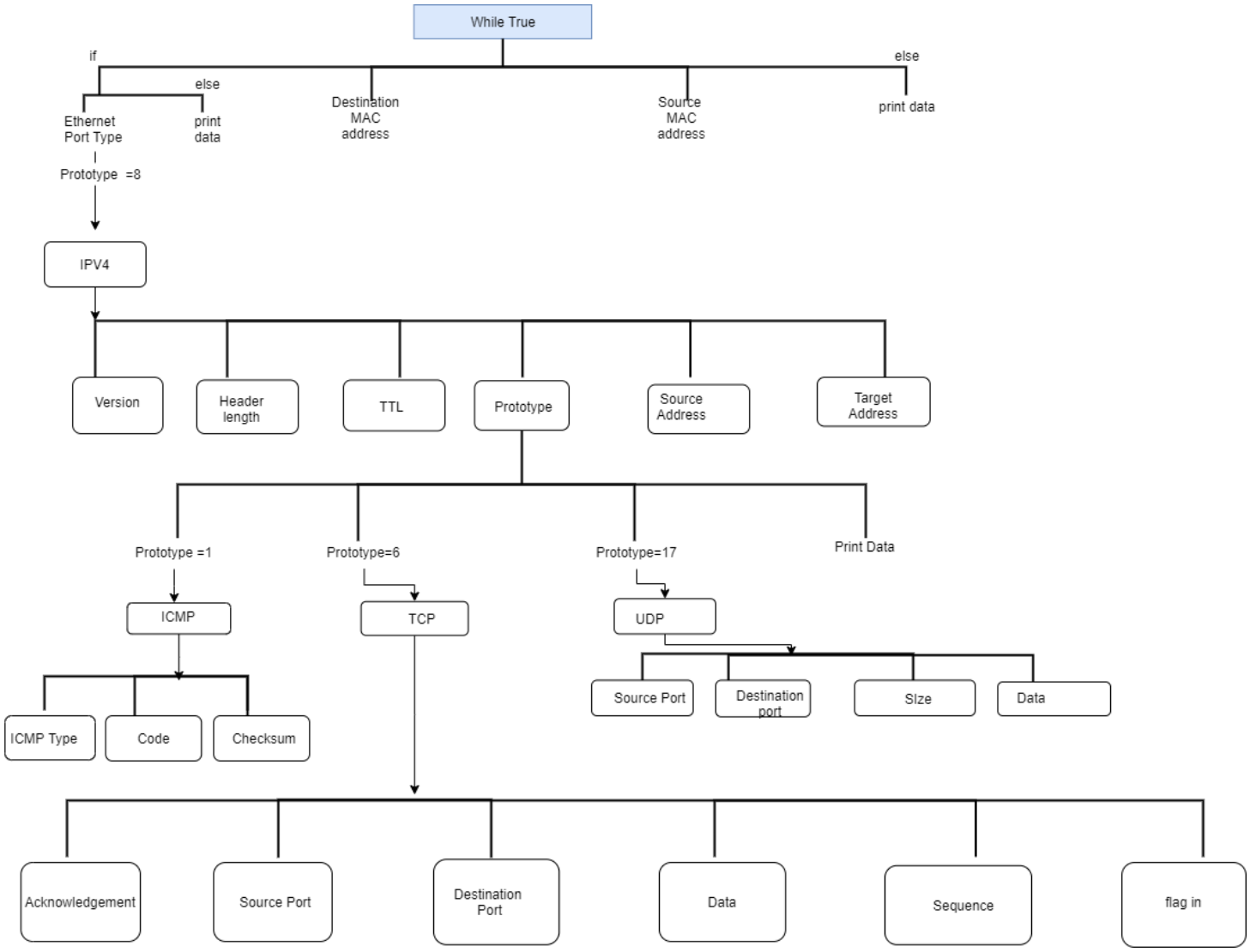
Step 14: pass the data into the format_multi_line function which will return correctly formatted data

Step 15: pass the data into the udp_segment function

Step 15.1: that function will unpack the data upto 8 characters in the format H H 2X H and return src_port, dest_port, size, data

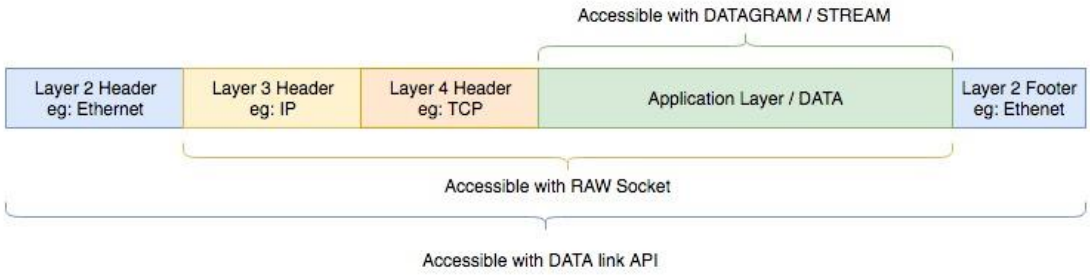
Step 16 : print data.

FLOW CHART:


While True
if
else
else
Ethernet Port Type
print data
Destination MAC address
Source MAC address
print data
Prototype =8
IPV4
Version
Header length
TTL
Prototype
Source Address
Target Address
Prototype =1
Prototype=6
Prototype=17
Print Data
ICMP
TCP
UDP
ICMP Type
Code
Checksum
Source Port
Destination port
SIze
Data
Acknowledgement
Source Port
Destination Port
Data
Sequence
flag in

IMPLEMENTATION DETAILS:

With help of Socket module, Packet sniffer can be created . Raw socket type is used to get the packets . A raw socket provides access to the underlying protocols, which support socket abstractions. Generally raw socket is the part of Internet Socket API , which were used to generate and receive IP packets.



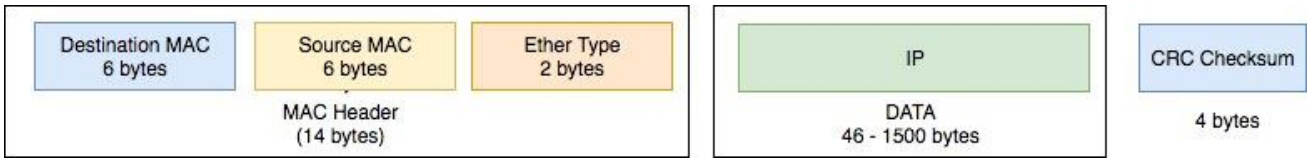
SYSTEM REQUIREMENTS:

We need to use a Linux OS to run this script. For Windows or macOS Virtual Linux environment should be installed . Also, most operating systems require root access to use raw socket APIs.

Parsing the Packet

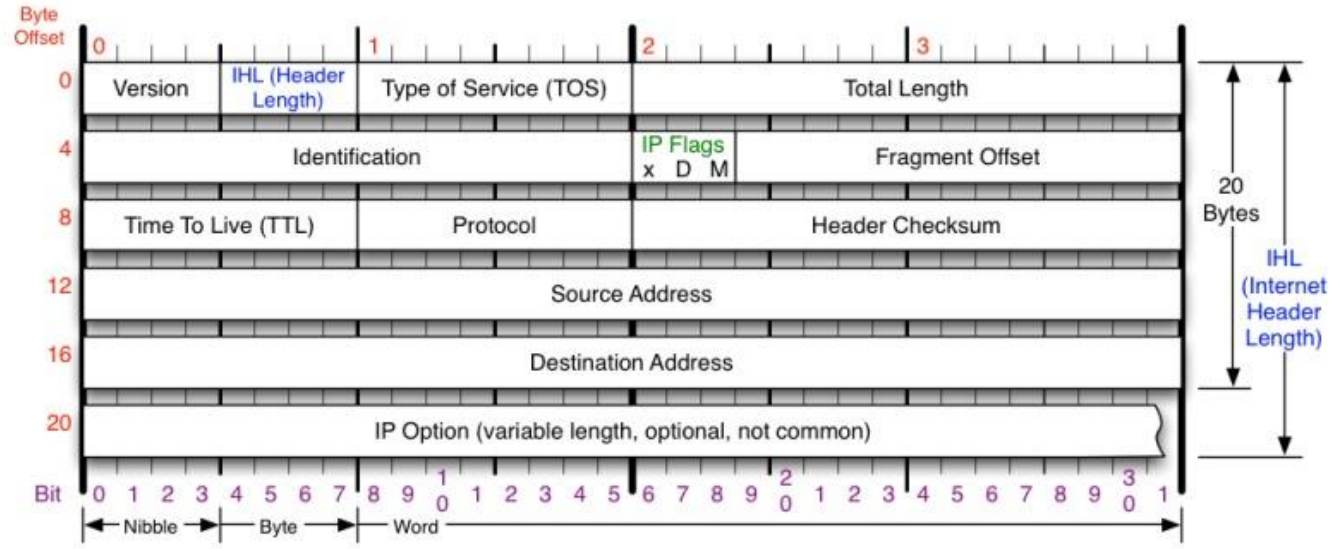
To parse the packet, we need to have a idea about Ethernet Frame and the packet headers of the IP.

The Ethernet Frame structure look like this



From the above mentioned structure First 6 bytes for **Destination MAC** and the next 6 bytes for the **Source MAC** and the last 2 bytes for the **Ether type** . The rest of the bytes includes **DATA** and **CRC Checksum .**

According to RFC 791, an IP header looks like the following:



The IP header includes the following sections:

• **Protocol Version (four bits):** The first four bits. This represents the current IP protocol.

• **Header Length (four bits):** The length of the IP header is represented in 32-bit words. Since this field is four bits, the maximum header length allowed is 60 bytes. Usually the value is 5, which means five 32-bit words: 5 * 4 = 20 bytes.

• **Type of Service (eight bits):** The first three bits are precedence bits, the next four bits represent the type of service, and the last bit is left unused.

• **Total Length (16 bits):** This represents the total IP datagram length in bytes. This a 16-bit field. The maximum size of the IP datagram is 65,535 bytes.

• **Flags (three bits):** The second bit represents the Don't Fragment bit. When this bit is set, the IP datagram is never fragmented. The third bit represents the More Fragment bit. If this bit is set, then it represents a fragmented IP datagram that has more fragments after it.

• **Time To Live (eight bits):** This value represents the number of hops that the IP datagram will go through before being discarded.

• **Protocol (eight bits):** This represents the transport layer protocol that handed over data to the IP layer.

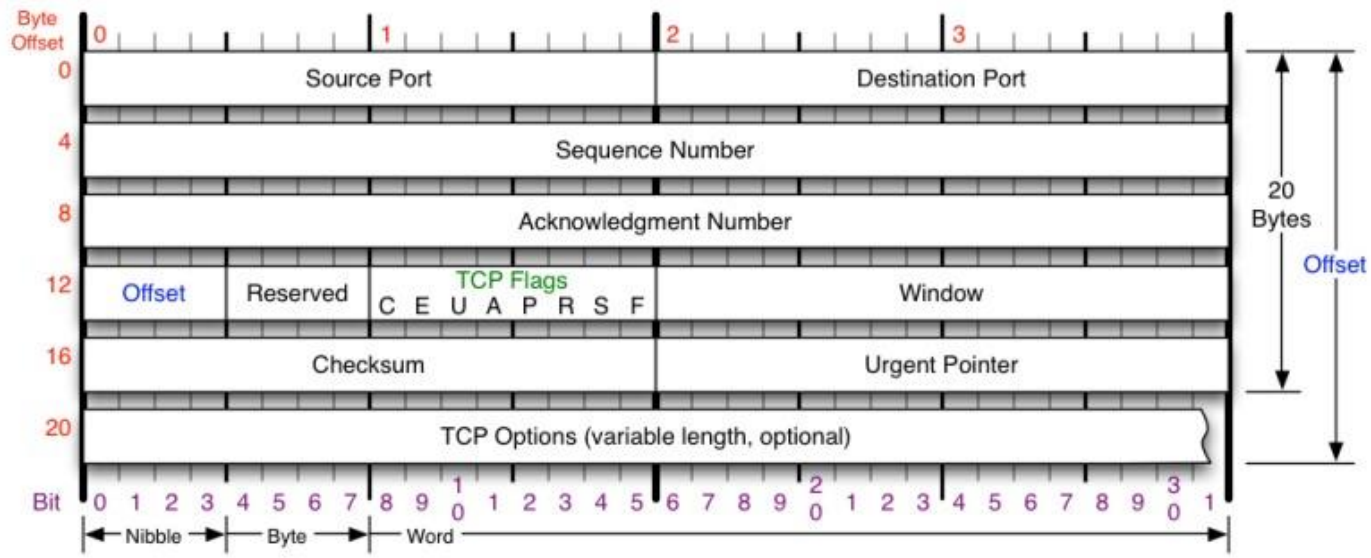
• **Header Checksum (16 bits):** This field helps to check the integrity of an IP datagram.

• **Source and destination IP (32 bits each):** These fields store the source and destination address, respectively.

Refer to the RFC 791 document for more details on IP headers: [RFC 791 - Internet Protocol (ietf.org)]

On going with the process after unpacking the Internet Layer(the steps will be mentioned in source code). We need to unpack the Transport layer. We can determine the protocol from the protocol ID in the IP header. One of the protocol ID is **TCP.** For that a function need to be created for unpacking the TCP packets(will be mentioned in the source code)

The TCP packets are unpacked according to the TCP packet header's structure:



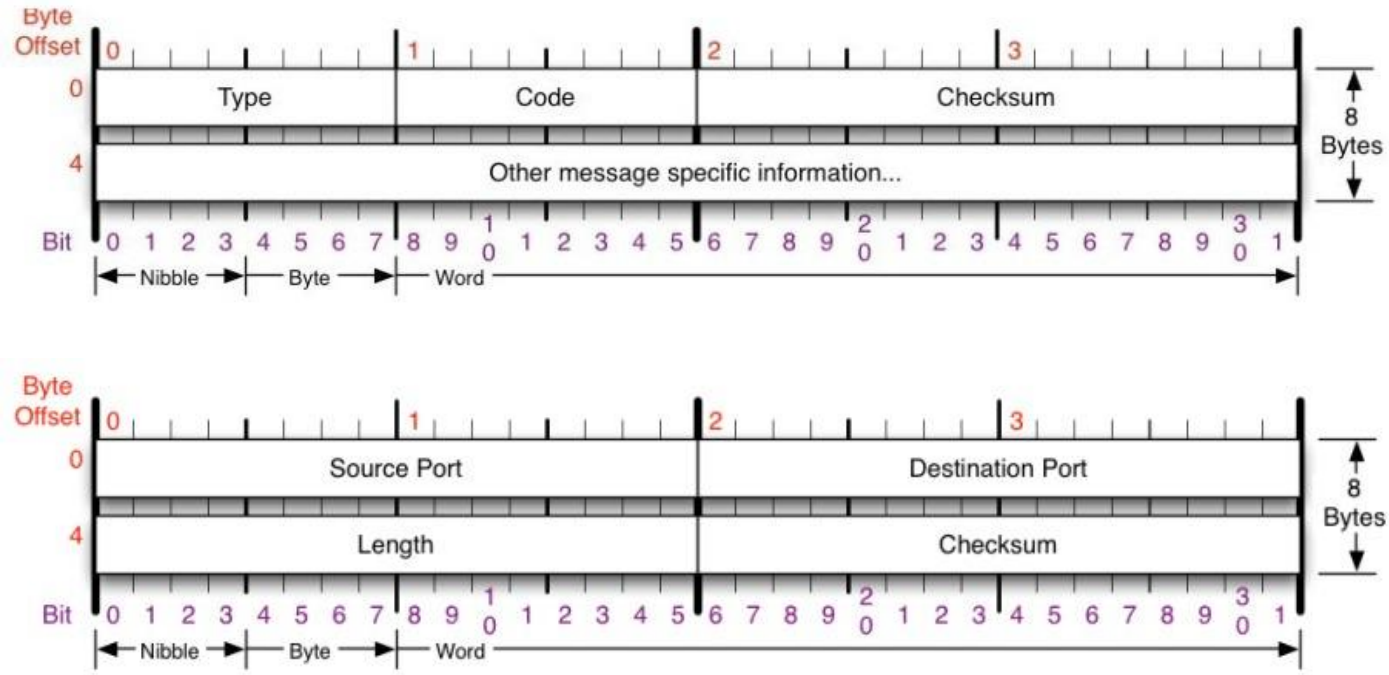
TCP packet header’s structure include the following sections:

- **Source Port:** 16 Bit number which identifies the Source Port Number (Sending Computer's TCP Port).
- **Destination Port:** 16 Bit number which identifies the Destination Port number (Receiving Port).
- **Sequence Number:** 32 Bit number used for byte level numbering of TCP segments. If you are using TCP, each byte of data is assigned a sequence number.
- **Acknowledgment Number:** 32 Bit number field which indicates the next sequence number that the sending device is expecting from the other device.
- **Checksum** : The 16-bit checksum field is used for error-checking of the header and data.
- **Window** : Indicates the size of the receive window, which specifies the number of bytes beyond the sequence number in the acknowledgment field that the receiver is currently willing to receive.

- **<u>Urgent Pointer</u>** : Shows the end of the urgent data so that interrupted data streams can continue. When the URG bit is set, the data is given priority over other data streams (Size 16 bits).

TCP FLAGS : C E U A P R S F

Similarly for ICMP AND UDP ,Head Structures are mentioned below



The Unpacking functions of ICMP and UDP are also mentioned in the below source code

Source Code :

```
import socket

import struct

import textwrap



# To beautify

TAB_1 = '\t - '

TAB_2 = '\t\t - '

TAB_3 = '\t\t\t - '
```

```
TAB_4 = '\t\t\t\t - '


DATA_TAB_1 = '\t '

DATA_TAB_2 = '\t\t '

DATA_TAB_3 = '\t\t\t '

DATA_TAB_4 = '\t\t\t\t '


#socket connection

#last argument is it make sure that it compatable for all mechines

def main():

        #The AF_PACKET socket in Linux allows an application to receive and send raw packets

    #AF_PACKET ---> is a address family constant for Low level packet interface

    #A raw socket is a type of socket that allows access to the underlying transport provider

    #SOCK_RAW ----> Raw socket

    #The ntohs() function translates a short integer from network byte order to host byte order so that human can read

    conn = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))

    print(TAB_1)

    print("================START================")

    print(TAB_1)


#runs forever

#looping forever and listening for packets when ever we see a packet take it and extract the information from it

    while True:

    #we were taking socket and when ever we see data we take it and store it in raw_data and address

    # the parameter passed into the method is buffer size

    #65565 is the mx buffer size

        raw_data, addr = conn.recvfrom(6553)


# program Enough upto here to just see a raw packets

# But human cant read or understand
```

```
        #now we pass that data to ethernet_frame function
        dest_mac, src_mac, eth_proto, data = ethernet_frame(raw_data)
        print('\nEthernet Frame:')
        print(TAB_1 + 'Destination: {}, Protocal: {}'.format(dest_mac, src_mac, eth_proto))

        # 8 is for IPV4
        if eth_proto == 8:
            (version, header_length, ttl, proto, src, target, data) = ipv4_packet(data)
            print(TAB_1 + 'IPV4 Packet:')
            print(TAB_2 + 'version: {}, Header Length: {}, TTL: {}'.format(version, header_length, ttl))
            print(TAB_2 + 'Protocal: {}, Source: {}, Target: {}'.format(proto, src, target))

                # 1 if for ICMP
                    if  proto == 1:
                        icmp_type, code, checksum, data = icmp_packet(data)
                        print(TAB_1 + 'ICMP Packet:')
                        print(TAB_2 + 'Type: {}, Code: {}, Checksum: {},'.format(icmp_type,
code, checksum))
                        print(TAB_2 + 'Data:')
                        print(format_multi_line(DATA_TAB_3, data))

                # 6 is for TCP
                    elif proto == 6:
                        (src_prot, dest_port, sequence, acknowledgment, flag_urg, flag_ack,
flag_psh, flag_rst, flag_syn, flag_fin)
                        print(TAB_1 + 'TCP Segment:')
                        print(TAB_2 + 'Source Port: {}, Destination Port: {}'.format(src_port,
dest_port))
                        print(TAB_2 + 'Sequence: {}, acknowledgment'.format(sequence,
acknowledgment))
                        print(TAB_2 + 'Flags:')
                        print(TAB_3 + 'URG: {}, ACK: {}, PSH: {}, RST: {}, SYN: {}, FIN:
{}'.format(flag_urg, flag_ack, flag_psh, flag_rst, flag_syn, flag_fin))
                        print(TAB_2 + 'Data:')
                        print(format_multi_line(DATA_TAB_3, data))
```

```
            # 17 is for UDP
                elif proto == 17:
                    src_port, dest_port, length, data = udp_segment(data)
                    print(TAB_1 + 'UDP Segment:')
                    print(TAB_2 + 'Source Port: {}, Destination Port: {}'.format(src_port,
dest_port))

            # for all other
                else:
                    print(TAB_1 + 'Data:')
                    print(format_multi_line(DATA_TAB_2, data))
            else:
                print('Data:')
                print(format_multi_line(DATA_TAB_1, data))



# unpacking the ethernet frame
# Here when ever we see the o's and 1's going across the network we pass it in to this function
# then this function unpackes that frame and find what are those 1's & 0's
#this returns 4 different things 1>destination 2>source 3>ethernet type and 4>actual payload
#6s and 6s is destination and source MAC address of 1st 6 bytes and H is last unsigned short for
ether type
def ethernet_frame(data):
    dest_mac, src_mac, proto = struct.unpack('! 6s 6s H', data[:14])
    return get_mac_addr(dest_mac), get_mac_addr(src_mac), socket.htons(proto), data[14:]

#This function will return the formated MAC address
#some thing which looks like 00:11:22:33:44:55
def get_mac_addr(bytes_addr):
    bytes_str = map('{:02x}'.format, bytes_addr)
    return ':'.join(bytes_str).upper()
```

```
# now lets unpack those IPV4 packets
# >> is shift to 4 characthers
def ipv4_packets(data):
        version_header_length = data[0]
        version = version_header_length >> 4
        #checks the length of the version_header_length is true or
        header_length = (version_header_length & 15) * 4
        ttl, proto, src, traget = struct.unpack('! 8x B B 2x 4s 4s', data[:20])
        return version, header_length, ttl, proto, src, ipv4(src), ipv4(target), data[header_length:]


#returns properly formated IPV4 address
def ipv4(addr):
        # to return some thing like 127.234.367.1.9
        return '.'.join(map(str, addr))


# unpacks ICMP (Internet Control Message Comtrol Protocal) packets
def icmp_packets(data):
        icmp_type, code, checksum = struct.unpack('! B B H', data[:4])
        return icmp_type, code, checksum, data[4:]

#unpack TCP
# mostly most of the packets across the network were TCP like facebook insta etc...
def tcp_segment(data):
        (src_port, dest_port, sequence, acknowledgment, offset_reserved_flags) = struct.unpack('! H H L L H', data[:14])
        offset = (offset_reserved_flags >> 12) * 4
        flag_urg = (offset_reserved_flages & 32) >> 5
        flag_ack = (offset_reserved_flages & 16) >> 4
        flag_psh = (offset_reserved_flages & 8) >> 3
        flag_rst = (offset_reserved_flages & 4) >> 2
        flag_syn = (offset_reserved_flages & 2) >> 1
```

```
        flag_fin = offset_reserved_flages & 1
        return src_port, dest_port, sequence, acknowledgment, flag_fin, data[offset:]


#this function unpacks the udp packets
def udp_segment(data):
        src_port, dest_port, size = struct.unpack('! H H 2x H', data[:8])
        return src_port, dest_port, size, data[8:]


#this function is to formate the multile line data
#in some cases we come across a large like 2000 10000 lines data then this function helps to break it line by line
def format_multi_line(prefix, string, size=80):
        size -= len(prefix)
        if isinstance(string, bytes):
                string = ''.join(r'\x{:02x}'.format(bytes) for bytes in string)
                if size % 2:
                        size -= 1
        return '\n'.join([prefix + line for line in textwrap.wrap(string, size)])


main()
```

Now save and run the script with the required permission

The output will print all the packets that were sniffed. So, it will continue printing until we stop it with a keyboard interrupt. The output will be as follows

OUTPUT:

```
┌──(venugopal㉿kali)-[~/Downloads]
└─$ sudo python3 packetsniffer.py
[sudo] password for venugopal:
-------------Packet Start-------------

Ethernet Frame:
        - Destination: 52:54:00:12:35:02, Protocal: 08:00:27:6A:A6:D6
        - IPV4 Packet:
                - version: 4, Header Length: 20, TTL: 64
                - Protocal: 6, Source: b'\n\x00\x02\x0f', Target: 10.0.2.15
-------------Packet Start-------------

Ethernet Frame:
        - Destination: 08:00:27:6A:A6:D6, Protocal: 52:54:00:12:35:02
        - IPV4 Packet:
                - version: 4, Header Length: 20, TTL: 64
                - Protocal: 6, Source: b'u\x12\xed\x1d', Target: 117.18.237.29
-------------Packet Start-------------

Ethernet Frame:
        - Destination: 52:54:00:12:35:02, Protocal: 08:00:27:6A:A6:D6
        - IPV4 Packet:
                - version: 4, Header Length: 20, TTL: 64
                - Protocal: 6, Source: b'\n\x00\x02\x0f', Target: 10.0.2.15
-------------Packet Start-------------

Ethernet Frame:
        - Destination: 52:54:00:12:35:02, Protocal: 08:00:27:6A:A6:D6
        - IPV4 Packet:
                - version: 4, Header Length: 20, TTL: 64
                - Protocal: 6, Source: b'\n\x00\x02\x0f', Target: 10.0.2.15
-------------Packet Start-------------

Ethernet Frame:
        - Destination: 08:00:27:6A:A6:D6, Protocal: 52:54:00:12:35:02
        - IPV4 Packet:
                - version: 4, Header Length: 20, TTL: 64
                - Protocal: 6, Source: b'\r#\xd2w', Target: 13.35.210.119
-------------Packet Start-------------

Ethernet Frame:
        - Destination: 08:00:27:6A:A6:D6, Protocal: 52:54:00:12:35:02
        - IPV4 Packet:
                - version: 4, Header Length: 20, TTL: 64
                - Protocal: 6, Source: b'\r#\xd2w', Target: 13.35.210.119
-------------Packet Start-------------

Ethernet Frame:
        - Destination: 52:54:00:12:35:02, Protocal: 08:00:27:6A:A6:D6
        - IPV4 Packet:
                - version: 4, Header Length: 20, TTL: 64
                - Protocal: 6, Source: b'\n\x00\x02\x0f', Target: 10.0.2.15
```