**A SEMINAR REPORT ON:**

**PACKET SNIFFER IN NETWORK LAYER**



**DEPARTMENT OF INFORMATION TECHNOLOGY**

**NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA**

**YEAR-2017-18**

**COURSE TITLE**

Computer Communication and Networking (IT251)

**SUBMITTED TO:**

**Dr. Geetha V.**

(Course Instructors)

**SUBMITTED BY:**

1. Padala Preethi (15IT225)

2. Rashika Chowlek (15IT135)

3. Manasa B. (15IT219)

4. Navya W Prakash (15IT222)

**SUBMISSION DATE:**

01 MAY 2017

**CERTIFICATE**

This is to certify that the mini project entitled **“PACKET SNIFFERS IN NETWORK LAYER”** has been presented by**,** **Padala** **Preethi (15IT225), Rashika Chowlek (15IT135), Manasa B. (15IT219) and Navya W Prakash (15IT222)**, students of second year Batch (IT), Department of Information Technology, National Institute of Technology Karnataka, Surathkal, on **1st May 2017**, during the even semester of the academic year 2016 - 2017, in partial fulfilment of the requirements for the award of the degree of Bachelor of Technology in Information Technology at NITK Surathkal.

Place: Surathkal

Date: 01/05/2017           (Signature of the Examiner)

**ACKNOWLEDGEMENT**

We would like to thank each and everyone who has helped and contributed to this report. We would like to express special gratitude to our teacher Dr. Geetha V. for giving us this opportunity to do this report in Computer Communication and Networking. Your support, motivation and advice helped us to finish the report in the prescribed time. We are thankful to the Department of Information Technology for giving me this opportunity to prepare the report.

**ABSTRACT**

This project is intended to develop a tool called Packet Sniffer. The Packet Sniffer allows the computer to examine and analyze all the traffic passing by its network connection. It decodes the network traffic and makes sense of it. When it is set up on a computer, the network interface of the computer is set to promiscuous mode, listening to all the traffic on the network rather than just those packets destined for it. Packet Sniffer is a tool that sniffs without modifying the network’s packet in anyway. It merely makes a copy of each packet flowing through the network interface and finds the source and destination Ethernet addresses of the packets. It also decodes the protocols in the packets given below: ARP (Address Resolution Protocol), RARP (Reverse Address Resolution Protocol), IP (InternetProtocol), TCP (TransmissionControl Protocol), UDP (User DatagramProtocol), ICMP (Internet Control Message Protocol) and IGMP (Internet Group Message Protocol).The output is appended into normal text file, so that the network administrator can understand the network traffic and later analyze it

**TABLE OF CONTENTS**

1. Introduction 6
2. Working
   1. How packet sniffers work 7
   2. Protocols and Headers 9
3. Output 15
4. References 16
5. Appendix 17

**INTRODUCTION:**

Packet sniffing is the act of capturing packets of data flowing across a computer network. The software or device used to do this is called a packet sniffer. Packet sniffing is to computer networks what wiretapping is to a telephone network. Packet sniffing has legitimate uses to monitor network performance or troubleshoot problems with network communication. However, it is also widely used by hackers and crackers to gather information illegally about networks they intend to break into.

Using a packet sniffer it is possible to capture data like passwords, IP addresses, protocols being used on the network and other information that will help the attacker infiltrate the network. All network data travels across the Internet, and then into and out of PC's, in the form of individual, variable size data packets. Since the typical PC user never "sees" any of this raw data, many spyware systems covertly send sensitive information out of the user's computer without their knowledge. A "Packet Sniffer" is a utility that sniffs without modifying the network's packets in any way. By comparison, a firewall sees all of a computer's packet traffic as well, but it has the ability to block and drop any packets that its programming dictates. Packet sniffers merely watch, display, and log this traffic. Unfortunately, this capability allows packet sniffers to be used as potent spying tools; this is obviously not an activity that is on the good side. Today's networks are increasingly employing "switch" technology, preventing this technique from being as successful as in the past.

It is still useful, though, as it is becoming increasingly easy to install mote sniffing programs on servers and routers, through which a lot of traffic flows. Today's networks may already contain built-in sniffing modules. Most hubs support the RMON standard, which allow the intruder to sniff remotely using SNMP, which has weak authentication. Many corporations employ Network Associates "Distributed Sniffer Servers", which are set up with easy to guess passwords. Windows NT machines often have a "Network Monitoring Agent" installed, which again allows for remote sniffing.

**WORKING:**

How do Packet Sniffers Work?

Packet sniffers work by intercepting and logging network traffic that they can 'see' via the wired or wireless network interface that the packet sniffing software has access to on its host computer. On a wired network, what can be captured depends on the structure of the network. A packet sniffer might be able to see traffic on an entire network or only a certain segment of it, depending on how the network switches are configured, placed, etc.

On wireless networks, packet sniffers can usually only capture one channel at a time unless the host computer has multiple wireless interfaces that allow for multichannel capture. This packet sniffing is able to happen because Ethernet was built around an “are able to "see" all the traffic on the same wire. Thus, Ethernet hardware is built with a "filter" that ignores all traffic that doesn't belong to it. It does this by ignoring all frames whose MAC address doesn't match. A wiretap program turns off this filter, putting the Ethernet hardware into "promiscuous mode".

Since many machines may share a single Ethernet wire, each must have an individual identifier. This doesn't happen with dial-up modems, because it is assumed that any data you send to the modem is destinated for the other side of the phone line. But when you send data out onto an Ethernet wire, you have to be clear which machine you intend to send the data to. Sure, in many cases today there are only two machines talking to each other, but you have to remember that Ethernet was designed for thousands of machines to share the same wire.

Raw transmission and reception on Ethernet is governed by the Ethernet equipment. You just can't send data raw over the wire, you must first do something to it that Ethernet understands. In much the same way, you can't stick a letter in a mailbox, you must first wrap it in an envelope with an address and stamp. All hardware adapters on the wire see the frame, including the ROUTER's adapter, the packet sniffer, and any other machines. Proper adapters, however, have a hardware chip that compares the frame's "destination MAC" with its own MAC address. If they don't match, then it discards the frame. This is done at the hardware level, so the machine the adapter is attached to is completely unaware of this process. When the ROUTER Ethernet adapter sees this frame, it reads it off the wire and removes the leading 14-bytes and the trailing 4-bytes. It looks at the 0x0800 ethertype and decides to send it to the TCP/IP stack for processing (which will presumably forward it to the next router in the chain toward the destination).In general ,only the ROUTER machine is supposed to see the Ethernet frame, and all other machines are supposed to ignore it. The wiretap, however, breaks the rules and copies the frame off the network, too.

Once the raw packet data is captured, the packet sniffing software must analyze it and present it in human-readable form so that the person using the packet sniffing software can make sense of it. The person analyzing the data can view details of the 'conversation' happening between two or more nodes on the network.

Network technicians can use this information to determine where a fault lies, such as determining which device failed to respond to a network request.

In this project,a protocol sniffer is implemented in the network layer . This method involves sniffing data related to the network protocols being used. First, a list of protocols in the network layer are analysed and the contents of the packet are printed in human readable form in a sniffreLogger.txt which is created based on the captured data. Thus this is further segregated to create special sniffers for each attack. For example, in a network sniff capture, if the ICMP protocol is not seen, it is assumed to be blocked. However, if UDP packets are seen, a separate UDP sniffer is started to capture and decipher Telnet, PPP, DNS and other related application details.

In order to achieve this first of all a raw socket is created .In standard sockets, the payload is automatically encapsulated according to the chosen transport layer protocol and the socket-user is unaware of the existence of protocol headers that are broadcast with the payload. When reading from a raw socket, the headers are usually included. When transmitting packets from a raw socket, the automatic addition of a header is optional. Raw sockets are used in security related applications like nmap. One possible use case for raw sockets is the implementation of new transport-layer protocols in user space. Raw sockets are typically available in network equipment, and used for routing protocols such as the Internet Group Management Protocol and in the Internet Control Message Protocol (ICMP).

***sock\_raw = socket( AF\_PACKET , SOCK\_RAW , htons(ETH\_P\_ALL))***

To receive packets we use a function called recvfrom() that calls are used to receive messages from a socket, and may be used to receive data on a socket whether or not it is connection-oriented.

**ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags, struct sockaddr \*src\_addr, socklen\_t \*addrlen)**

After the packet is received it is parsed where the ip->protocol value is checked in order to identify the necessary protocol. In this project implementation, the used protocols are as following

switch(ip->protocol)

case 1: //ICMP Protocol

case 2://IGMP Protocol

case 6: //TCP Protocol

case 17: //UDP Protocol

case 54://ARP Protocol

case 91://DHCP protocol

default: //Some Other Protocol like ARP etc.

end switch

After the Protocol of the received packet is known the headers of the particular protocol along with Ethernet header and IP header are extracted using the standard format of headers given and printed in the sniffrelog.txt file in an intended way.

Thus the features of this sniffer created are

1. Sniff both incoming and outgoing traffic.

2. Sniff ALL ETHERNET FRAMES , which includes all kinds of IP packets and even more if there are any.

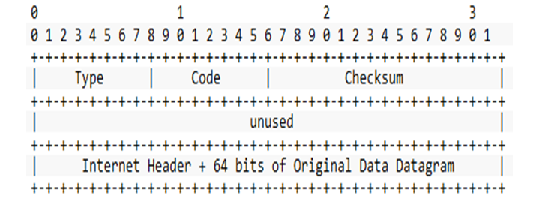
3. Provides the Ethernet headers too , which contain the Mac addresses.

PROTOCOLS AND HEADERS:

ICMP:

The **Internet Control Message Protocol** (**ICMP**) is a supporting protocol in the Internet protocol suite. It is used by network devices, including routers, to send error messages and operational information indicating, for example, that a requested service is not available or that a host or router could not be reached. ICMP differs from transport protocols such as TCP and UDP in that it is not typically used to exchange data between systems, nor is it regularly employed by end-user network applications (with the exception of some diagnostic tools like ping and traceroute).The ICMP packet is then encapsulated in an IPv4 packet.The packet consists of header and data sections.

The ICMP header starts after the IPv4 header and is identified by IP protocol number '1'. All ICMP packets have an 8-byte header and variable-sized data section. The first 4 bytes of the header have fixed format, while the last 4 bytes depend on the type/code of that ICMP packet.



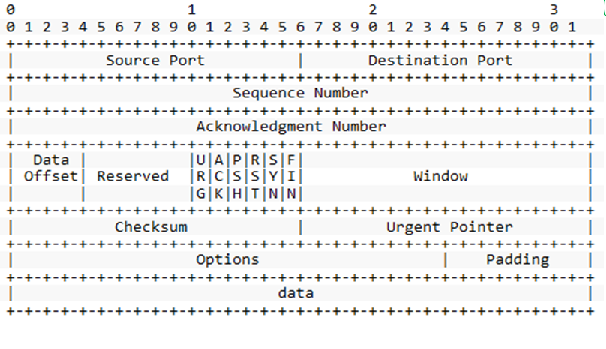
**Fig 1: ICMP header**

**TCP:**

The **Transmission Control Protocol** (**TCP**) is one of the main protocols of the Internet protocol suite. It originated in the initial network implementation in which it complemented the Internet Protocol (IP). Therefore, the entire suite is commonly referred to as **TCP/IP**. TCP provides reliable, ordered, and error-checked delivery of a stream of octets between applications running on hosts communicating by an IP network. Major Internet applications such as the World Wide Web, email, remote administration, and file transfer rely on TCP. Applications that do not require reliable data stream service may use the User Datagram Protocol (UDP), which provides a connectionless datagram service that emphasizes reduced latency over reliability.

Transmission Control Protocol accepts data from a data stream, divides it into chunks, and adds a TCP header creating a TCP segment. The TCP segment is then encapsulated into an Internet Protocol (IP) datagram, and exchanged with peers. The term *TCP packet* appears in both informal and formal usage, whereas in more precise terminology *segment* refers to the TCP protocol data unit (PDU), *datagram*to the IP PDU, and *frame* to the data link layer PDU:

Processes transmit data by calling on the TCP and passing buffers of data as arguments. The TCP packages the data from these buffers into segments and calls on the internet module [e.g. IP] to transmit each segment to the destination TCP.A TCP segment consists of a segment *header* and a *data* section. The TCP header contains 10 mandatory fields, and an optional extension field (*Options*, pink background in table). The data section follows the header. Its contents are the payload data carried for the application. The length of the data section is not specified in the TCP segment header. It can be calculated by subtracting the combined length of the TCP header and the encapsulating IP header from the total IP datagram length (specified in the IP header).



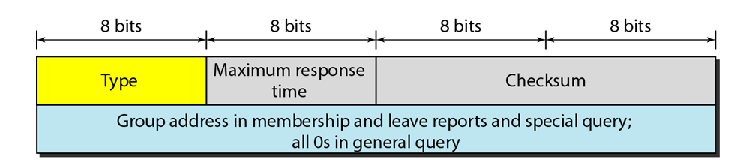
**Fig 2: TCP header**

**IGMP:**

The **Internet Group Management Protocol** (**IGMP**) is a communications protocol used by hosts and adjacent routers on IPv4 networks to establish multicast group memberships. IGMP is an integral part of IP multicast.

IGMP can be used for one-to-many networking applications such as online streaming video and gaming, and allows more efficient use of resources when supporting these types of applications. IGMP is used on IPv4 networks. Multicast management on IPv6 networks is handled by Multicast Listener Discovery (MLD) which uses ICMPv6 messaging in contrast to IGMP's bare IP encapsulation. IGMP messages are carried in bare IP packets with IP protocol number 2.There is no transport layer used with IGMP messaging, similar to the Internet Control Message Protocol.

There are several types of IGMP messages: Membership Queries (general and group-specific), Membership Reports, and Leave Group messages. Membership Queries are sent by multicast routers to determine which multicast addresses are of interest to systems attached to its network. Routers periodically send General Queries to refresh the group membership state for all systems on its network. Group-Specific Queries are used for determining the reception state for a particular multicast address. Group-and-Source-Specific Queries allow the router to determine if any systems desire reception of messages sent to a multicast group from a source address specified in a list of unicast addresses.

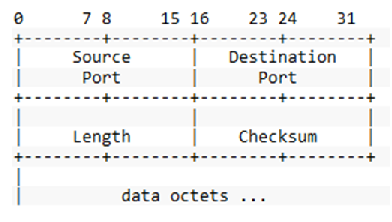


**Fig 3: IGMP header**

**UDP:**

UDP uses a simple connectionless transmission model with a minimum of protocol mechanism. UDP provides checksums for data integrity, and port numbers for addressing different functions at the source and destination of the datagram. It has no handshaking dialogues, and thus exposes the user's program to any unreliability of the underlying network: there is no guarantee of delivery, ordering, or duplicate protection. If error-correction facilities are needed at the network interface level, an application may use the Transmission Control Protocol (TCP) or Stream Control Transmission Protocol (SCTP) which are designed for this purpose.

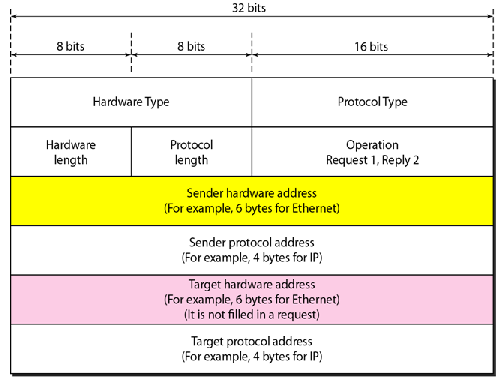
UDP is suitable for purposes where error checking and correction are either not necessary or are performed in the application; UDP avoids the overhead of such processing at the level of the network interface. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets, which may not be an option in a real-time system.



**Fig 4: UDP header**

**ARP:**

ARP is used for mapping a network address to a physical address like an Ethernet address (also named a MAC address). ARP has been implemented with many combinations of network and data link layer technologies, like IPv4, Chaosnet, DECnet and Xerox PARC Universal Packet (PUP) using IEEE 802 standards, FDDI, X.25, Frame Relay and Asynchronous Transfer Mode (ATM). IPv4 over IEEE 802.3 and IEEE 802.11 is the most common usage.



**Fig 5: ARP header**

**IP:**

The **Internet Protocol** (**IP**) is the principal communications protocol in the Internet protocol suite for relaying datagrams across network boundaries. Its routing function enables internetworking, and essentially establishes the Internet.

IP has the task of delivering packets from the source host to the destination host solely based on the IP addresses in the packet headers. For this purpose, IP defines packet structures that encapsulate the data to be delivered. It also defines addressing methods that are used to label the datagram with source and destination information. Each datagram has two components: a header and a payload. The IP header is tagged with the source IP address, the destination IP address, and other meta-data needed to route and deliver the datagram. The payload is the data that is transported. This method of nesting the data payload in a packet with a header is called encapsulation.

### IP.png

**Fig 6: IP header**

**ETHERNET:**

**Ethernet** is a family of computer networking technologies commonly used in local area networks (LAN), metropolitan area networks (MAN) and wide area networks (WAN). It was commercially introduced in 1980 and first standardized in 1983 as IEEE 802.3, and has since been refined to support higher bit rates and longer link distances. Over time, Ethernet has largely replaced competing wired LAN technologies such as token ring, FDDI and ARCNET. Systems communicating over Ethernet divide a stream of data into shorter pieces called frames. Each frame contains source and destination addresses, and error-checking data so that damaged frames can be detected and discarded; most often, higher-layer protocols trigger retransmission of lost frames. As per the OSI model, Ethernet provides services up to and including the data link layer.

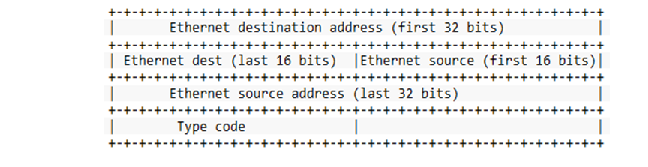
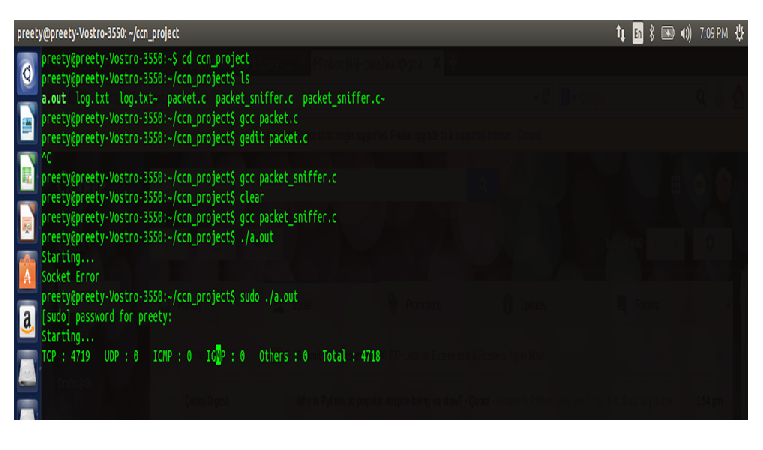
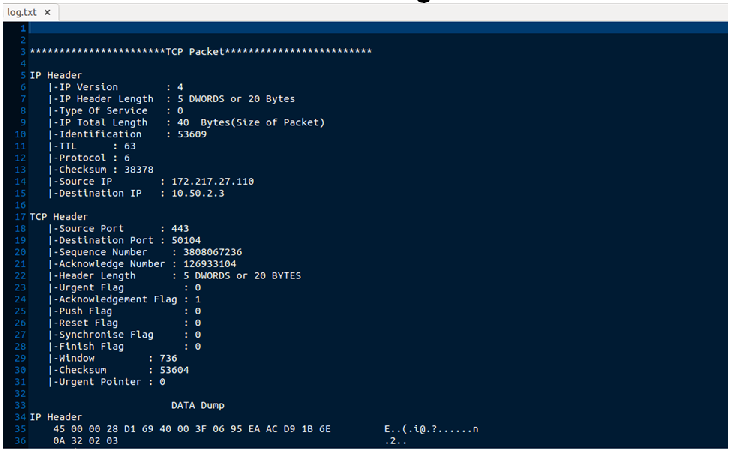


Fig 7: Ethernet header

**OUTPUT:**





**REFERENCES:**

* https://www.lifewire.com/what-is-a-packet-sniffer-248731
* https://en.wikipedia.org/wiki/Packet\_analyzer
* https://en.wikipedia.org/wiki/Ethernet\_frame
* https://en.wikipedia.org/wiki/Internet\_Protocol
* https://en.wikipedia.org/wiki/Address\_Resolution\_Protocol
* https://en.wikipedia.org/wiki/Internet\_Group\_Management\_Protocol

**APPENDIX:**

//to implement packet sniffer in network layer using raw sockets  
  
#include<netinet/in.h>  
#include<stdio.h> //For standard things  
#include<stdlib.h> //malloc  
#include<string.h> //strlen  
#include<netinet/ip\_icmp.h> //Provides declarations for icmp header  
#include<netinet/udp.h> //Provides declarations for udp header  
#include<netinet/tcp.h> //Provides declarations for tcp header  
#include<netinet/ip.h>  
#include<netinet/if\_ether.h> //For ETH\_P\_ALL  
#include<net/ethernet.h> //For ether\_header  
#include<sys/socket.h>  
#include<arpa/inet.h>  
#include<sys/time.h>  
  
void ProcessPacket(unsigned char\* , int);  
void print\_ip\_header(unsigned char\* , int);  
void print\_tcp\_packet(unsigned char \* , int );  
void print\_udp\_packet(unsigned char \* , int );  
void print\_icmp\_packet(unsigned char\* , int );  
void print\_arp\_packet(unsigned char\*, int );  
void PrintData (unsigned char\* , int);  
  
FILE \*logfile;  
FILE \*fptr;  
  
struct sockaddr\_in source, dest;  
int tcp=0,udp=0,icmp=0,igmp=0,arp=0,total=0,dhcp=0,i,j;  
  
int main()  
{  
 int saddr\_size , data\_size;  
 struct sockaddr saddr;  
 fptr=fopen("sniffer.csv","w");  
 if(fptr==NULL)  
  {   
  printf("Error!");  
  exit(1);  
  }  
 fprintf(fptr,"TCP,UDP,ICMP,ARP,DHCP,Total\n");  
 unsigned char \*buffer = (unsigned char \*) malloc(65536); //Its Big!  
 logfile=fopen("snifferlog.txt","w");  
 if(logfile==NULL)  
 {  
 printf("Unable to create log.txt file.");  
 }  
  
printf("SNiffing Starting...\n");  
  
int sock\_raw = socket( AF\_PACKET , SOCK\_RAW , htons(ETH\_P\_ALL)) ;  
 if(sock\_raw < 0)  
 {  
 //Print the error with proper message  
 perror("Socket Error");  
 return 1;  
 }  
 while(1)  
 {  
 saddr\_size = sizeof saddr;  
 //Receive a packet  
 data\_size = recvfrom(sock\_raw , buffer , 65536 , 0 , &saddr , (socklen\_t\*)&saddr\_size);  
 if(data\_size <0 )  
 {  
 printf("Recvfrom error , failed to get packets\n");  
 return 1;  
 }  
 //Now process the packet  
 ProcessPacket(buffer , data\_size);  
 }  
 fclose(fptr);  
 close(sock\_raw);  
 printf("Finished");  
 return 0;  
}  
  
void ProcessPacket(unsigned char\* buffer, int size)  
{  
  
 //Get the IP Header part of this packet , excluding the ethernet header  
 struct iphdr \*iph = (struct iphdr\*)(buffer + sizeof(struct ethhdr));  
 ++total;  
 switch (iph->protocol) //Check the Protocol and do accordingly...  
 {  
 case 1: //ICMP Protocol  
        ++icmp;  
        print\_icmp\_packet( buffer , size);  
         break;  
 case 2://IGMP Protocol  
        ++igmp;  
        break;  
case 6: //TCP Protocol  
       ++tcp;   
       print\_tcp\_packet(buffer , size);  
       break;  
case 17: //UDP Protocol  
       ++udp;  
       print\_udp\_packet(buffer , size);  
       break;  
case 54:  
        ++arp;  
        print\_arp\_packet(buffer , size);  
        break;  
case 91:  
              ++dhcp;  
        break;  
default: //Some Other Protocol like ARP etc.  
      break;  
}  
  
 printf(" TCP:%d ,UDP:%d ,ICMP:%d ,IGMP:%d ,ARP:%d ,DHCP:%d,TOTAL:%d\r", tcp , udp , icmp , igmp , arp ,dhcp, total);  
 fprintf(fptr," %d ,%d ,%d ,%d ,%d ,%d,%d\n", tcp , udp , icmp ,igmp , arp ,dhcp, total);  
}  
  
void print\_ethernet\_header(unsigned char\* Buffer, int Size)  
{  
 struct ethhdr \*eth = (struct ethhdr \*)Buffer;  
 fprintf(logfile , "\n");  
 fprintf(logfile , "Ethernet Header\n");  
  fprintf(logfile , " |-Destination Address : %.2X-%.2X-%.2X-%.2X-%.2X-%.2X \n", eth->h\_dest[0] , eth->h\_dest[1] , eth->h\_dest[2] , eth->h\_dest[3] , eth->h\_dest[4] , eth->h\_dest[5] );  
 fprintf(logfile , " |-Source Address : %.2X-%.2X-%.2X-%.2X-%.2X-%.2X \n", eth->h\_source[0] , eth->h\_source[1] , eth->h\_source[2] , eth->h\_source[3] , eth->h\_source[4] , eth->h\_source[5] );  
 fprintf(logfile , " |-Protocol : %u \n",(unsigned short)eth->h\_proto);  
}  
  
void print\_ip\_header(unsigned char\* Buffer, int Size)  
{  
 print\_ethernet\_header(Buffer , Size);  
 unsigned short iphdrlen;  
 struct iphdr \*iph = (struct iphdr \*)(Buffer + sizeof(struct ethhdr) );  
 iphdrlen =iph->ihl\*4;  
 memset(&source, 0, sizeof(source));  
 source.sin\_addr.s\_addr = iph->saddr;  
 memset(&dest, 0, sizeof(dest));  
 dest.sin\_addr.s\_addr = iph->daddr;  
   
 fprintf(logfile , "\n");  
 fprintf(logfile , "IP Header\n");  
 fprintf(logfile , " |-IP Version : %d\n",(unsigned int)iph->version);  
 fprintf(logfile , " |-IP Header Length : %d DWORDS or %d Bytes\n",(unsigned int)iph->ihl,((unsigned int)(iph->ihl))\*4);  
 fprintf(logfile , " |-Type Of Service : %d\n",(unsigned int)iph->tos);  
 fprintf(logfile , " |-IP Total Length : %d Bytes(Size of Packet)\n",ntohs(iph->tot\_len));  
 fprintf(logfile , " |-Identification : %d\n",ntohs(iph->id));  
 fprintf(logfile , " |-TTL : %d\n",(unsigned int)iph->ttl);  
 fprintf(logfile , " |-Protocol : %d\n",(unsigned int)iph->protocol);  
 fprintf(logfile , " |-Checksum : %d\n",ntohs(iph->check));  
 fprintf(logfile , " |-Source IP : %s\n",inet\_ntoa(source.sin\_addr));  
 fprintf(logfile , " |-Destination IP : %s\n",inet\_ntoa(dest.sin\_addr));  
}  
  
void print\_tcp\_packet(unsigned char\* Buffer, int Size)  
{  
 unsigned short iphdrlen;  
 struct iphdr \*iph = (struct iphdr \*)( Buffer + sizeof(struct ethhdr) );  
 iphdrlen = iph->ihl\*4;  
 struct tcphdr \*tcph=(struct tcphdr\*)(Buffer + iphdrlen + sizeof(struct ethhdr));  
 int header\_size = sizeof(struct ethhdr) + iphdrlen + tcph->doff\*4;  
   
 fprintf(logfile , "\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*TCP Packet\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 print\_ip\_header(Buffer,Size);  
 fprintf(logfile , "\n");  
 fprintf(logfile , "TCP Header\n");  
 fprintf(logfile , " |-Source Port : %u\n",ntohs(tcph->source));  
 fprintf(logfile , " |-Destination Port : %u\n",ntohs(tcph->dest));  
 fprintf(logfile , " |-Sequence Number : %u\n",ntohl(tcph->seq));  
 fprintf(logfile , " |-Acknowledge Number : %u\n",ntohl(tcph->ack\_seq));  
 fprintf(logfile , " |-Header Length : %d DWORDS or %d BYTES\n" ,(unsigned int)tcph->doff,(unsigned int)tcph->doff\*4);  
 //fprintf(logfile , " |-CWR Flag : %d\n",(unsigned int)tcph->cwr);  
 //fprintf(logfile , " |-ECN Flag : %d\n",(unsigned int)tcph->ece);  
 fprintf(logfile , " |-Urgent Flag : %d\n",(unsigned int)tcph->urg);  
 fprintf(logfile , " |-Acknowledgement Flag : %d\n",(unsigned int)tcph->ack);  
 fprintf(logfile , " |-Push Flag : %d\n",(unsigned int)tcph->psh);  
 fprintf(logfile , " |-Reset Flag : %d\n",(unsigned int)tcph->rst);  
 fprintf(logfile , " |-Synchronise Flag : %d\n",(unsigned int)tcph->syn);  
 fprintf(logfile , " |-Finish Flag : %d\n",(unsigned int)tcph->fin);  
 fprintf(logfile , " |-Window : %d\n",ntohs(tcph->window));  
 fprintf(logfile , " |-Checksum : %d\n",ntohs(tcph->check));  
 fprintf(logfile , " |-Urgent Pointer : %d\n",tcph->urg\_ptr);  
 fprintf(logfile , "\n");  
 fprintf(logfile , " DATA Dump ");  
 fprintf(logfile , "\n");  
 fprintf(logfile , "IP Header\n");  
 PrintData(Buffer,iphdrlen);  
 fprintf(logfile , "TCP Header\n");  
 PrintData(Buffer+iphdrlen,tcph->doff\*4);  
 fprintf(logfile , "Data Payload\n");  
 PrintData(Buffer + header\_size , Size - header\_size );  
 fprintf(logfile , "\n###########################################################");  
 }  
  
void print\_arp\_packet(unsigned char\* Buffer, int Size)   
 {   
 unsigned short iphdrlen;  
 struct iphdr \*iph = (struct iphdr \*)( Buffer + sizeof(struct ethhdr) );  
 iphdrlen = iph->ihl\*4;  
 struct ether\_arp \*arpheader=(struct ether\_arp\*)(Buffer + iphdrlen + sizeof(struct ethhdr));  
  
 fprintf(logfile , "\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ARP Packet\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 /\* If is Ethernet and IPv4, print packet contents \*/  
 fprintf(logfile,"Sender MAC: ");  
 for(i=0; i<6;i++)  
  fprintf(logfile,"%02X:", arpheader->arp\_sha[i]);  
  fprintf(logfile,"\nSender IP: ");  
 for(i=0; i<4;i++)  
  fprintf(logfile,"%d.", arpheader->arp\_spa[i]);  
  fprintf(logfile,"\nTarget MAC: ");  
 for(i=0; i<6;i++)  
  fprintf(logfile,"%02X:", arpheader->arp\_tha[i]);  
  fprintf(logfile,"\nTarget IP: ");  
 for(i=0; i<4; i++)  
  fprintf(logfile,"%d.", arpheader->arp\_tpa[i]);  
  fprintf(logfile,"\n");  
 fprintf(logfile,"\n###########################################################");  
 fprintf(logfile , "\n###########################################################");  
}  
  
void print\_udp\_packet(unsigned char \*Buffer , int Size)  
 {  
 unsigned short iphdrlen;  
 struct iphdr \*iph = (struct iphdr \*)(Buffer + sizeof(struct ethhdr));  
 iphdrlen = iph->ihl\*4;  
 struct udphdr \*udph = (struct udphdr\*)(Buffer + iphdrlen + sizeof(struct ethhdr));  
 int header\_size = sizeof(struct ethhdr) + iphdrlen + sizeof udph;  
   
 fprintf(logfile , "\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*UDP Packet\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");   
 print\_ip\_header(Buffer,Size);  
 fprintf(logfile , "\nUDP Header\n");  
 fprintf(logfile , " |-Source Port : %d\n" , ntohs(udph->source));  
 fprintf(logfile , " |-Destination Port : %d\n" , ntohs(udph->dest));  
 fprintf(logfile , " |-UDP Length : %d\n" , ntohs(udph->len));  
 fprintf(logfile , " |-UDP Checksum : %d\n" , ntohs(udph->check));  
 fprintf(logfile , "\n");  
 fprintf(logfile , "IP Header\n");  
 PrintData(Buffer , iphdrlen);  
 fprintf(logfile , "UDP Header\n");  
 PrintData(Buffer+iphdrlen , sizeof udph);  
 fprintf(logfile , "Data Payload\n");  
 //Move the pointer ahead and reduce the size of string  
 PrintData(Buffer + header\_size , Size - header\_size);  
 fprintf(logfile , "\n###########################################################");  
 }  
  
void print\_icmp\_packet(unsigned char\* Buffer , int Size)  
 {  
 unsigned short iphdrlen;  
 struct iphdr \*iph = (struct iphdr \*)(Buffer + sizeof(struct ethhdr));  
 iphdrlen = iph->ihl \* 4;  
 struct icmphdr \*icmph = (struct icmphdr \*)(Buffer + iphdrlen + sizeof(struct ethhdr));  
 int header\_size = sizeof(struct ethhdr) + iphdrlen + sizeof icmph;  
 fprintf(logfile , "\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ICMP Packet\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  
 print\_ip\_header(Buffer , Size);  
 fprintf(logfile , "\n");  
 fprintf(logfile , "ICMP Header\n");  
 fprintf(logfile , " |-Type : %d",(unsigned int)(icmph->type));  
 if((unsigned int)(icmph->type) == 11)  
 {  
  fprintf(logfile , " (TTL Expired)\n");  
 }  
 else if((unsigned int)(icmph->type) == ICMP\_ECHOREPLY)  
 {  
  fprintf(logfile , " (ICMP Echo Reply)\n");  
 }  
 fprintf(logfile , " |-Code : %d\n",(unsigned int)(icmph->code));  
 fprintf(logfile , " |-Checksum : %d\n",ntohs(icmph->checksum));  
 //fprintf(logfile , " |-ID : %d\n",ntohs(icmph->id));  
 //fprintf(logfile , " |-Sequence : %d\n",ntohs(icmph->sequence));  
 fprintf(logfile , "\n");  
 fprintf(logfile , "IP Header\n");  
 PrintData(Buffer,iphdrlen);  
 fprintf(logfile , "UDP Header\n");  
 PrintData(Buffer + iphdrlen , sizeof icmph);  
 fprintf(logfile , "Data Payload\n");  
 //Move the pointer ahead and reduce the size of string  
 PrintData(Buffer + header\_size , (Size - header\_size) );  
 fprintf(logfile , "\n###########################################################");  
 }  
  
void PrintData (unsigned char\* data , int Size)   
 {  
 int i , j;  
 for(i=0 ; i < Size ; i++)  
 {  
  if( i!=0 && i%16==0) //if one line of hex printing is complete...  
   {  
   fprintf(logfile , " ");  
   for(j=i-16 ; j<i ; j++)   
    {  
    if(data[j]>=32 && data[j]<=128)  
     fprintf(logfile , "%c",(unsigned char)data[j]); //if its a number or alphabet  
    else fprintf(logfile , "."); //otherwise print a dot  
    }  
  fprintf(logfile , "\n");  
  }  
  if(i%16==0) fprintf(logfile , " ");  
   fprintf(logfile , " %02X",(unsigned int)data[i]);  
  if( i==Size-1) //print the last spaces   
   {  
     for(j=0;j<15-i%16;j++)   
      {  
      fprintf(logfile , " "); //extra spaces  
       }  
      fprintf(logfile , " ");  
    for(j=i-i%16 ; j<=i ; j++)  
    {  
     if(data[j]>=32 && data[j]<=128)  
     {  
       fprintf(logfile , "%c",(unsigned char)data[j]);  
     }  
     else  
      {  
      fprintf(logfile , ".");   
     }   
   }  
  fprintf(logfile , "\n" );  
 }  
 }  
}