School of Computer Science and Engineering

Network Analysis using Packet Sniffer

CSE3501 – Information Security Analysis and Audit

Submitted to:

Dr. Prasad M

Submitted by:

K Sreekar Reddy	(19BCE1227)
Chithirala Naga Chandan	(19BCE1604)
Bollineni Nishanth	(19BCE1805)

In partial fulfilment for the award of the degree of

Bachelor of Technology

in

Computer Science and Engineering

Problem Statement

I. Idea:

In the past few decades, computer networks have been increasing all over the world very rapidly. Number of its users have increased and also the traffic flow in these networks. So, it is very important to keep an eye on the traffic that flows between many servers and systems. It is also necessary to maintain the user's privacy and to keep the network smooth and it's a very tough task to maintain and monitor the network, because of the large amount of data available. For this purpose, packet sniffing is used. Packet sniffing is the process of capturing the information transmitted across networks. In this process NIC captures all traffic that flows inside or outside the network. Packet Sniffing mainly used in network management, monitoring and ethical hacking. To perform sniffing we use a tool named packet sniffer. A packet sniffer, sometimes referred to as a network analyser, which can be used by a network administrator to monitor and troubleshoot network traffic.

Sniffers work by examining streams of data packets that flow between computers on a network as well as between networked computers and the larger Internet. These packets are intended for — and addressed to — specific machines, but using a packet sniffer in "promiscuous mode" allows IT professionals, end users or malicious intruders to examine any packet, regardless of destination. It's possible to configure sniffers in two ways. The first is "unfiltered," meaning they will capture all packets possible and write them to a local hard drive for later examination. Next is "filtered" mode, meaning analysers will only capture packets that contain specific data elements.

Using a sniffer, it's possible to capture almost any information — for example, which websites that a user visits, what is viewed on the site, the contents and destination of any email along with details about any downloaded files. Protocol analysers are often used by companies to keep track of network use by employees and are also a part of many reputable antivirus software packages. Outward-facing sniffers scan incoming network traffic for specific elements of malicious code, helping to prevent computer virus infections and limit the spread of malware.

II. SCOPE:

The scope of the project mainly focusses on the basis of packet sniffer and working principle which is used to analyse the network traffic and also to implement a basic packet sniffer using python as a programming language. This application comprises of different modules which handle different tasks efficiently and also takes less memory constraints.

III. NOVELTY:

Majority of packet sniffers available on the network most are written in C language, but we implemented using python. The primary motivation of this language was the need for a platform-independent (i.e., architecture neutral) language that could be used to create software to be embedded in various consumer electronic devices. The application also uses less memory and very simple to use.

IV. Comparative Statement:

https://www.researchgate.net/publication/323949095_A_survey_on_sniffing_attacks_on_computer_networks

"*Generally Malicious users make use of different attacks at different levels to steal different level of data. Some of the sniffing attacks that can be used in different levels of networking/transmission are Media Access Control (MAC) Flooding, Dynamic Host Configuration Protocol (DHCP) Attacks, DHCP Starvation Attack, Rogue DHCP Server Attack, Address Resolution*

Protocol (ARP) Spoofing, MAC spoofing and Domain Name Server (DNS) Poisoning. In this paper, a comparative study has been done with the above-mentioned sniffing attacks and the level of recovery that can be done with each sniffing attack."

Existing Packet Sniffers:

Tcpdump by McCanne, Leres and Jacobson

It is one of the most popular packet sniffers. Tcpdump is accompanied by the libpcap library. It was originally written in 1987 at the Lawrence Berkeley National Laboratory and published a few years later and quickly gained users attention.

Libpcap is a C library for capturing packets. The procedures included in libpcap provide a standardized interface to all common (UNIX-based) operating systems, including Linux and FreeBSD. The interface of the libpcap is usable even under Windows but there the library is called winpcap.

Tcpdump is a common packet analyser that runs under the command line and parsing tool ported to several platforms. It allows the user to intercept and display TCP/IP and other packets being transmitted or received over a network to which the computer is attached. Tcpdump works by capturing and displaying packet headers and matching them against a set of criteria.

It runs on most UNIX-like operating systems - e.g. Linux, BSD, Solaris, Mac OS X, HP-UX and AIX amongst others making use of the libpcap library to capture packets.

Wireshark by Gerald Combs

Wireshark is a free and open- source packet analyser and it is written in C. It is used for network troubleshooting, analysis, software and communications protocol development, and education. Originally named Ethereal, in May 2006 the project was renamed Wireshark due to trademark issues.

Wireshark is very similar to tcpdump, but has a graphical front-end, plus some integrated sorting and filtering options.

Wireshark allows the user to put network interface controllers that support promiscuous mode into that mode, in order to see all traffic visible on that interface, not just traffic addressed to one of the interface's configured addresses and broadcast/multicast traffic. However, when capturing with a packet analyser in promiscuous mode on a port on a network switch, not all of the traffic traveling through the switch will necessarily be sent to the port on which the capture is being done, so capturing in promiscuous mode will not necessarily be sufficient to see all traffic on the network. Port mirroring or various network taps extend capture to any point on net; simple passive taps are extremely resistant to malware tampering.

V. DATASET:

There is no particular dataset used in the project. We studied existing packet sniffing software like Wireshark, TCPDUMP etc and how they work.

Here we used the Packet sniffer we implemented, to capture the packets to analyse the captured packets.

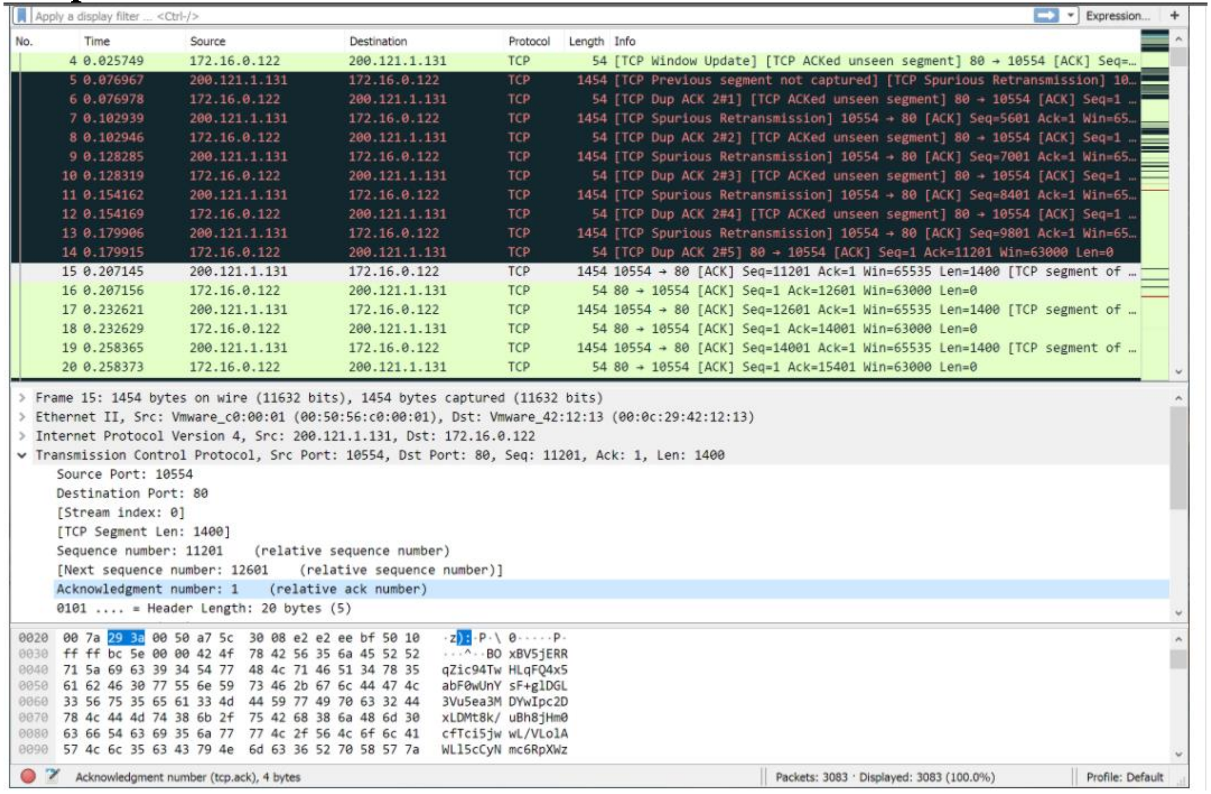
VI. TEST BED:

OS	-	Linux
Virtual Machine	-	Oracle virtual box
Browser	-	IE/FireFox
Programming Language	-	Python
Python version	-	version 3.9
IDE	-	Pycharm

VII. Expected Result:

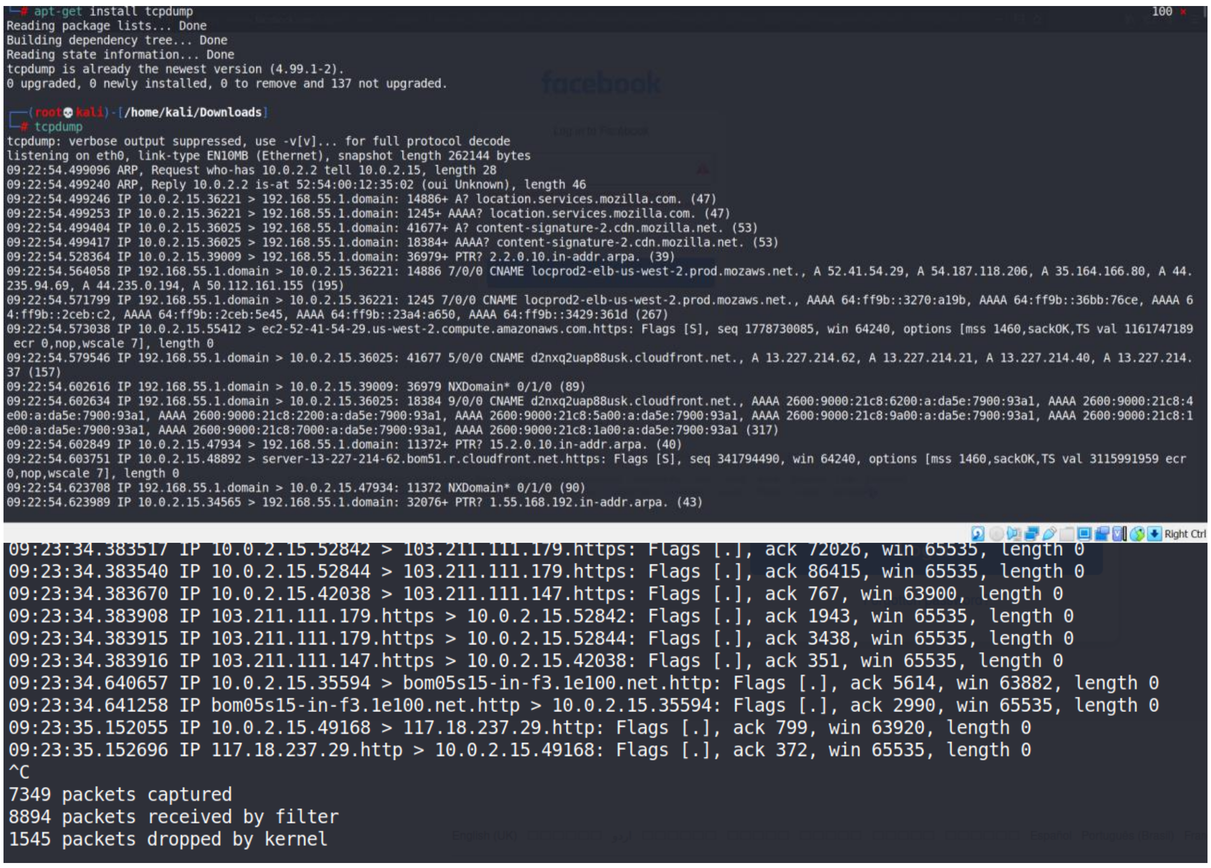
We are required to capture the packets from the system that are transferred over the network using packet sniffer that has been implemented using python and gain the information about the packets like its source and destination MAC addresses and IP addresses. It should also the describe the protocol of IP packets like TCP, UDP, ICMP and also raw data in encrypted form.

Output of Wireshark:



```
No.  Time      Source         Destination    Protocol Length Info
   4 0.025749  172.16.0.122   200.121.1.131  TCP        54 [TCP Window Update] [TCP ACKed unseen segment] 80 → 10554 [ACK] Seq=…
   5 0.076967  200.121.1.131  172.16.0.122   TCP      1454 [TCP Previous segment not captured] [TCP Spurious Retransmission] 10…
   6 0.076978  172.16.0.122   200.121.1.131  TCP        54 [TCP Dup ACK 2#1] [TCP ACKed unseen segment] 80 → 10554 [ACK] Seq=1 …
   7 0.102939  200.121.1.131  172.16.0.122   TCP      1454 [TCP Spurious Retransmission] 10554 → 80 [ACK] Seq=5601 Ack=1 Win=65…
   8 0.102946  172.16.0.122   200.121.1.131  TCP        54 [TCP Dup ACK 2#2] [TCP ACKed unseen segment] 80 → 10554 [ACK] Seq=1 …
   9 0.128285  200.121.1.131  172.16.0.122   TCP      1454 [TCP Spurious Retransmission] 10554 → 80 [ACK] Seq=7001 Ack=1 Win=65…
  10 0.128319  172.16.0.122   200.121.1.131  TCP        54 [TCP Dup ACK 2#3] [TCP ACKed unseen segment] 80 → 10554 [ACK] Seq=1 …
  11 0.154162  200.121.1.131  172.16.0.122   TCP      1454 [TCP Spurious Retransmission] 10554 → 80 [ACK] Seq=8401 Ack=1 Win=65…
  12 0.154169  172.16.0.122   200.121.1.131  TCP        54 [TCP Dup ACK 2#4] [TCP ACKed unseen segment] 80 → 10554 [ACK] Seq=1 …
  13 0.179906  200.121.1.131  172.16.0.122   TCP      1454 [TCP Spurious Retransmission] 10554 → 80 [ACK] Seq=9801 Ack=1 Win=65…
  14 0.179915  172.16.0.122   200.121.1.131  TCP        54 [TCP Dup ACK 2#5] 80 → 10554 [ACK] Seq=1 Ack=11201 Win=63000 Len=0
  15 0.207145  200.121.1.131  172.16.0.122   TCP      1454 10554 → 80 [ACK] Seq=11201 Ack=1 Win=65535 Len=1400 [TCP segment of …
  16 0.207156  172.16.0.122   200.121.1.131  TCP        54 80 → 10554 [ACK] Seq=1 Ack=12601 Win=63000 Len=0
  17 0.232621  200.121.1.131  172.16.0.122   TCP      1454 10554 → 80 [ACK] Seq=12601 Ack=1 Win=65535 Len=1400 [TCP segment of …
  18 0.232629  172.16.0.122   200.121.1.131  TCP        54 80 → 10554 [ACK] Seq=1 Ack=14001 Win=63000 Len=0
  19 0.258365  200.121.1.131  172.16.0.122   TCP      1454 10554 → 80 [ACK] Seq=14001 Ack=1 Win=65535 Len=1400 [TCP segment of …
  20 0.258373  172.16.0.122   200.121.1.131  TCP        54 80 → 10554 [ACK] Seq=1 Ack=15401 Win=63000 Len=0

> Frame 15: 1454 bytes on wire (11632 bits), 1454 bytes captured (11632 bits)
> Ethernet II, Src: Vmware_c0:00:01 (00:50:56:c0:00:01), Dst: Vmware_42:12:13 (00:0c:29:42:12:13)
> Internet Protocol Version 4, Src: 200.121.1.131, Dst: 172.16.0.122
v Transmission Control Protocol, Src Port: 10554, Dst Port: 80, Seq: 11201, Ack: 1, Len: 1400
    Source Port: 10554
    Destination Port: 80
    [Stream index: 0]
    [TCP Segment Len: 1400]
    Sequence number: 11201    (relative sequence number)
    [Next sequence number: 12601    (relative sequence number)]
    Acknowledgment number: 1    (relative ack number)
    0101 .... = Header Length: 20 bytes (5)

0020  00 7a 29 3a 00 50 a7 5c  30 08 e2 e2 ee bf 50 10   ·z):·P·\ 0·····P·
0030  ff ff bc 5e 00 00 42 4f  78 42 56 35 6a 45 52 52   ···^··BO xBV5jERR
0040  71 5a 69 63 39 34 54 77  48 4c 71 46 51 34 78 35   qZic94Tw HLqFQ4x5
0050  61 62 46 30 77 55 6e 59  73 46 2b 67 6c 44 47 4c   abF0wUnY sF+glDGL
0060  33 56 75 35 65 61 33 4d  44 59 77 49 70 63 32 44   3Vu5ea3M DYwIpc2D
0070  78 4c 44 4d 74 38 6b 2f  75 42 68 38 6a 48 6d 30   xLDMt8k/ uBh8jHm0
0080  63 66 54 63 69 35 6a 77  77 4c 2f 56 4c 6f 6c 41   cfTci5jw wL/VLolA
0090  57 4c 6c 35 63 43 79 4e  6d 63 36 52 70 58 57 7a   WLl5cCyN mc6RpXWz

Acknowledgment number (tcp.ack), 4 bytes    Packets: 3083 · Displayed: 3083 (100.0%)    Profile: Default
```

Output of TCPDUMP



```
└─# apt-get install tcpdump
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
tcpdump is already the newest version (4.99.1-2).
0 upgraded, 0 newly installed, 0 to remove and 137 not upgraded.

┌──(root💀kali)-[/home/kali/Downloads]
└─# tcpdump
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
09:22:54.499096 ARP, Request who-has 10.0.2.2 tell 10.0.2.15, length 28
09:22:54.499240 ARP, Reply 10.0.2.2 is-at 52:54:00:12:35:02 (oui Unknown), length 46
09:22:54.499246 IP 10.0.2.15.36221 > 192.168.55.1.domain: 14886+ A? location.services.mozilla.com. (47)
09:22:54.499253 IP 10.0.2.15.36221 > 192.168.55.1.domain: 1245+ AAAA? location.services.mozilla.com. (47)
09:22:54.499404 IP 10.0.2.15.36025 > 192.168.55.1.domain: 41677+ A? content-signature-2.cdn.mozilla.net. (53)
09:22:54.499417 IP 10.0.2.15.36025 > 192.168.55.1.domain: 18384+ AAAA? content-signature-2.cdn.mozilla.net. (53)
09:22:54.528364 IP 10.0.2.15.39009 > 192.168.55.1.domain: 36979+ PTR? 2.2.0.10.in-addr.arpa. (39)
09:22:54.564058 IP 192.168.55.1.domain > 10.0.2.15.36221: 14886 7/0/0 CNAME locprod2-elb-us-west-2.prod.mozaws.net., A 52.41.54.29, A 54.187.118.206, A 35.164.166.80, A 44.
235.94.69, A 44.235.0.194, A 50.112.161.155 (195)
09:22:54.571799 IP 192.168.55.1.domain > 10.0.2.15.36221: 1245 7/0/0 CNAME locprod2-elb-us-west-2.prod.mozaws.net., AAAA 64:ff9b::3270:a19b, AAAA 64:ff9b::36bb:76ce, AAAA 6
4:ff9b::2ceb:c2, AAAA 64:ff9b::2ceb:5e45, AAAA 64:ff9b::23a4:a650, AAAA 64:ff9b::3429:361d (267)
09:22:54.573038 IP 10.0.2.15.55412 > ec2-52-41-54-29.us-west-2.compute.amazonaws.com.https: Flags [S], seq 1778730085, win 64240, options [mss 1460,sackOK,TS val 1161747189
 ecr 0,nop,wscale 7], length 0
09:22:54.579546 IP 192.168.55.1.domain > 10.0.2.15.36025: 41677 5/0/0 CNAME d2nxq2uap88usk.cloudfront.net., A 13.227.214.62, A 13.227.214.21, A 13.227.214.40, A 13.227.214.
37 (157)
09:22:54.602616 IP 192.168.55.1.domain > 10.0.2.15.39009: 36979 NXDomain* 0/1/0 (89)
09:22:54.602634 IP 192.168.55.1.domain > 10.0.2.15.36025: 18384 9/0/0 CNAME d2nxq2uap88usk.cloudfront.net., AAAA 2600:9000:21c8:6200:a:da5e:7900:93a1, AAAA 2600:9000:21c8:4
e00:a:da5e:7900:93a1, AAAA 2600:9000:21c8:2200:a:da5e:7900:93a1, AAAA 2600:9000:21c8:5a00:a:da5e:7900:93a1, AAAA 2600:9000:21c8:9a00:a:da5e:7900:93a1, AAAA 2600:9000:21c8:1
e00:a:da5e:7900:93a1, AAAA 2600:9000:21c8:7000:a:da5e:7900:93a1, AAAA 2600:9000:21c8:1a00:a:da5e:7900:93a1 (317)
09:22:54.602849 IP 10.0.2.15.47934 > 192.168.55.1.domain: 11372+ PTR? 15.2.0.10.in-addr.arpa. (40)
09:22:54.603751 IP 10.0.2.15.48892 > server-13-227-214-62.bom51.r.cloudfront.net.https: Flags [S], seq 341794490, win 64240, options [mss 1460,sackOK,TS val 3115991959 ecr
0,nop,wscale 7], length 0
09:22:54.623708 IP 192.168.55.1.domain > 10.0.2.15.47934: 11372 NXDomain* 0/1/0 (90)
09:22:54.623989 IP 10.0.2.15.34565 > 192.168.55.1.domain: 32076+ PTR? 1.55.168.192.in-addr.arpa. (43)
```

```
09:23:34.383517 IP 10.0.2.15.52842 > 103.211.111.179.https: Flags [.], ack 72026, win 65535, length 0
09:23:34.383540 IP 10.0.2.15.52844 > 103.211.111.179.https: Flags [.], ack 86415, win 65535, length 0
09:23:34.383670 IP 10.0.2.15.42038 > 103.211.111.147.https: Flags [.], ack 767, win 63900, length 0
09:23:34.383908 IP 103.211.111.179.https > 10.0.2.15.52842: Flags [.], ack 1943, win 65535, length 0
09:23:34.383915 IP 103.211.111.179.https > 10.0.2.15.52844: Flags [.], ack 3438, win 65535, length 0
09:23:34.383916 IP 103.211.111.147.https > 10.0.2.15.42038: Flags [.], ack 351, win 65535, length 0
09:23:34.640657 IP 10.0.2.15.35594 > bom05s15-in-f3.1e100.net.http: Flags [.], ack 5614, win 63882, length 0
09:23:34.641258 IP bom05s15-in-f3.1e100.net.http > 10.0.2.15.35594: Flags [.], ack 2990, win 65535, length 0
09:23:35.152055 IP 10.0.2.15.49168 > 117.18.237.29.http: Flags [.], ack 799, win 63920, length 0
09:23:35.152696 IP 117.18.237.29.http > 10.0.2.15.49168: Flags [.], ack 372, win 65535, length 0
^C
7349 packets captured
8894 packets received by filter
1545 packets dropped by kernel
```

Architecture:

I. High level design (Black box design):

The proposed system will be written in Python unlike the other Sniffers that are written in C language. It will capture packets and size of the packet and source and destination machine IP addresses which are involved in the packet transferring. It can show this process in graphical manner. It also shows the working of different layers in graphical manner. It gives complete information about the captured packets; like which layers are involved and which protocols are in use at a particular time

The design of the proposed system discusses the various requirements that will make up the application. By conducting the requirements analysis we listed out the requirements that are useful to restate the problem definition.

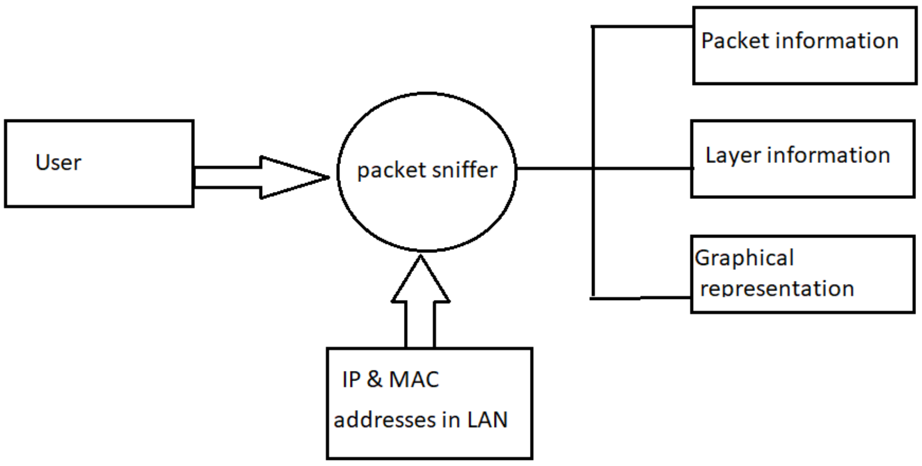
- Analyse network Layer
- Analyse Transport Layer
- Analyse Application Layer

- Analyse UDP Protocol
- Analyse TCP Protocol
- Analyse HTTP Protocol
- Find out the Packets over network

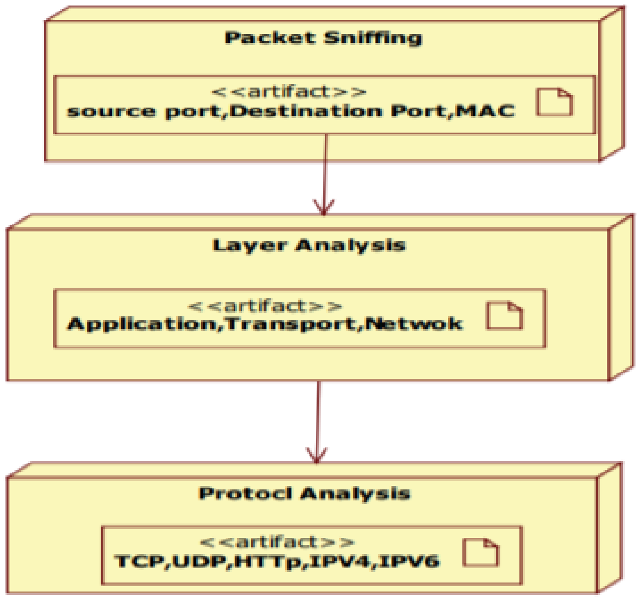
This application is designed into 4 independent modules which take care of different tasks efficiently.

1. User Interface Module.
2. Packet Sniffing Module.
3. Analyse layers Module.
4. Protocol Analysis Module.

DATA FLOW DIAGRAM



II. Low level design



User Interface Module

User Interface Module Actually every application has one user interface for accessing the entire application. The user interface for the Packet sniffer application is designed completely based on the end users. It provides an easy-to-use interface to the users. This user interface has an attractive look and provides ease of navigation.

Packet Sniffing Module

This module takes care of capturing packets that are seen by a machine's network interface. It grabs all the packets that goes in and out of the Network Interface Card (NIC) of the machine on which the sniffer is installed. This means that, if the NIC is set to the promiscuous mode, then it will receive all the packets sent to the network.

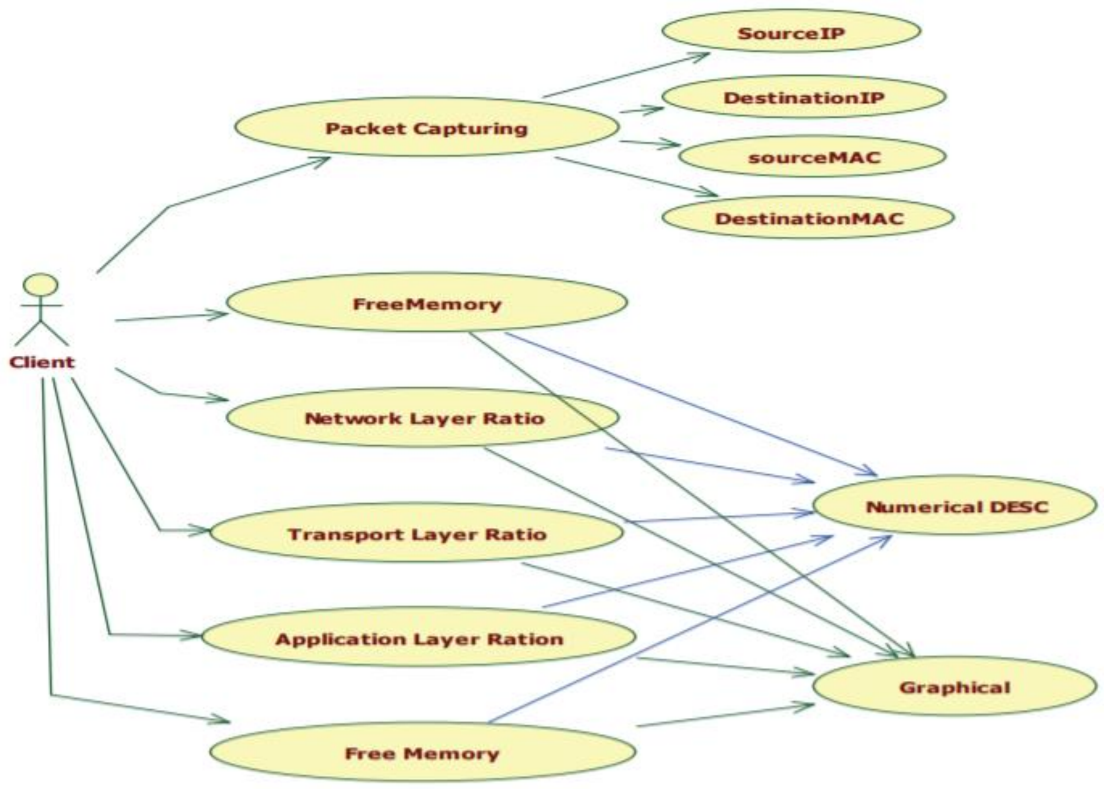
Analyse layers Module

This module contains the code for analysing the layers in the system. Mostly in this module we have to discuss about three layers Transport layer, Application Layer, Network Layer.

Protocol Analysis Module

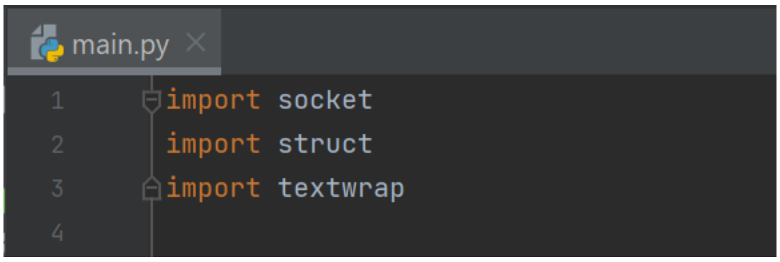
This module analyses the protocols of the layers. Like Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Hypertext Transfer Protocol (HTTP) etc. It can show the source port, destination port and packet length of the system of each protocol.

USECASE DIAGRAM



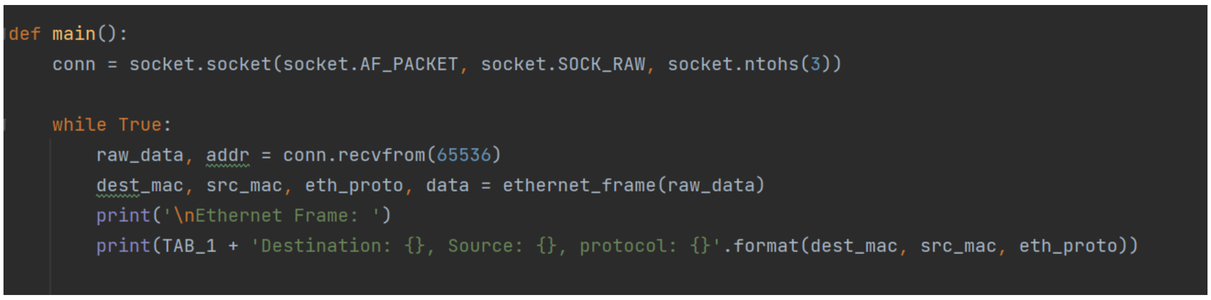
Implementation

I. Capturing packets using socket library



```python
import socket
import struct
import textwrap
```

Importing socket library in python, A simple packet sniffer in Python can be created with the help socket module. We can use the raw socket type to get the packets. A raw socket provides access to the underlying protocols, which support socket abstractions. Since raw sockets are part of the internet socket API, they can only be used to generate and receive IP packets.



```python
def main():
    conn = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))

    while True:
        raw_data, addr = conn.recvfrom(65536)
        dest_mac, src_mac, eth_proto, data = ethernet_frame(raw_data)
        print('\nEthernet Frame: ')
        print(TAB_1 + 'Destination: {}, Source: {}, protocol: {}'.format(dest_mac, src_mac, eth_proto))
```

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

Both reading and writing to a raw socket require creating a raw socket first. Here we use the INET family raw socket. The family parameter for a socket describes the address family of the socket. The following are the address family constants:

• AF_LOCAL: Used for local communication

• AF_UNIX: Unix domain sockets
• AF_INET: IP version 4
• AF_INET6: IP version 6
• AF_IPX: Novell IPX
• AF_NETLINK: Kernel user-interface device
• AF_X25: Reserved for X.25 project
• AF_AX25: Amateur Radio AX.25
• AF_APPLETALK: Appletalk DDP
• AF_PACKET: Low-level packet interface
• AF_ALG: Interface to kernel crypto API

The next parameter passed is the type of the socket. The following are the possible values for the socket type:

• SOCK_STREAM: Stream (connection) socket
• SOCK_DGRAM: Datagram (connection-less) socket
• SOCK_RAW: RAW socket
• SOCK_RDM: Reliably delivered message
• SOCK_SEQPACKET: Sequential packet socket
• SOCK_PACKET: Linux-specific method of getting packets at the development level

The last parameter is the protocol of the packet. This protocol number is defined by the Internet Assigned Numbers Authority (IANA). We have to be aware of the family of the socket; then we can only choose a protocol. As we selected AF_INET (IPV4), we can only select IP-based protocols.

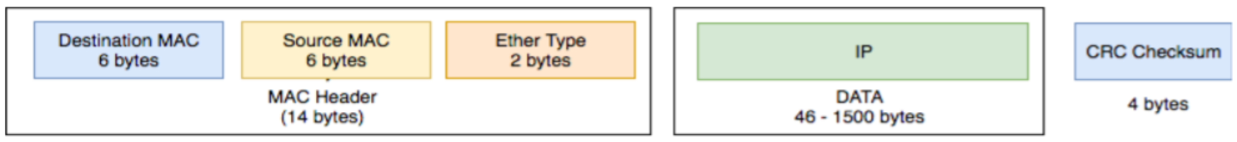
Next, start an infinite loop to receive data from the socket:

```
while True:
        print(conn.recvfrom(65565))
```

The recvfrom method in the socket module helps us to receive all the data from the socket. The parameter passed is the buffer size; 65565 is the maximum buffer size

II. Parsing the packet

Now we can try to parse the data that we sniffed, and unpack the headers. To parse a packet, we need to have an idea of the Ethernet frame and the packet headers of the IP. The Ethernet frame structure is as follows:



The first six bytes are for the Destination MAC address and the next six bytes are for the Source MAC. The last two bytes are for the Ether Type. The rest 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.

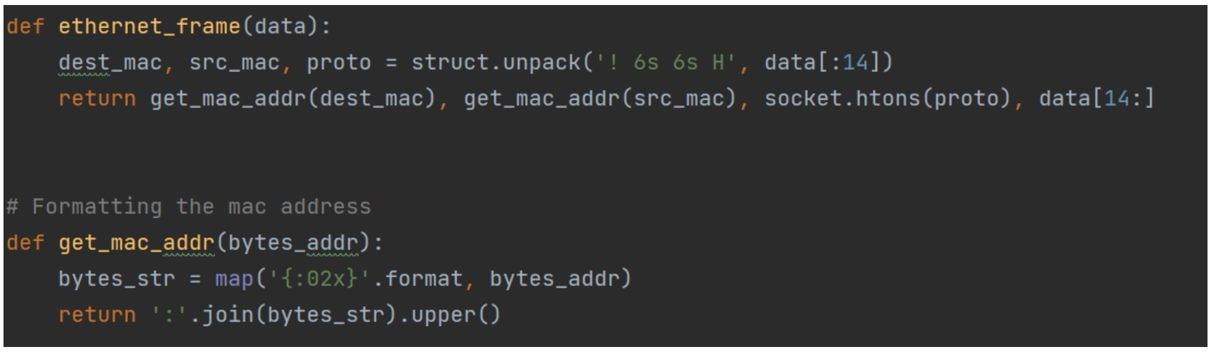
• **Flags** (three bits): The second bit represents the Don't Fragment bit. When this bit is set, the IP datagram is never fragmented. The third bit represents the More Fragment bit. If this bit is set, then it represents a fragmented IP datagram that has more fragments after it.

• **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.

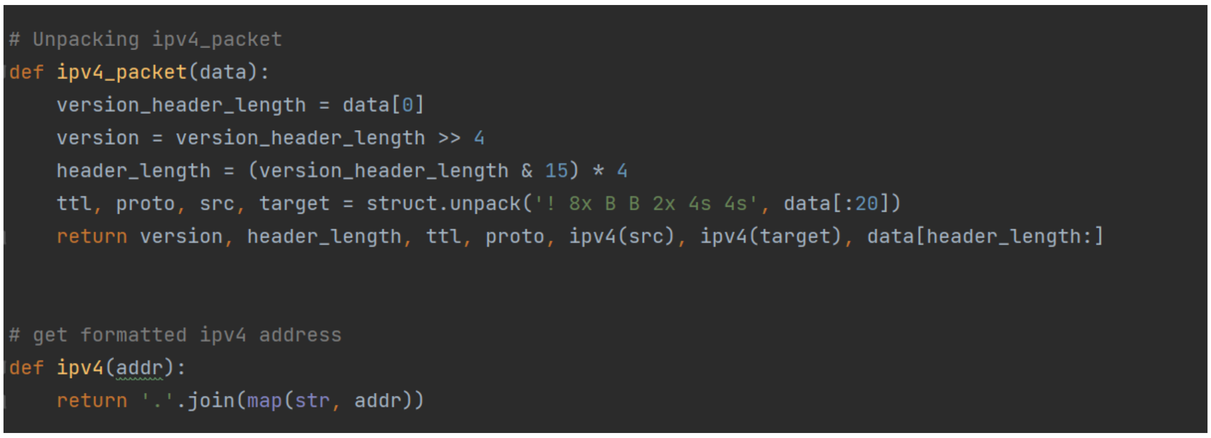
Refer to the RFC 791 document for more details on IP headers: https://tools.ietf.org/html/rfc791



```
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:]


# Formatting the mac address
def get_mac_addr(bytes_addr):
    bytes_str = map('{:02x}'.format, bytes_addr)
    return ':'.join(bytes_str).upper()
```

The above code unpacks ethernet frame from raw data collected from socket and unpacks and formats also returns the destination and Source MAC addresses, protocol and remaining payload.

III. Unpacking and formatting IP packets



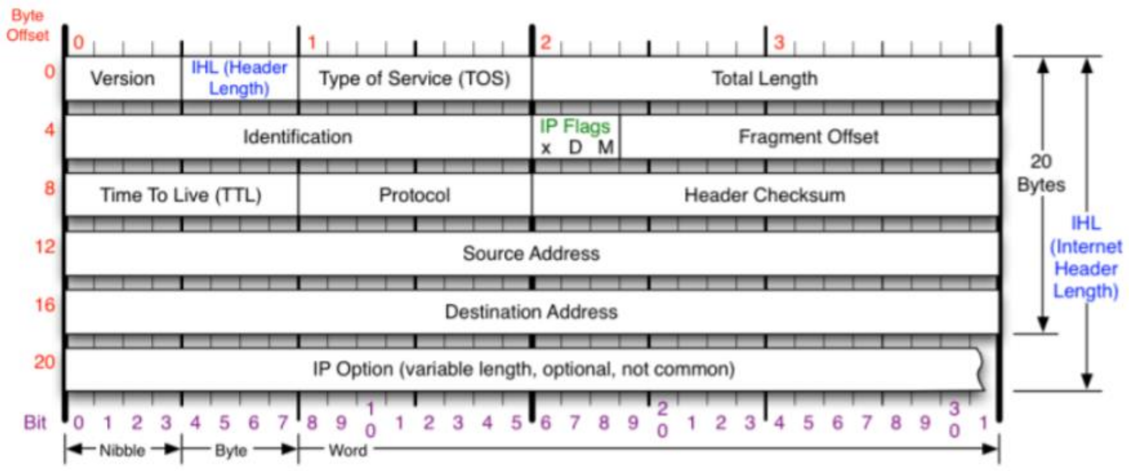
```python
# Unpacking ipv4_packet
def ipv4_packet(data):
    version_header_length = data[0]
    version = version_header_length >> 4
    header_length = (version_header_length & 15) * 4
    ttl, proto, src, target = struct.unpack('! 8x B B 2x 4s 4s', data[:20])
    return version, header_length, ttl, proto, ipv4(src), ipv4(target), data[header_length:]


# get formatted ipv4 address
def ipv4(addr):
    return '.'.join(map(str, addr))
```

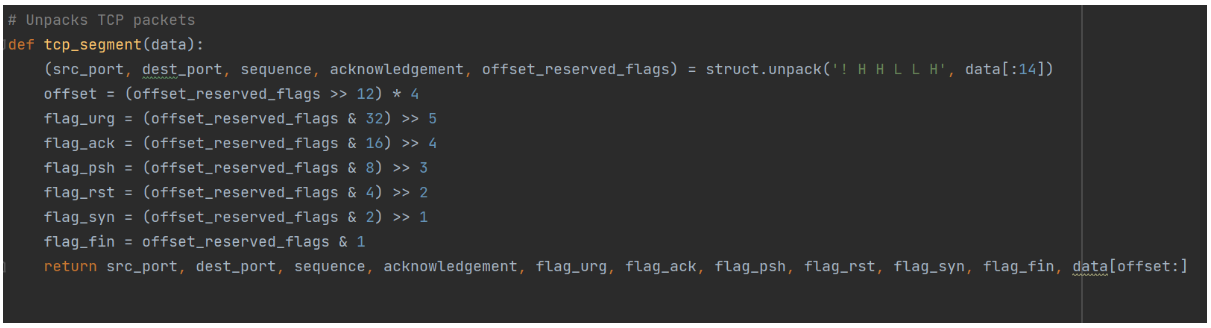
IV. Protocol Identification:

Now that we have the internet layer unpacked, the next layer we have to unpack is the transport layer. We can determine the protocol from the protocol ID in the IP header. The following are the protocol IDs for some of the protocols:

- TCP: 6
- ICMP: 1
- UDP: 17

V. TCP segmentation





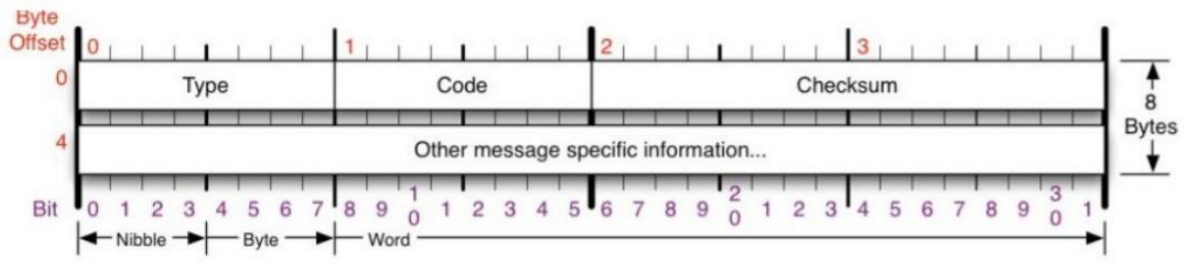
```
# Unpacks TCP packets
def tcp_segment(data):
    (src_port, dest_port, sequence, acknowledgement, offset_reserved_flags) = struct.unpack('! H H L L H', data[:14])
    offset = (offset_reserved_flags >> 12) * 4
    flag_urg = (offset_reserved_flags & 32) >> 5
    flag_ack = (offset_reserved_flags & 16) >> 4
    flag_psh = (offset_reserved_flags & 8) >> 3
    flag_rst = (offset_reserved_flags & 4) >> 2
    flag_syn = (offset_reserved_flags & 2) >> 1
    flag_fin = offset_reserved_flags & 1
    return src_port, dest_port, sequence, acknowledgement, flag_urg, flag_ack, flag_psh, flag_rst, flag_syn, flag_fin, data[offset:]
```

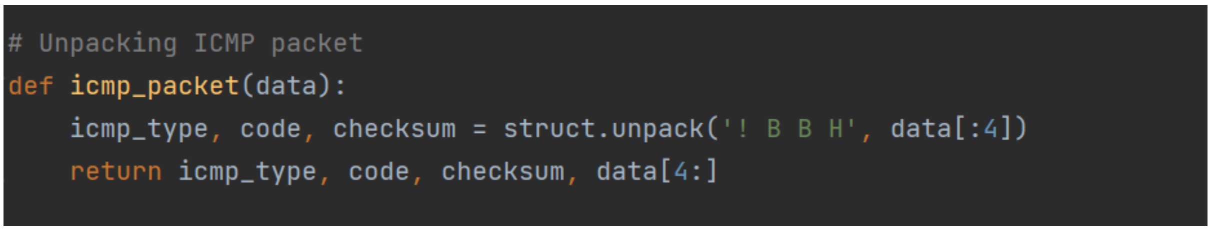
The above code unpacks TCP packets and returns the following:

- Source port
- Destination port
- Sequence
- Acknowledgement
- Flags

VI. Unpacks ICMP packets

The packets are unpacked according to the packet header structure. Here is the packet header structure for ICMP:





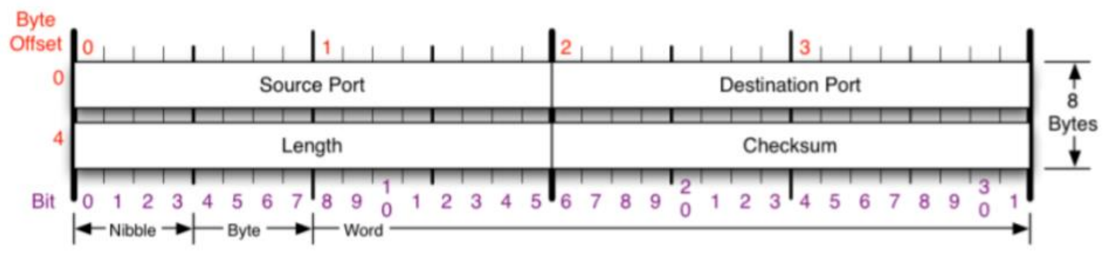
```
# Unpacking ICMP packet
def icmp_packet(data):
    icmp_type, code, checksum = struct.unpack('! B B H', data[:4])
    return icmp_type, code, checksum, data[4:]
```

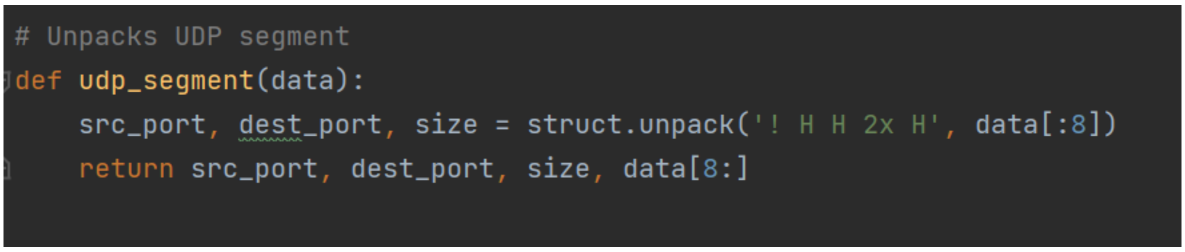
The above code returns the following data:

- TYPE
- CODE
- CHECKSUM

VII. Unpack UDP packets

Here is the packet header structure for UDP:





```
# Unpacks UDP segment
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:]
```

The following code returns:

- Source port
- Destination port
- Length of packet

Results and Discussions

Code:

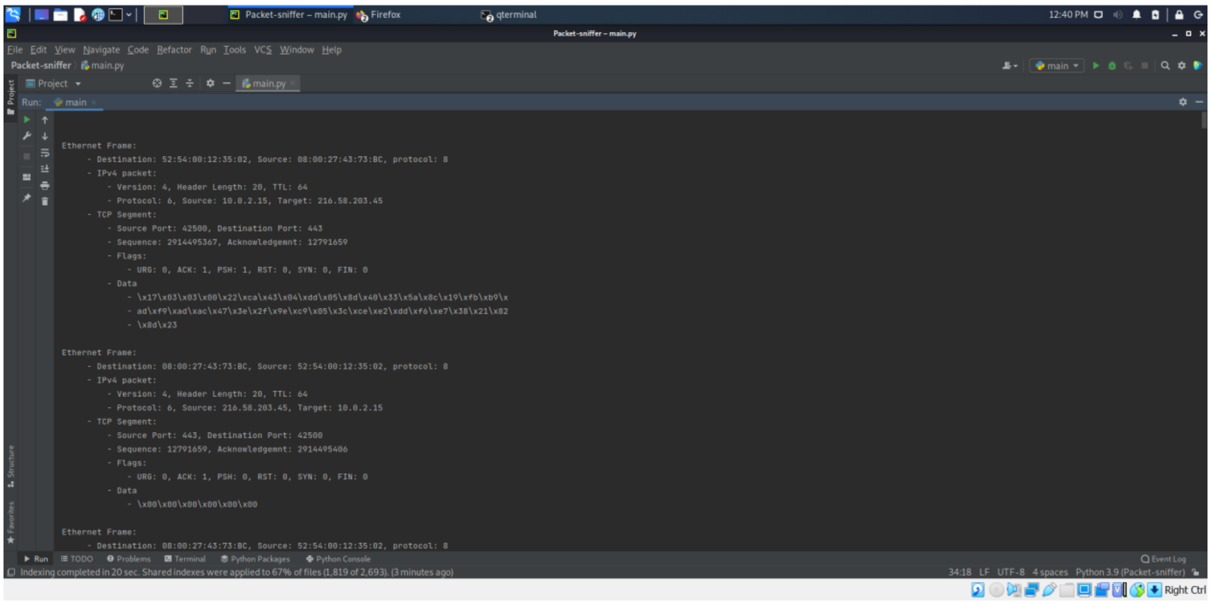
```
import socket
import struct
import textwrap

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 - '


def main():
    conn = socket.socket(socket.AF_PACKET, socket.SOCK_RAW,
socket.ntohs(3))

    while True:
        raw_data, addr = conn.recvfrom(65536)
        dest_mac, src_mac, eth_proto, data = ethernet_frame(raw_data)
        print('\nEthernet Frame: ')
        print(TAB_1 + 'Destination: {}, Source: {}, protocol:
{}'.format(dest_mac, src_mac, eth_proto))

        # 8 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 + 'Protocol: {}, Source: {}, Target:
{}'.format(proto, src, target))

            # 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))

            # TCP
            elif proto == 6:
                (src_port, dest_port, sequence, acknowledgement, flag_urg,
flag_ack, flag_psh, flag_rst, flag_syn,
                 flag_fin, data) = tcp_segment(data)
                print(TAB_1 + 'TCP Segment:')
                print(TAB_2 + 'Source Port: {}, Destination Port:
{}'.format(src_port, dest_port))
                print(TAB_2 + 'Sequence: {}, Acknowledgemnt:
{}'.format(sequence, acknowledgement))
                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))

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

```python
                print(TAB_2 + 'Data')
                print(format_multi_line(DATA_TAB_3, data))

            # Other
            else:
                print(TAB_1 + 'Data')
                print(format_multi_line(DATA_TAB_2, data))


# Unpack ethernet frame
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:]


# Formatting the mac address
def get_mac_addr(bytes_addr):
    bytes_str = map('{:02x}'.format, bytes_addr)
    return ':'.join(bytes_str).upper()


# Unpacking ipv4_packet
def ipv4_packet(data):
    version_header_length = data[0]
    version = version_header_length >> 4
    header_length = (version_header_length & 15) * 4
    ttl, proto, src, target = struct.unpack('! 8x B B 2x 4s 4s', data[:20])
    return version, header_length, ttl, proto, ipv4(src), ipv4(target),
data[header_length:]


# get formatted ipv4 address
def ipv4(addr):
    return '.'.join(map(str, addr))


# Unpacking ICMP packet
def icmp_packet(data):
    icmp_type, code, checksum = struct.unpack('! B B H', data[:4])
    return icmp_type, code, checksum, data[4:]


# Unpacks TCP packets
def tcp_segment(data):
    (src_port, dest_port, sequence, acknowledgement, offset_reserved_flags)
= struct.unpack('! H H L L H', data[:14])
    offset = (offset_reserved_flags >> 12) * 4
    flag_urg = (offset_reserved_flags & 32) >> 5
    flag_ack = (offset_reserved_flags & 16) >> 4
    flag_psh = (offset_reserved_flags & 8) >> 3
    flag_rst = (offset_reserved_flags & 4) >> 2
    flag_syn = (offset_reserved_flags & 2) >> 1
    flag_fin = offset_reserved_flags & 1
    return src_port, dest_port, sequence, acknowledgement, flag_urg,
flag_ack, flag_psh, flag_rst, flag_syn, flag_fin, data[offset:]


# Unpacks UDP segment
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:]


# Formatting Output
def format_multi_line(prefix, string, size=80):
    size -= len(prefix)
    if isinstance(string, bytes):
        string = ''.join(r'\x{:02x}'.format(byte) for byte in string)
        if size % 2:
            size -= 1
    return '\n'.join([prefix + line for line in textwrap.wrap(string,
size)])


main()
```

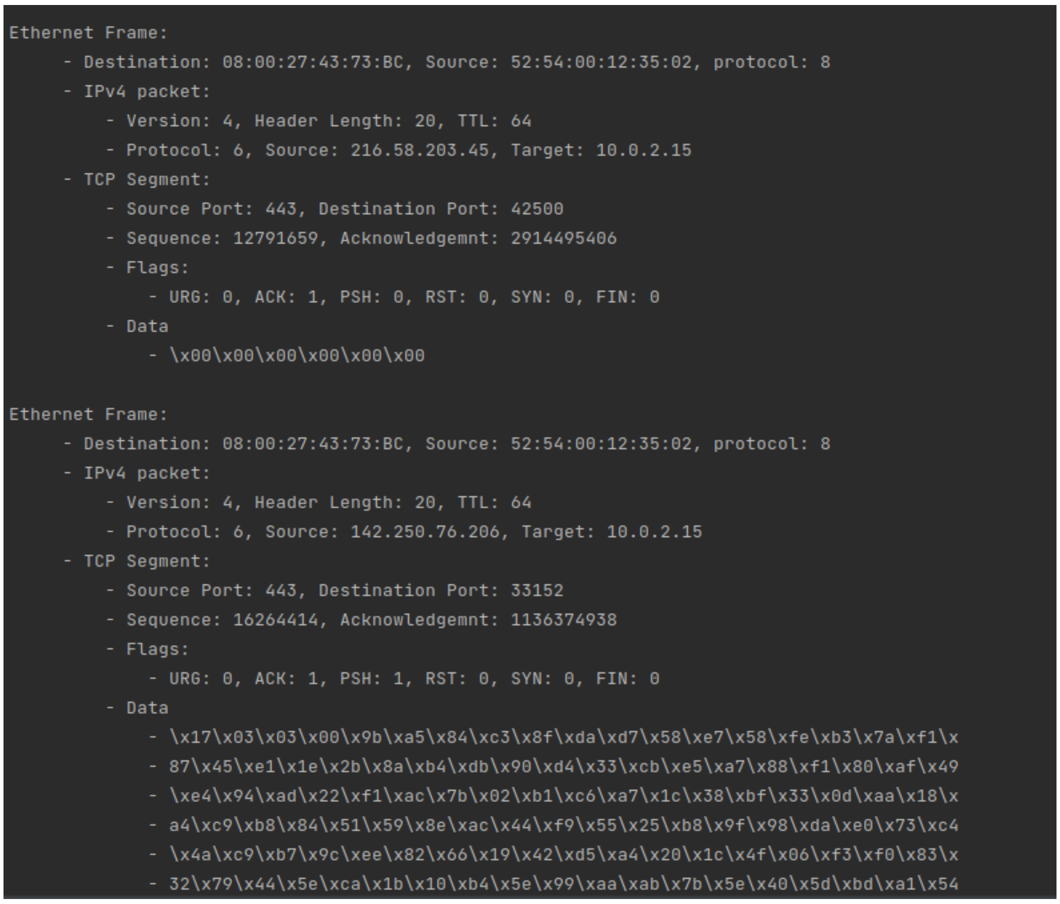
Running and Output:

Run the code in pycharm ide in kali Linux.



```
Ethernet Frame:
    - Destination: 52:54:00:12:35:02, Source: 08:00:27:43:73:BC, protocol: 8
    - IPv4 packet:
        - Version: 4, Header Length: 20, TTL: 64
        - Protocol: 6, Source: 10.0.2.15, Target: 216.58.203.45
    - TCP Segment:
        - Source Port: 42500, Destination Port: 443
        - Sequence: 2914495367, Acknowledgemnt: 12791659
        - Flags:
            - URG: 0, ACK: 1, PSH: 1, RST: 0, SYN: 0, FIN: 0
        - Data
            - \x17\x03\x03\x00\x22\xca\x43\x04\xdd\x05\x8d\x40\x33\x5a\x8c\x19\xfb\xb9\x
            - ad\xf9\xad\xac\x47\x3e\x2f\x9e\xc9\x05\x3c\xce\xe2\xdd\xf6\xe7\x38\x21\x82
            - \x8d\x23

Ethernet Frame:
    - Destination: 08:00:27:43:73:BC, Source: 52:54:00:12:35:02, protocol: 8
    - IPv4 packet:
        - Version: 4, Header Length: 20, TTL: 64
        - Protocol: 6, Source: 216.58.203.45, Target: 10.0.2.15
    - TCP Segment:
        - Source Port: 443, Destination Port: 42500
        - Sequence: 12791659, Acknowledgemnt: 2914495406
        - Flags:
            - URG: 0, ACK: 1, PSH: 0, RST: 0, SYN: 0, FIN: 0
        - Data
            - \x00\x00\x00\x00\x00\x00

Ethernet Frame:
    - Destination: 08:00:27:43:73:BC, Source: 52:54:00:12:35:02, protocol: 8
```

Indexing completed in 20 sec. Shared indexes were applied to 67% of files (1,819 of 2,693). (3 minutes ago)



```
Ethernet Frame:
     - Destination: 08:00:27:43:73:BC, Source: 52:54:00:12:35:02, protocol: 8
     - IPv4 packet:
         - Version: 4, Header Length: 20, TTL: 64
         - Protocol: 6, Source: 216.58.203.45, Target: 10.0.2.15
     - TCP Segment:
         - Source Port: 443, Destination Port: 42500
         - Sequence: 12791659, Acknowledgemnt: 2914495406
         - Flags:
             - URG: 0, ACK: 1, PSH: 0, RST: 0, SYN: 0, FIN: 0
         - Data
             - \x00\x00\x00\x00\x00\x00

Ethernet Frame:
     - Destination: 08:00:27:43:73:BC, Source: 52:54:00:12:35:02, protocol: 8
     - IPv4 packet:
         - Version: 4, Header Length: 20, TTL: 64
         - Protocol: 6, Source: 142.250.76.206, Target: 10.0.2.15
     - TCP Segment:
         - Source Port: 443, Destination Port: 33152
         - Sequence: 16264414, Acknowledgemnt: 1136374938
         - Flags:
             - URG: 0, ACK: 1, PSH: 1, RST: 0, SYN: 0, FIN: 0
         - Data
             - \x17\x03\x03\x00\x9b\xa5\x84\xc3\x8f\xda\xd7\x58\xe7\x58\xfe\xb3\x7a\xf1\x
             - 87\x45\xe1\x1e\x2b\x8a\xb4\xdb\x90\xd4\x33\xcb\xe5\xa7\x88\xf1\x80\xaf\x49
             - \xe4\x94\xad\x22\xf1\xac\x7b\x02\xb1\xc6\xa7\x1c\x38\xbf\x33\x0d\xaa\x18\x
             - a4\xc9\xb8\x84\x51\x59\x8e\xac\x44\xf9\x55\x25\xb8\x9f\x98\xda\xe0\x73\xc4
             - \x4a\xc9\xb7\x9c\xee\x82\x66\x19\x42\xd5\xa4\x20\x1c\x4f\x06\xf3\xf0\x83\x
             - 32\x79\x44\x5e\xca\x1b\x10\xb4\x5e\x99\xaa\xab\x7b\x5e\x40\x5d\xbd\xa1\x54
```

As we can see from the output, we get information about each ethernet frame so we got the following information from packet sniffing.

- Destination MAC address: it is the mac of address of the Oracle virtual box we used to sniff packets i.e. 08:00:27:43:73:BC

- Source MAC address: 52:54:00:12:35:02

- Protocol type: 8 which is ipv4 protocol

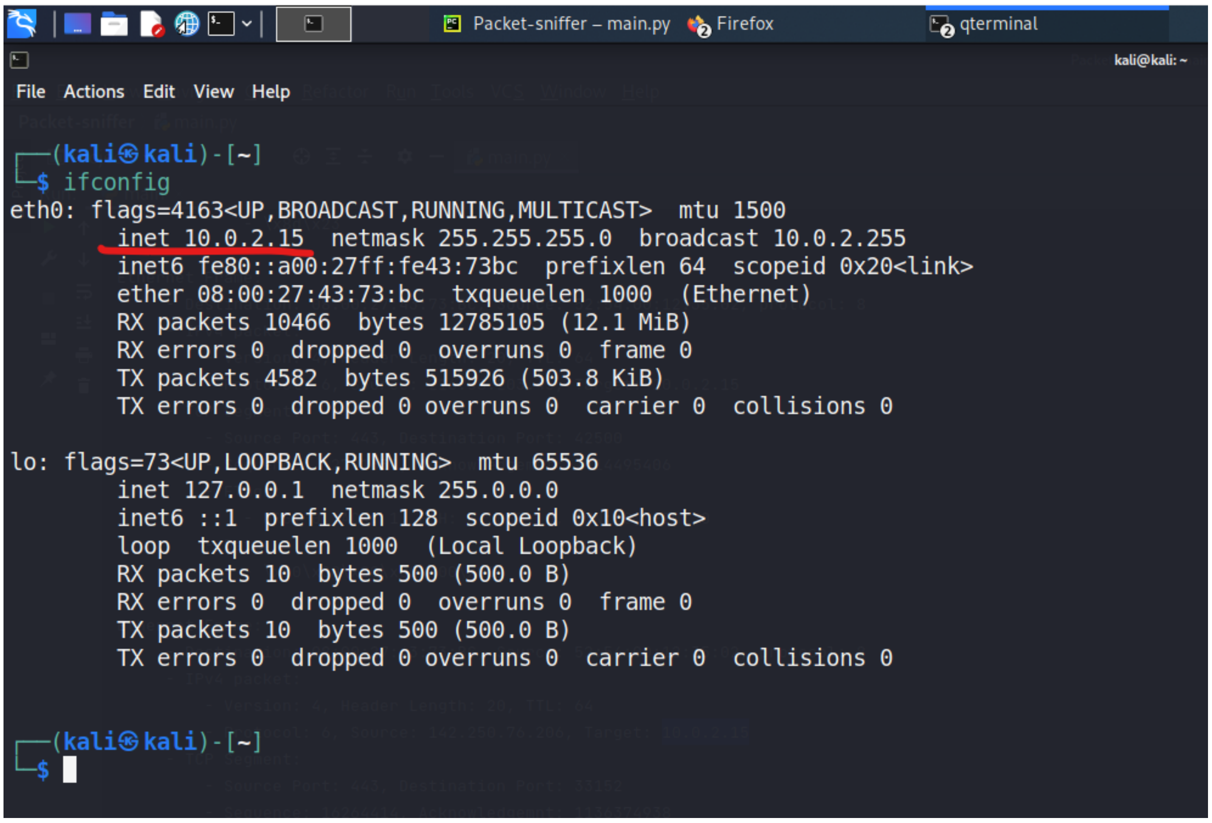
- Version of header: 4

- Header length: 20

- Time to Live: 64

- Ip source address: 142.250.76.206 which is google

- Ip destination address: 10.0.2.15 which is the VM's IP



```
┌──(kali㉿kali)-[~]
└─$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fe80::a00:27ff:fe43:73bc  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:43:73:bc  txqueuelen 1000  (Ethernet)
        RX packets 10466  bytes 12785105 (12.1 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4582  bytes 515926 (503.8 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 10  bytes 500 (500.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 10  bytes 500 (500.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0


┌──(kali㉿kali)-[~]
└─$ 
```

- Ip protocol type: 6, which is TCP as we discussed in implementation part and TCP packets information also the payload.

CONCLUSION

Packet sniffing attacks happen daily. Now and then, sniffers are using their hardware and network configuration tools to sniff data from networks. The best way to handle these attacks and avoid them is by using a private network that is both safe and secure. Also, you need to make use of protocols that have "security" as a part of its game plan.

References:

- https://en.wikipedia.org/wiki/Packet_analyzer
- https://en.wikipedia.org/wiki/Sniffing_attack
- https://www.researchgate.net/publication/267908713_Packet_Sniffing_Network_Wiretapping