Network Packet Sniffing methods, **SimpSniff**

Sidharth Juyal

05/MCA/20
MCA vith semester
Kumaon Engineering College

sidharth_juyal@rediffmail.com

Abstract

This project intends to give basic introduction to network packet sniffers.

Thus, we will first try to recall some basic network concepts, as well as some standard protocols. Then, we will explain you the contexts in which sniffers evolve and we will mention several sniffing methods. Finally, we will demonstrate with SimpSniff, how libcap (a freely available library for capturing packets) can be used to write a small and efficient sniffing tool.

This project can be further easily scaled to support many other TCP/IP Network model based protocols like FTP, POP3 etc, or even to other network models like IPX, AppleTalk, etc.

The practical purpose of this project is to provide monitoring ability to the administrator of a network for various purposes like diagnosis of network usage, etc. This project can also be deployed at home PCs or laptops by users or parents to monitor the internet access.

Contents

Introduction

1.1 Purpose and Scope

This project intends to explain the working procedure of a network packet sniffer and its application areas.

We will start by first reviewing some basic networking concepts.
Then, we will answer several questions related to packet sniffing:

What is a network packet sniffer?
How does a sniffer work?
What kind of information does a sniffer provide?
Why should we use a sniffer?
What are some available sniffers?
Finally, we will introduce SimpSniff, a sniffing tool developed to analyze packets in a network.

1.2 Basic Network Concepts

1.2.1 Network notions

A computer network is made up of several computers inter-connected to each other using some medium (wires, network cards and other equipments which ensure proper flow of data). We call physical topology the actual physical layout of a network or how it looks, and its physical connection scheme.

There are three topologies most widely used

A. Star Topology
B. Ring Topology and
C. Bus Topology.

There are also combinations of the three which yield even more possible topologies.

We make a clear distinction between physical and logical topology.

Logical topology: represents the way data transit in the wires. The main logical topologies are Ethernet, Token Ring and FDDI.

Physical topology: represents the way various computers are physically interconnected.

Bus topology is the simplest network organization. In this network layout, one trunk cable provides the main path of communications. Each node is then connected to the bus. Bus topology main advantage is the simplicity to deploy a network based on that technology. Yet, this type of topology is also very poor at handling cable problems as all of the stations beyond the affected area will be brought down.

Star topology is a physical layout of a network in which computers are connected to a hub or a switch. Networks founded on star topology are less vulnerable than networks founded on bus topology since they allow easy fault isolation of a single station. Yet, failure a the hub or switch will bring the entire network segment down.

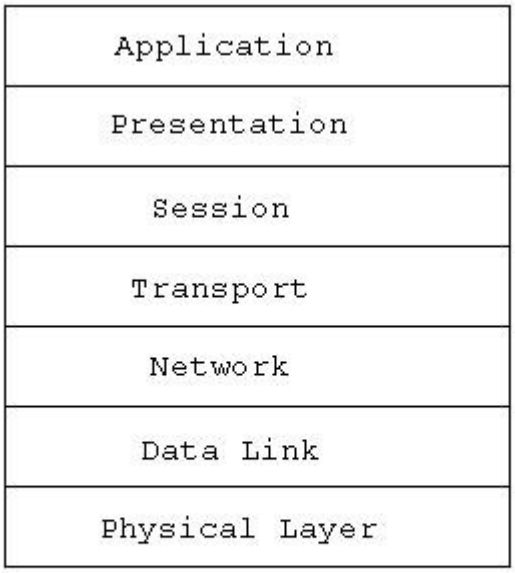
Ring topology is a network design where data is passed from one station to another in a closed circular loop. In fact, stations in a network founded on ring topology are not really linked to each other in a ring way, they are linked to a Multistation Access Unit which will organize data flow between computers.

1.2.2 OSI Model

The International Standards Organization (ISO) developed a model that would allow the sending and receiving of data between computers. It works on a layer approach where each layer is responsible for performing a set of functions.

Layered Approach:

The OSI model consists of seven layers, each of which can (and typically does) have several sublayers.
The upper layers of the OSI model (application, presentation, and session—Layers 7, 6, and 5) are oriented more toward services to the applications. The lower four layers (transport, network, data link, and physical—Layers 4, 3, 2, and 1) are oriented more toward the flows of data from end to end through the network.



Various Layers of OSI Networking Architecture

Now we discuss each layer in brief:

A. Layer 7: Application

Functional Description:

An application that communicates with other computers is implementing OSI application layer concepts. The application layer refers to communications services to applications. For example, a word processor that lacks communications capabilities would not implement code for communications, and word processor programmers would not be concerned about OSI Layer 7. However, if an option for transferring a file were added, then the word processor would need to implement OSI Layer 7 (or the equivalent layer in another protocol specification).

Example Protocols:

Telnet, HTTP, FTP,
WWW browsers, NFS,
SMTP gateways
(Eudora, CC:mail),
SNMP, X.400 mail,
FTAM

B. Layer 6: Presentation

Functional Description:

This layer's main purpose is defining data formats, such as ASCII text, EBCDIC text, binary, BCD, and JPEG. Encryption is also defined by OSI as a presentation layer service. For example, FTP enables you to choose binary or ASCII transfer. If binary is selected, the sender and receiver do not modify the contents of the file. If ASCII is chosen, the sender translates the text from the sender's character set to a standard ASCII and sends the data. The receiver translates back from the standard ASCII to the character set used on the receiving computer.

Example Protocols:

JPEG, ASCII, EBCDIC,
TIFF, GIF, PICT,
encryption, MPEG,
MIDI

C. Layer 5: Session

Functional Description:

The session layer defines how to start, control, and end conversations (called sessions). This includes the control and management of multiple bidirectional messages so that the application can be notified if only some of a series of messages are completed. This allows the presentation layer to have a seamless view of an incoming stream of data. The presentation layer can be presented with data if all flows occur in some cases. For example, an automated teller machine transaction in which you withdraw cash from your checking account should not debit your account, and then fail, before handing you the cash, recording the transaction even though you did not receive money. The session layer creates ways to imply which flows are part of the same session and which flows must complete before any are considered complete.

Example Protocols:

RPC, SQL, NFS,
NetBios names,
AppleTalk ASP, DECnet
SCP

D. Layer 4: Transport

Functional Description:

Layer 4 includes the choice of protocols that either do or do not provide error recovery. Multiplexing of incoming data for different flows to applications on the same host (for example, TCP sockets) is also performed. Reordering of the incoming data stream when packets arrive out of order is included.

Example Protocols:

TCP, UDP, SPX

E. Layer 3: Network

Functional Description:

This layer defines end-to-end delivery of packets. To accomplish this, the network layer defines logical addressing so that any endpoint can be identified. It also defines how routing works and how routes are learned so that the packets can be delivered. The network layer also defines how to fragment a packet into smaller packets to accommodate media with smaller maximum transmission unit sizes. (Note: Not all Layer 3 protocols use fragmentation.) The network layer of OSI defines most of the details that a Cisco router considers when routing. For example, IP running in a Cisco router is responsible for examining the destination IP address of a packet, comparing that address to the IP routing table, fragmenting the packet if the outgoing interface requires smaller packets, and queuing the packet to be sent out to the interface.

Example Protocols:

IP, IPX, AppleTalk DDP

F. Layer 2: Data link

Functional Description:

The data link (Layer 2) specifications are concerned with getting data across one particular link or medium. The data link protocols define delivery across an individual link. These protocols are necessarily concerned with the type of media in question; for example, 802.3 and 802.2 are specifications from the IEEE, which are referenced by OSI as valid data link (Layer 2) protocols. These specifications define how Ethernet works. Other protocols, such as High-Level Data Link Control (HDLC) for a point-to-point WAN link, deal with the different details of a WAN link. As with other protocol specifications, OSI often does not create any original specification for the data link layer but instead relies on other standards bodies such as IEEE to create new standards for the data link layer and the physical layer.

Example Protocols:

IEEE 802.3/802.2,
HDLC, Frame Relay,
PPP, FDDI, ATM, IEEE
802.5/ 802.2

G. Layer 1: Physical

Functional Description:

These physical layer (Layer 1) specifications, which are also typically standards from other organizations that are referred to by OSI, deal with the physical characteristics of the transmission medium. Connectors, pins, use of pins, electrical currents, encoding, and light modulation are all part of different physical layer specifications. Multiple specifications are sometimes used to complete all details of the physical layer. For example, RJ-45 defines the shape of the connector and the number of wires or pins in the cable. Ethernet and 802.3 define the use of wires or pins 1, 2, 3, and 6. So, to use a category 5 cable, with an RJ-45 connector for an Ethernet connection, Ethernet and RJ-45 physical layer specifications are used.

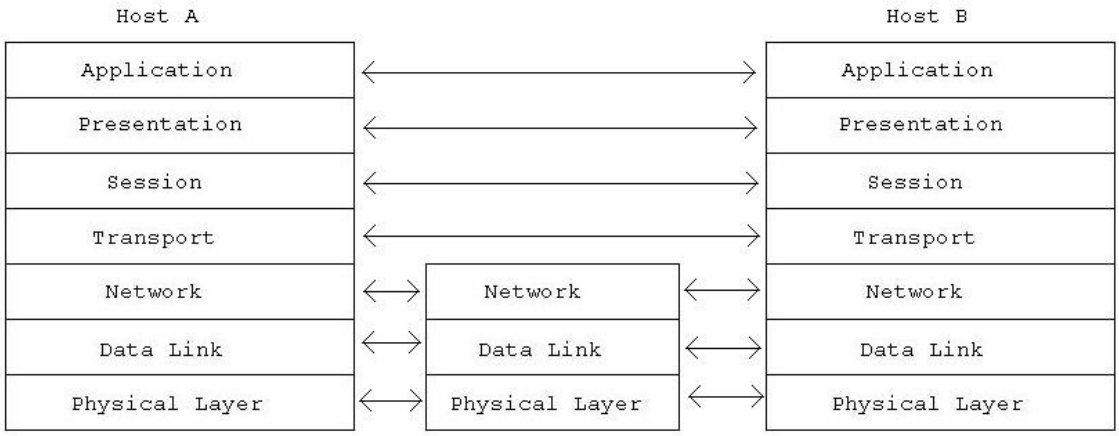
Example Protocols:

EIA/TIA-232, V.35,
EIA/TIA- 449, V.24,
RJ45, Ethernet, 802.3,
802.5, FDDI, NRZI,
NRZ, B8ZS

Some protocols define details of multiple layers. For example, because the TCP/IP application layer correlates to OSI Layers 5 through 7, the Network File System (NFS) implements elements matching all three layers. Likewise, the 802.3, 802.5, and Ethernet standards define details for the data link and physical layers.

Interactions Between Adjacent Layers on the Same Computer:

To provide services to the next higher layer, a layer must know about the standard interfaces defined between layers. These interfaces include definitions of what Layer N+1 must provide to Layer N to get services, as well as what information Layer N must provide back to Layer N+1.



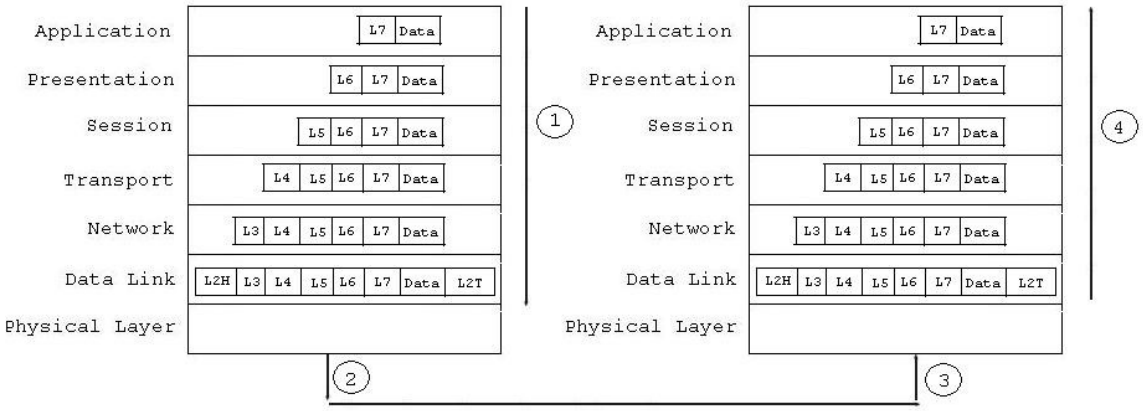
Interaction among various layers

A layer is created where a different level of abstraction is needed.
A layer performs a specific function.
The layer boundaries are chosen to minimize the information flaw across the interfaces.

Encapsulation: The seven OSI layers use various forms of control information to communicate with peer layers in other computer systems. This control information consists of specific requests and instructions that are exchanged between peer OSI layers. Control information typically takes one of two forms:

A. Headers are prepended to the data passed down from upper layers.
B. Trailers are appended to data passed down from upper layers.

System B receives the request on Layer 1, and begins the decapsulation process, stripping the Layer 1 headers and footers off to reveal the Layer 2 information, and so forth, all the way up to the 7th layer.



Layer Interaction by appending corresponding Headers

1.2.3 Network Protocols

Following are common protocols encountered while sniffing packets over a network:

ARP: Address Resolution Protocol
RARP: Reverse Address Resolution Protocol
BOOTP: Bootstrap Protocol
DHCP: Dynamic Host Configuration Protocol
IP: Internet Protocol
TCP: Transmission Control Protocol
UDP: User Datagram Protocol

1.3 Definition, context and components of a packet sniffer

1.3.1 Definition of a packet sniffer

A packet sniffer is a program that captures, monitors and analyzes data traveling over a network.

Its a cruel irony in information security that many of the features that make using computers easier or more efficient and the tools used to protect and secure the network can also be used to exploit and compromise the same computers and networks. This is the case with packet sniffing.

A packet sniffer, sometimes referred to as a network monitor or network analyzer, can be used legitimately by a network or system administrator to monitor and troubleshoot network traffic. Using the information captured by the packet sniffer an administrator can identify erroneous packets and use the data to pinpoint bottlenecks and help maintain efficient network data transmission.

In its simple form a packet sniffer simply captures all of the packets of data that pass through a given network interface.

Typically, the packet sniffer would only capture packets that were intended for the machine in question. However, if placed into promiscuous mode, the packet sniffer is also capable of capturing ALL packets traversing the network regardless of destination.

By placing a packet sniffer on a network in promiscuous mode, a malicious intruder can capture and analyze all of the network traffic. Within a given network, username and password information is generally transmitted in clear text which means that the information would be viewable by analyzing the packets being transmitted.

A packet sniffer can only capture packet information within a given subnet. So, its not possible for a malicious attacker to place a packet sniffer on their home ISP network and capture network traffic

from inside your corporate network (although there are ways that exist to more or less "hijack" services running on your internal network to effectively perform packet sniffing from a remote location). In order to do so, the packet sniffer needs to be running on a computer that is inside the corporate network as well. However, if one machine on the internal network becomes compromised through a Trojan or other security breach, the intruder could run a packet sniffer from that machine and use the captured username and password information to compromise other machines on the network.

Detecting rogue packet sniffers on your network is not an easy task. By its very nature the packet sniffer is passive. It simply captures the packets that are traveling to the network interface it is monitoring. That means there is generally no signature or erroneous traffic to look for that would identify a machine running a packet sniffer. There are ways to identify network interfaces on your network that are running in promiscuous mode though and this might be used as a means for locating rogue packet sniffers.

Sniffers can record network packets, decode them or simply display them. They are often used for network troubleshooting and for stealing information off a network.

1.3.2 Context of a packet sniffer

In an Ethernet network, a system broadcasts the data using an Ethernet frame. The destination system is specified in the Ethernet frame using its Ethernet address. All the systems in the network listen for an Ethernet frame with their Ethernet address in it. When a system receives an Ethernet frame with its address in it, it processes the frame and sends it to the higher layers (example: IP) for future processing. What is important to understand is that all machines in a network segment receive network data and are able to capture it, no matter if it is addressed to them or not. In this way, it is possible to imagine a program that would turn off the filter that ignores data that does not belong to it. This particular mode in which anybody situated on an Ethernet wire can observe traffic of systems on the same Ethernet wire is called **promiscuous mode** and programs using that mode are called sniffers.

1.3.3 Components of a packet sniffer

Packet sniffers are often made of the following components:

Driver: in charge of network traffic capture, the driver filters for specific data and stores it in a buffer.

Buffer: this component depends on the driver that fills it up or uses it as a round robin resource

(newest data replaces oldest data).

Decoder: decoding is an essential part of a packet sniffer as it displays the content of network traffic with information/analyses about network activity.

Editor: this feature enables edition/transmission of network packets onto the network.

1.3.4 Capablities of a packet sniffer

On wired broadcast LANs, depending on the network structure (hub or switch), one can capture traffic on all or just parts of the traffic from a single machine within the network; however, there are some methods to avoid traffic narrowing by switches to gain access to traffic from other systems on the network (e.g. ARP spoofing). For network monitoring purposes it may also be desirable to monitor all data packets in a LAN by using a network switch with a so-called monitoring port, whose purpose is to mirror all packets passing through all ports of the switch. When systems (computers) are connected to a switch port rather than a hub the analyzer will be unable to read the data due to the intrinsic nature of switched networks. In this case a shadow port must be created in order for the sniffer to capture the data.

On wireless LANs, one can capture traffic on a particular channel.

On wired broadcast and wireless LANs, in order to capture traffic other than unicast traffic sent to the machine running the sniffer software, multicast traffic sent to a multicast group to which that machine is listening, and broadcast traffic, the network adapter being used to capture the traffic must be put into promiscuous mode; some sniffers support this, others don't. On wireless LANs, even if the adapter is in promiscuous mode, packets not for the service set for which the adapter is configured will usually be ignored; in order to see those packets, the adapter must be put into monitor mode.

1.3.5 Uses of a packet sniffer

The versatility of packet sniffers means they can be used to:

- Analyze network problems.
- Detect network intrusion attempts.
- Gain information for effecting a network intrusion.
- Monitor network usage.
- Gather and report network statistics.
- Filter suspect content from network traffic.
- Spy on other network users and collect sensitive information such as passwords (depending on any content encryption methods which may be in use)
- Reverse engineer protocols used over the network.
- Debug client/server communications.
- Debug network protocol implementations.

Example uses:

- A packet sniffer for a token ring network could detect that the token has been lost or the presence of too many tokens (verifying the protocol).
- A packet sniffer could detect that messages are being sent to a network adapter; if the network adapter did not report receiving the messages then this would localize the failure to the adapter.
- A packet sniffer could detect excessive messages being sent by a port, detecting an error in the implementation.
- A packet sniffer could collect statistics on the amount of traffic (number of messages) from a process detecting the need for more bandwidth or a better method.
- A packet sniffer could be used to extract messages and reassemble into a complete form the traffic from a process, allowing it to be reverse engineered.
- A packet sniffer could be used to diagnose operating system connectivity issues like web, ftp, sql, active directory, etc.
- A packet sniffer could be used to analyse data sent to and from secure systems in order to understand and circumvent security measures, for the purposes of penetration testing or illegal activities.
- A packet sniffer can passively capture data going between a web visitor and the web servers, decode it at the HTTP and HTML level and create web log files as a substitute for server logs and page tagging for web analytics.

Introduction to sniffing methods

2.1 Sniffing methods

2.1.1 Broadcasts

The easiest way to sniff is probably to monitor broadcasts, multi-casts, and semi-directed packets. A malicious user can take advantage of broadcast network protocols such as NetBIOS, SNMP, bootp/DHCP, etc...

2.1.2 Switch Jamming

On some switches, unexpected failures result on removal of security provisions. This mechanism is known as fail open behavior. By over flowing switches address tables with generated MAC addresses or corrupted data, malicious users put switches into promiscuous mode where all frames are broadcast on all ports.

2.1.3 ARP redirect

A host in an Ethernet network can communicate with another host, only if it knows the Ethernet address (MAC address) of that host. ARP is used to get the Ethernet address of a host from its IP address. ARP is extensively used by all the hosts in an Ethernet network. Considering the ARP mechanism described above, one can imagine the fatal consequence that could result from a malicious user sending fake ARP frames. For instance, one could broadcast an ARP claiming to be some network router, in which case everyone will try to route through it.

Conversely, one could send an ARP to a router's MAC address claiming to be the victim.

2.1.4 ICMP redirect

According to RFC 792, the gateway sends a redirect message to a host in the following situation:

1. A gateway, G1, receives an Internet datagram from a host on a network to which the gateway is attached.

2. The gateway, G1, checks its routing table and obtains the address of the next gateway, G2, on the route to the datagram's Internet destination network, X.

3. If G2 and the host identified by the Internet source address of the datagram are on the same network, a redirect message is sent to the host.

4. The redirect message advises the host to send its traffic for network X directly to gateway G2 as this is a shorter path to the destination.

5. The gateway forwards the original datagram's data to its Internet destination.

A malicious user could subvert this mechanism by sending a redirect to host X claiming that it should send it host Y's packets.

2.1.5 ICMP router advertisements

A router sends out a periodic Router Advertisement using an ICMP message to notify the hosts on the network that the router is still available. A malicious user could claim to be a router by sending similar messages and then make systems forward their traffic through it.

2.1.6 MAC address faking

Using the auto-learning feature of a switch, one can send out frames with the source address of the victim to cause the switch to start forwarding it the frames destined to the victim.

Obviously, such a trick causes problems:

First, the victim will continue sending and receiving frames from the switch. Second, the malicious users can't just interrupt a victim's communication without interrupting the sniffing process.

Several solutions are available to an attacker:

A. Redirect the switch and continue with the connection as if nothing happened.

B. Periodically send out packets using victim's MAC address to occasionally capture data without interrupting victim's connection.

C. Receive victim's data and broadcast it back out.

2.1.7 Cable-taps

Cable-taps are physical products designed to tap into Ethernet cable.

2.2 Application areas

Some protocols are particularly vulnerable to sniffing. Following is a non-exhaustive list:

A. HTTP: Use of Basic authentication which send passwords across the network in plain-text. Data sent in clear-text.

B. SNMP: SNMP passwords - also called community-strings - are sent across the network in clear text.

C. NNTP: Password sent in clear. Data sent in clear.

D. POP: Password sent in clear. Data sent in clear.

E. FTP: Password sent in clear. Data sent in clear.

F. IMAP: Password sent in clear. Data sent in clear.

G. TELNET: Password sent in clear. Data sent in clear.

2.3 Sniffing tools (UNIX platforms)

UNIX sniffing tools are often base upon libpcap and/or Berkeley Packet Filter (BPF).

Following is a non-exhaustive list of some popular sniffers, as well as some tools to inject packets strongly formatted on the wire.

Ethereal/Wireshark:
Ethereal® is used by network professionals around the world for troubleshooting, analysis, software and protocol development, and education. It has all of the standard features you would expect in a protocol analyzer, and several features not seen in any other product. Its open source license allows talented experts in the networking community to add enhancements. It runs on all popular computing platforms, including Unix, Linux, and Windows.

IP Sniffer:
IP Sniffer is a suite of IP Tools built around a packet sniffer. The sniffer has basic features like filter, decode, replay, parse…

PlasticSniffer:
PlasticSniffer is a totally free, compact (<100k download!) and portable (thanks to .NET) packet sniffer. It supports file logging as well as filtering of ports and IP addresses.

PacketMon:
AnalogX PacketMon allows you to capture IP packets that pass through your network interface - whether they originated from the machine on which PacketMon is installed, or a completely different machine on your network! Once the packet is received, you can use the built in viewer to examine the header as well as the contents, and you can even export the results into a standard comma-delimited file to importing into your favorite program. As if that's not enough, PacketMon

has a powerful rule system that allows you to narrow down the packets it captures to ensure you get EXACTLY what you're after, without tons of unrelated information.

SmartSniff:
SmartSniff allows you to capture TCP/IP packets that pass through your network adapter, and view the captured data as sequence of conversations between clients and servers. You can view the TCP/IP conversations in Ascii mode (for text-based protocols, like HTTP, SMTP, POP3 and FTP.) or as hex dump. (for non-text base protocols, like DNS)

Sniffer.NET:
Two times PlanetSourceCode.com Contest winner. Sniffer.NET is an open source Network Packet Capture and Network Monitoring Tool in VB.NET. It utilize the power of Packmon.NET Library written in VB.NET which can monitor all network traffic. Now Supports User plugins which can be written in any .Net Language till they expose the IPlugin interface. Sample Plugin is also provided with the Source.

Sniphere:
Sniphere is an another network wiretapping program for Windows using winpcap. Nevertheless, Sniphere is a pretty handy program with a lot of possibilities which most of free sniffers do not have.

VisualSniffer:
VisualSniffer is a powerful packet capture tool and protocol analyzer (packet sniffer or ip sniffer) for Windows system. VisualSniffer can be used by LAN administrators, and security professionals for network monitoring intrusion detection, and network traffic logging. It can also be used by network programmers for checking what the developing program has sent and received, or others to get a full picture of the network traffic.

What Is Transferring:
What Is Transferring is a easy-to-use packet sniffer for Windows 2000/XP. It is able to capture TCP/IP packets that pass through your network adapter, and view the captured data in Text mode (for HTML page, e-mail) or in Hex/ASCII mode (for ZIP, JPEG, GIF). With this software, you can check if there is any unwanted connections.

These links provide some more available packet sniffers on the internet:

TCPDUMP: ftp://ftp.ee.lbl.gov/
SNIFFIT: http://reptile.rug.ac.be/~coder/sniffit/sniffit.html
SNORT: http://www.clark.net/~roesch/security.html
KARPSKI: http://www.trinux.org/
SUPERSNIFFER V1.3: http://www.mobis.com/~ajax/projects/
ESNIFF: http://www.phrack.com/search.phtml?view\&article=p45-5
EXDUMP: http://exscan.netpedia.net/exdump.html
LIBNET: http://www.packetfactory.net/libnet/
ROOTSHELL: http://www.rootshell.com/
NET::RAWIP: http://www.ic.al.lg.ua/~ksv/index.shtml

2.4 Protocol analysis

Protocol analysis consists in capturing network traffic and analyzing it to figure out network activity.

Below is a screen capture which shows sniffing with program ethereal.

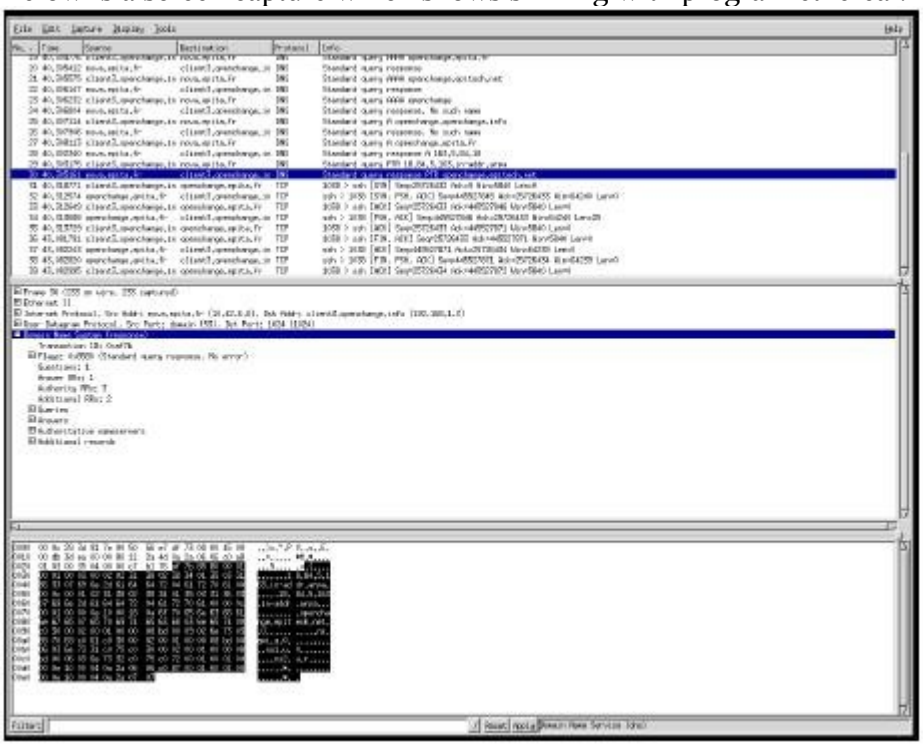


Figure 1: Ethereal screenshot example

The example above demonstrates how ethereal separates frames according to protocol layers. Indeed, frame 30 is composed of Ethernet data, IP data, UDP data and Domain Name System data which illustrate layers data encapsulation.

In general, protocol analysis is long and fastidious. It also requires a good background on network protocols. However most sniffers contain filters which enable to easily distinguish different layers in a same frame.

Introduction to packet sniffing using libcap

3.1 Introduction to pcap library

The Packet Capture library provides a high level interface to capture all packets on the network. In other words, pcap library enables us to get access to the underlying facility provided by the operating system to grab packets in their raw form using a filtering mechanism based on Berkeley packet filter
(BPF).

3.1.1 Where to get pcap library?

http://www.tcpdump.org

3.2 General layout of a pcap sniffer

Caution: For cross-compilation purpose it is advised to redefine IP and TCP header files since they are often different from a platform to another. We also recommend using libpcap version 0.8.3 or higher.

3.2.1 Which header files to include?

```
#include <pcap.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <netinet/if_ether.h>
. . .
```

3.2.2 Getting started

The packet sniffing basically includes following steps:

- We begin by determining which device or interface we want to sniff on. In Linux this may be something like eth0, in BSD it may be xl1, etc. We can either define this device in a string, or we can ask pcap to provide us with the name of an interface that will do the job.
- Initialize pcap. This is where we actually tell pcap what device we are sniffing on. We can, if we want to, sniff on multiple devices. We differentiate between using file handles. Just like opening a file for reading or writing, we must name our sniffing "session" so we can tell it apart from other such sessions.
- Filtering: In the event that we only want to sniff specific traffic (e.g.: only TCP/IP packets, only packets going to port 80, etc) we must create a rule set, "compile" it, and apply it. This is a three phase process, all of which is closely related. The rule set is kept in a string, and is converted into a format that pcap can read (hence compiling it.) The compilation is actually just done by calling a function within our program; it does not involve the use of an external application. Then we tell pcap to apply it to whichever session we wish for it to filter.
- Finally, we tell pcap to enter its primary execution loop. In this state, pcap waits until it has received however many packets we want it to. Every time it gets a new packet in, it calls another function that we have already defined. The function that it calls can do anything we want; it can dissect the packet and print it to the user, it can save it in a file, or it can do nothing at all.
- After our sniffing needs are satisfied, we close our session and are complete.

Now we discuss each of these stages in detail.

3.2.3 Setting the device

There are two techniques for setting the device that we wish to sniff on.

The first is that we can simply have the user tell us. Consider the following program:

```
#include <stdio.h>
#include <pcap.h>
int main (int argc, char *argv [])
{
    char *dev = argv[1];
    printf("Device: %s\n", dev);
    return(0);
}
```

The user specifies the device by passing the name of it as the first argument to the program.

$. /simpsniff eth0

Now the string "dev" holds the name of the interface that we will sniff on in a format that pcap can understand (assuming, of course, the user gave us a real interface).

The other technique is equally simply. Look at this program:

```
#include <stdio.h>
#include <pcap.h>
int main()
{
    char *dev, errbuf[PCAP_ERRBUF_SIZE];
    dev = pcap_lookupdev(errbuf);
    printf("Device: %s\n", dev);
    return(0);
}
```

In this case, pcap just sets the device on its own. Most of the pcap functions allow us to pass a string as an argument. The purpose of this string is in the event that some error occurs, the string is populated with a description of the error.

In this case, if pcap_lookupdev() fails, it will store an error message in errbuf.

If there are more than one devices available we can better use pcap_findalldevs() to find all the devices available.

Prototype:

```
int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf)
```

Description:

pcap_findalldevs() constructs a list of network devices that can be opened with **pcap_open_live()**. (Note that there may be network devices that cannot be opened with **pcap_open_live()** by the process calling **pcap_findalldevs()**, because, for example, that process might not have sufficient privileges to open them for capturing; if so, those devices will not appear on the list.).

Arguments:

1. **alldevsp***:* is set to point to the first element of the list; each element of the list is of type **pcap_if_t**, and has the following members:

   next
   if not **NULL**, a pointer to the next element in the list; **NULL** for the last element of the list
   name
   a pointer to a string giving a name for the device to pass to **pcap_open_live()**
   description
   if not **NULL**, a pointer to a string giving a human-readable description of the device
   addresses
   a pointer to the first element of a list of addresses for the interface
   flags
   interface flags:

PCAP_IF_LOOPBACK
set if the interface is a loopback interface

Each element of the list of addresses is of type **pcap_addr_t**, and has the following members:

next
if not **NULL**, a pointer to the next element in the list; **NULL** for the last element of the list
addr
a pointer to a **struct sockaddr** containing an address
netmask
if not **NULL**, a pointer to a **struct sockaddr** that contains the netmask corresponding to the address pointed to by **addr**
broadaddr
if not **NULL**, a pointer to a **struct sockaddr** that contains the broadcast address corresponding to the address pointed to by **addr**; may be null if the interface doesn't support broadcasts
dstaddr
if not **NULL**, a pointer to a **struct sockaddr** that contains the destination address corresponding to the address pointed to by **addr**; may be null if the interface isn't a point-to-point interface

Note that not all the addresses in the list of addresses are necessarily IPv4 or IPv6 addresses - you must check the **sa_family** member of the **struct sockaddr** before interpreting the contents of the address.

2. **Return value: -1** is returned on failure; **0** is returned on success.

3. **errbuf:** on failure **errbuf** is filled in with an appropriate error message.

3.2.4 Opening the device for sniffing

For creating a sniffing session, we use pcap_open_live(). The prototype of this function (from the pcap man page) is as follows:

Prototype:

```
pcap_t *pcap_open_live(char *device, int snaplen,
              int promisc, int to_ms, char *errbuf);
```

Description:

pcap_open_live() is used to obtain a packet capture descriptor to look at packets on the network.

Arguments:

- **device:** is a string that specifies the network device to open; on Linux systems with 2.2 or later kernels, a device argument of "any" or NULL can be used to capture packets from all interfaces.
- **snaplen:** specifies the maximum number of bytes to capture. If this value is less than the size of a packet that is captured, only the first snaplen bytes of that packet will be captured and provided as packet data. A value of 65535 should be sufficient, on most if not all networks, to capture all the data available from the packet.
- **promisc:** specifies if the interface is to be put into promiscuous mode. (Note that even if this parameter is false, the interface could well be in promiscuous mode for some other reason.) For now, this doesn't work on the "any" device; if an argument of "any" or NULL is supplied, the promisc flag is ignored.
- **to_ms:** specifies the read timeout in milliseconds. The read timeout is used to arrange that the read not necessarily return immediately when a packet is seen, but that it wait for some amount of time to allow more packets to arrive and to read multiple packets from the OS kernel in one operation. Not all platforms support a read timeout; on platforms that don't, the read timeout is ignored. A zero value for to_ms, on platforms that support a read timeout, will cause a read to wait forever to allow enough packets to arrive, with no timeout.
- **errbuf:** is used to return error or warning text. It will be set to error text when pcap_open_live() fails and returns NULL. errbuf may also be set to warning text when pcap_open_live() succeds; to detect this case the caller should store a zero-length string in

errbuf before calling pcap_open_live() and display the warning to the user if errbuf is no longer a zero-length string.

- **Return value:** The function returns the session handler.

To demonstrate, consider this code snippet:

```
#include <pcap.h>
...
pcap_t *handle;
handle = pcap_open_live(dev, BUFSIZ, 1, 0, errbuf);
```

This code fragment opens the device stored in the strong "dev", tells it to read however many bytes are specified in BUFSIZ (which is defined in pcap.h). We are telling it to put the device into promiscuous mode, to sniff until an error occurs, and if there is an error, store it in the string errbuf.

Promiscuous vs. non-promiscuous sniffing:

The two techniques are very different in style. In standard, non-promiscuous sniffing, a host is sniffing only traffic that is directly related to it. Only traffic to, from, or routed through the host will be picked up by the sniffer. Promiscuous mode, on the other hand, sniffs all traffic on the wire. In a non-switched environment, this could be all network traffic. The obvious advantage to this is that it provides more packets for sniffing, which may or may not be helpful depending on the reason you are sniffing the network. However, there are regressions. Promiscuous mode sniffing is detectable; a host can test with strong reliability to determine if another host is doing promiscuous sniffing. Second, it only works in a non-switched environment (such as a hub, or a switch that is being ARP flooded). Third, on high traffic networks, the host can become quite taxed for system resources.

3.2.5 Filtering traffic

It is useful for times when we may only be interested in specific traffic. For instance, there may be times when all we want is to sniff only on port 21 (telnet). Or perhaps we want to know the details of a file being sent over port 21 (FTP). Or maybe we just want DNS traffic (port 53 UDP). The fact is that at times when lots of computers are connected with the system on which the sniffer is running, there often arrives necessity to avoid unnecessary traffic. This can be easily achieved using following “filter” mechanisms provided by libcap:

Prototype:

```
int pcap_compile(pcap_t *handle,
       struct bpf_program *fp,const char *str,
              int optimize, bpf_u_int32 netmask);
```

Description:

pcap_compile() is used to compile the string str into a filter program.

Arguments:

1. **handle:** it is the handle to the sniffing session created using pcap_open_live().
2. **fp:** is a pointer to a bpf_program struct and is filled in by pcap_compile().
3. **str:** contains the string that is to be compiled.
4. **optimize:** controls whether optimization on the resulting code is performed.
5. **netmask:** specifies the IPv4 netmask of the network on which packets are being captured; it is used only when checking for IPv4 broadcast addresses in the filter program. If the netmask of the network on which packets are being captured isn't known to the program, or if packets are being captured on the Linux "any" pseudo-interface that can capture on more than one network, a value of 0 can be supplied; tests for IPv4 broadcast addreses won't be done correctly, but all other tests in the filter program will be OK.
6. **Return value:** A return of -1 indicates an error in which case pcap_geterr() may be used to display the error text.

pcap_compile_nopcap() is similar to pcap_compile() except that instead of passing a pcap structure, one passes the snaplen and linktype explicitly. It is intended to be used for compiling filters for

direct BPF usage, without necessarily having called pcap_open(). A return of -1 indicates an error; the error text is unavailable. (pcap_compile_nopcap() is a wrapper around pcap_open_dead(), pcap_compile(), and pcap_close(); the latter three routines can be used directly in order to get the error text for a compilation error.)

Prototype:

```
int pcap_setfilter(pcap_t *handle, struct bpf_program *fp)
```

Description:

pcap_setfilter() is used to specify a filter program.

Arguments:

1. **handle:** it is the session handle created above using pcap_open_live()
2. **fp:** is a pointer to a bpf_program struct, usually the result of a call to pcap_compile(). -1 is returned on failure, in which case pcap_geterr() may be used to display the error text; 0 is returned on success.

pcap's filter is far more efficient, because it does it directly with the BPF filter; we eliminate numerous steps by having the BPF driver do it directly.

Perhaps another code sample would help to better understand:

```
    #include <pcap.h>
    ...
    pcap_t *handle;                 /* Session handle */
    char dev[] = "eth0";          /* Device to sniff on */
    char errbuf[PCAP_ERRBUF_SIZE];    /* Error string */
/* The compiled filter expression */
    struct bpf_program filter;
/* The filter expression */
    char filter_app[] = "port 23";
/* The netmask of our sniffing device */
```

```
bpf_u_int32 mask;

bpf_u_int32 net;  /* The IP of our sniffing device */
pcap_lookupnet(dev, &net, &mask, errbuf);
handle = pcap_open_live(dev, BUFSIZ, 1, 0, errbuf);
pcap_compile(handle, &filter, filter_app, 0, net);
pcap_setfilter(handle, &filter);
```

This program prepares the sniffer to sniff all traffic coming from or going to port 23, in promiscuous mode, on the device eth0.

A word of caution: In actual experience it may be observed that this filter does not work across all operating systems. It has been observed that OpenBSD 2.9 with a default kernel does support this type of filter, but FreeBSD 4.3 with a default kernel does not.

3.2.6 Sniffing

There are two main techniques for capturing packets. We can either capture a single packet at a time, or we can enter a loop that waits for n number of packets to be sniffed before being done. We will begin by looking at how to capture a single packet, then look at methods of using loops. For this we use pcap_next().

Prototype:

```
u_char *pcap_next(pcap_t *handle, struct pcap_pkthdr *hdr);
```

Description:

pcap_next() reads the next packet (by calling pcap_dispatch() with a cnt of 1)

Arguments:

1. **handle:** is the session handle as created using pcap_open_live()
2. **hdr:** it is a pointer to a structure defined in pcap.h as:

```
struct pcap_pkthdr
 {
    struct timeval ts; /* time stamp */
    bpf_u_int32 caplen; /* length of portion present */
    bpf_u_int32 len; /* length this packet (off wire) */
};
```

The data members of this structure are:

ts: a *struct timeval* containing the time when the packet was captured
caplen: a *bpf_u_int32* giving the number of bytes of the packet that are available from the capture
len: a *bpf_u_int32* giving the length of the packet, in bytes (which might be more than the number of bytes available from the capture, if the length of the packet is larger than the maximum number of bytes to capture)

This structure holds general information about the packet, specifically the time in which it was sniffed, the length of this packet, and the length of his specific portion (incase it is fragmented, for example.)

3. **Return value:** a u_char pointer to the data in that packet. (The pcap_pkthdr struct for that packet is not supplied.) NULL is returned if an error occured, or if no packets were read from a live capture (if, for example, they were discarded because they didn't pass the packet filter, or if, on platforms that support a read timeout that starts before any packets arrive, the timeout expires before any packets arrive, or if the file descriptor for the capture device is in non-blocking mode and no packets were available to be read), or if no more packets are available in a ``savefile." Unfortunately, there is no way to determine whether an error occured or not.

Here is a simple demonstration of using pcap_next() to sniff a packet.

```
    #include <pcap.h>
    #include <stdio.h>
    int main()
    {
        pcap_t *handle;                /* Session handle */
        char *dev;           /* The device to sniff on */
        char errbuf[PCAP_ERRBUF_SIZE]; /* Error string */
        struct bpf_program filter;/* The compiled filter */
/* The filter expression */
        char filter_app[] = "port 23";
        bpf_u_int32 mask;              /* Our netmask */
        bpf_u_int32 net;                   /* Our IP */
/* The header that pcap gives us */
        struct pcap_pkthdr header;
        const u_char *packet;      /* The actual packet */

/* Define the device */
        dev = pcap_lookupdev(errbuf);
        /* Find the properties for the device */
        pcap_lookupnet(dev, &net, &mask, errbuf);
        /* Open the session in promiscuous mode */
```

```
        handle = pcap_open_live(dev, BUFSIZ, 1, 0, errbuf);


        /* Compile and apply the filter */
        pcap_compile(handle, &filter, filter_app, 0, net);
        pcap_setfilter(handle, &filter);

        /* Grab a packet */
        packet = pcap_next(handle, &header);
        /* Print its length */
        printf("The length of packet: [%d]\n", header.len);
        /* And close the session */
        pcap_close(handle);
        return(0);
    }
```

This application sniffs on whatever device is returned by pcap_lookupdev() by putting it into promiscuous mode. It finds the first packet to come across port 23 (telnet) and tells the user the size of the packet (in bytes).This program includes a new call, pcap_close(), which we will discuss later (although it really is quite self explanatory).

The other technique we can use is more complicated, and probably more useful. Few sniffers (if any) actually use pcap_next(). More often than not, they use pcap_loop() or pcap_dispatch() (which then themselves use pcap_loop()). To understand the use of these two functions, you must understand the idea of a callback function.

Callback functions are not anything new, and are very common in many API's. The concept behind a callback function is fairly simple. Suppose I have a program that is waiting for an event of some sort. For the purpose of this example, lets pretend that my program wants a user to press a key on the keyboard. Every time they press a key, I want to call a function which then will determine that to do. The function I am utilizing is a callback function. Every time the user presses a key, my program will call the callback function.

Callbacks are used in pcap, but instead of being called when a user presses a key, they are called when pcap sniffs a packet. The two functions that one can use to define their callback is pcap_loop() and pcap_dispatch(). pcap_loop() and pcap_dispatch() are very similar in their usage of callbacks. Both of them call a callback function every time a packet is sniffed that meets our filter requirements (if any filter exists, of course. If not, then all packets that are sniffed are sent to

the callback.)

Prototype:

```
int pcap_loop(pcap_t * handle, int cnt,
              pcap_handler callback, u_char *user)

int pcap_dispatch(pcap_t * handle, int cnt,
                  pcap_handler callback, u_char *user)
```

Description :

pcap_dispatch() is used to collect and process packets.

pcap_loop() is similar to **pcap_dispatch()** except it keeps reading packets until *cnt* packets are processed or an error occurs. It does **not** return when live read timeouts occur. Rather, specifying a non-zero read timeout to **pcap_open_live()** and then calling **pcap_dispatch()** allows the reception and processing of any packets that arrive when the timeout occurs. A negative *cnt* causes **pcap_loop()** to loop forever (or at least until an error occurs). -1 is returned on an error; 0 is returned if *cnt* is exhausted; -2 is returned if the loop terminated due to a call to **pcap_breakloop()** before any packets were processed. **If your application uses pcap_breakloop(), make sure that you explicitly check for -1 and -2, rather than just checking for a return value < 0.**

Arguments:

1. **handle:** it is the session handle.
2. **cnt:** specifies the maximum number of packets to process before returning. This is not a minimum number; when reading a live capture, only one bufferful of packets is read at a time, so fewer than *cnt* packets may be processed. A *cnt* of -1 processes all the packets received in one buffer when reading a live capture, or all the packets in the file when reading a ``savefile''.

3. **Return value:** The number of packets read is returned. 0 is returned if no packets were read

from a live capture (if, for example, they were discarded because they didn't pass the packet filter, or if, on platforms that support a read timeout that starts before any packets arrive, the timeout expires before any packets arrive, or if the file descriptor for the capture device is in non-blocking mode and no packets were available to be read) or if no more packets are available in a ``savefile.'' A return of -1 indicates an error in which case **pcap_perror()** or **pcap_geterr()** may be used to display the error text. A return of -2 indicates that the loop terminated due to a call to **pcap_breakloop()** before any packets were processed. **If your application uses pcap_breakloop(), make sure that you explicitly check for -1 and -2, rather than just checking for a return value < 0.**

NOTE: when reading a live capture, **pcap_dispatch()** will not necessarily return when the read times out; on some platforms, the read timeout isn't supported, and, on other platforms, the timer doesn't start until at least one packet arrives. This means that the read timeout should **NOT** be used in, for example, an interactive application, to allow the packet capture loop to ``poll'' for user input periodically, as there's no guarantee that **pcap_dispatch()** will return after the timeout expires.

4. **user:** This argument is useful in some applications, but many times is simply set as NULL. Suppose we have arguments of our own that we wish to send to our callback function, in addition to the arguments that pcap_loop() sends. This is where we do it. Obviously, you must typecast to a u_char pointer to ensure the results make it there correctly; as we will see later, pcap makes use of some very interesting means of passing information in the form of a u_char pointer.

5. **callback:** it is the callback routine with prototype as:

Prototype:

```
void callback(u_char *args,
      const struct pcap_pkthdr *header,
                  const u_char *packet)
```

Arguments:

1. **args:** this is a pointer to the u_char *user, passed from pcap_dispatch() or pcap_loop(). Whatever value is passed as the last argument to pcap_loop() is passed to the first argument of our callback function every time the function is called.
2. **header:** it a pointer to structure pcap_pkthdr. This argument contains information about when the packet was sniffed, how large it is, etc.
3. **Return type:** the function has a void return type. This is logical, because pcap_loop() wouldn't know how to handle a return value anyway.

4. **packet:** a *const u_char* pointer to the first **caplen** (as given in the *struct pcap_pkthdr* a pointer to which is passed to the callback routine) bytes of data from the packet (which won't necessarily be the entire packet; to capture the entire packet, you will have to provide a value for *snaplen* in your call to **pcap_open_live()** that is sufficiently large to get all of the packet's data - a value of 65535 should be sufficient on most if not all networks).

The use of this variable (named "packet" in our prototype) is as following:

A packet contains many attributes, so as you can imagine, it is not really a string, but actually a collection of structures (for instance, a TCP/IP packet would have an Ethernet header, an IP header, a TCP header, and lastly, the packet's payload). This u_char is the serialized version of these structures. To make any use of it, we must do some interesting typecasting.

First, we must have the actual structures define before we can typecast to them. The following is the structure definitions that I use to describe a TCP/IP packet over Ethernet. All three definitions that I use are taken directly out of the POSIX libraries. Normally I would have simply just used the definitions in those libraries, but it has been my experience that the libraries vary slightly from platform to platform, making it complicated to implement them quickly. So for demonstration purposes we will just avoid that mess and simply copy the relevant structures. All of these, incidentally, can be found in include/netinet on the local Unix system. Here are the structures:

```
/* Ethernet header */
struct sniff_ethernet
 {
/* Destination host address */
    u_char ether_dhost[ETHER_ADDR_LEN];
/* Source host address */
    u_char ether_shost[ETHER_ADDR_LEN];
    u_short ether_type; /* IP? ARP? RARP? etc */
};

/* IP header */
struct sniff_ip {
    #if BYTE_ORDER == LITTLE_ENDIAN
    u_int ip_hl:4, /* header length */
```

```
    ip_v:4; /* version */
    #if BYTE_ORDER == BIG_ENDIAN
    u_int ip_v:4, /* version */
    ip_hl:4; /* header length */
    #endif
    #endif /* not _IP_VHL */
    u_char ip_tos; /* type of service */
    u_short ip_len; /* total length */
    u_short ip_id; /* identification */
    u_short ip_off; /* fragment offset field */
    #define IP_RF 0x8000 /* reserved fragment flag */
    #define IP_DF 0x4000 /* dont fragment flag */
    #define IP_MF 0x2000 /* more fragments flag */
/* mask for fragmenting bits */
    #define IP_OFFMASK 0x1fff
    u_char ip_ttl; /* time to live */
    u_char ip_p; /* protocol */
    u_short ip_sum; /* checksum */
/* source and dest address */
    struct in_addr ip_src,ip_dst;
};

/* TCP header */
struct sniff_tcp {
    u_short th_sport; /* source port */
    u_short th_dport; /* destination port */
    tcp_seq th_seq; /* sequence number */
    tcp_seq th_ack; /* acknowledgement number */
    #if BYTE_ORDER == LITTLE_ENDIAN
    u_int th_x2:4, /* (unused) */
    th_off:4; /* data offset */
    #endif
    #if BYTE_ORDER == BIG_ENDIAN
    u_int th_off:4, /* data offset */
    th_x2:4; /* (unused) */
    #endif
    u_char th_flags;
    #define TH_FIN 0x01
    #define TH_SYN 0x02
    #define TH_RST 0x04
    #define TH_PUSH 0x08
    #define TH_ACK 0x10
```

```
    #define TH_URG 0x20
    #define TH_ECE 0x40
    #define TH_CWR 0x80
    #define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|
TH_CWR)
    u_short th_win; /* window */
    u_short th_sum; /* checksum */
    u_short th_urp; /* urgent pointer */
};
```

If we assume that we are dealing with a TCP/IP packet over Ethernet. (This same technique applies to any packet; the only difference is the structure types that you actually use.)

We begin by declaring the variables we will need to deconstruct the packet u_char.

```
/* The ethernet header */
const struct sniff_ethernet *ethernet;

const struct sniff_ip *ip; /* The IP header */

const struct sniff_tcp *tcp; /* The TCP header */

const char *payload; /* Packet payload */

/* For readability, we'll make variables for the sizes of each of
the structures */
int size_ethernet = sizeof(struct sniff_ethernet);
int size_ip = sizeof(struct sniff_ip);
int size_tcp = sizeof(struct sniff_tcp);
```

And then we do the typecasting:

```
ethernet = (struct sniff_ethernet*)(packet);

ip = (struct sniff_ip*)(packet + size_ethernet);

tcp = (struct sniff_tcp*)(packet +
```

```
                        size_ethernet + size_ip);

payload = (u_char *)(packet +
                     size_ethernet + size_ip + size_tcp);
```

The basic principle behind this typecasting is the layout of the packet u_char in memory. Basically, all that has happened when pcap stuffed these structures into a u_char is that all of the data contained within them was put in a string, and that string was sent to our callback. The convenient thing is that, regardless of the values set to these structures, their size always remains the same. For instance, a sniff_ethernet structure has a size of 14 bytes. a sniff_ip structure is 20 bytes, and likewise a sniff_tcp structure is 20 bytes. The u_char pointer is really just a variable containing an address in memory.

That's what a pointer is; it points to a location in memory. For the sake of simplicity, we'll say that the address this pointer is set to is the value X. Well, if our three structures are just sitting in line, the first of them (sniff_ethernet) being located in memory at the address X, then we can easily find the address of the other structures. So lets make a chart:

Variable	Location (in bytes)
sniff_ethernet	X
sniff_ip	X + 14
sniff_tcp	X + 14 + 20
Payload	X + 14 + 20 + 20

The sniff_ethernet structure, being the first in line, is simply at location X. sniff_ip, who follows directly after sniff_ethernet, is at the location X, plus however much space sniff_ethernet consumes (14 in this example). sniff_tcp is after both sniff_ip and sniff_ethernet, so it is location at X plus the sizes of sniff_ethernet and sniff_ip (14 and 20 byes, respectively). Lastly, the payload (which isn't really a structure, just a character string) is located after all of them.

Note: It is important not to assume that variables will always have these sizes. Hence, we should always use the sizeof() function to ensure that the sizes are accurate. This is because the members of each of these structures can have different sizes on different platforms.

Example code for pcap_loop( ):

```
main()
{
 char *dev;
 pcap_t *handle;
 int packet_count;
 dev = pcap_lookupdev(errbuf);
 if(dev==NULL)
  errors(errbuf);
 printf("Device: %s\n",dev);
handle = pcap_open_live(dev,BUFSIZ,1,0,errbuf);
if(handle == NULL)
  errors(errbuf);
printf("Sniffing session created for device %s.\n",dev);
printf("\nEnter the total no of packets to be captured"
     " (-ve value means until some error occurs)\t:");
scanf(" %d",&packet_count);
packets_read =
      pcap_loop(handle,packet_count,
                packet_found,(u_char *)file_ptr);
printf("\nClosing sniffing session ...\n");
printf("Total pcakets captured: %d\n",packets_read);
if(packets_read == -1)
 {
 pcap_perror(handle,errbuf);
 errors(errbuf);
 }
if(packets_read == -2)
errors("loop terminated due to
                call of pcap_breakloop() !");
pcap_close(handle);
}
```

3.2.7 Closing the session

Closing the session is as simple as closing a file. This task can be easily achieved using pcap_close() function defined in <pcap.h>

Prototype:

```
void pcap_close(pcap_t *handle);
```

the only argument is the session handle returned by pcap_open_live().

One important thing to keep in mind is to close all devices before closing the session, if the devices were opened using pcap_findalldevs(). This can be achieved using the method pcap_freealldevs().

Prototype:

```
int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf)

void pcap_freealldevs(pcap_if_t *alldevs)
```

SimpSniff: a practical example of a pcap application

4.1 Introduction to SimpSniff

SimpSniff is a simple network packet sniffer that capture the data being sent across the network in a very raw form. The captured packets can then be analyzed (a process known as packet analysis or protocol analysis) to reveal information about the packets their protocol, source, destination etc. This information can then be used to keep a watch on the incoming and outgoing packets which can again be used to troubleshoot network problems.

4.2 Supported Environment

Hardware Requirement:

System Configuration:

Processor:	Intel x86 processor family.
RAM:	32 MB or more.
Harddisk:	1GB or more.

Network Interface: 10/100 Mbps Ethernet, Wireless 802.11

Software Requirement:

Operating System: SuSE Linux, Fedora Linux, Debian Linux, Mandrake Linux
Red Hat Linux or any other flavour.

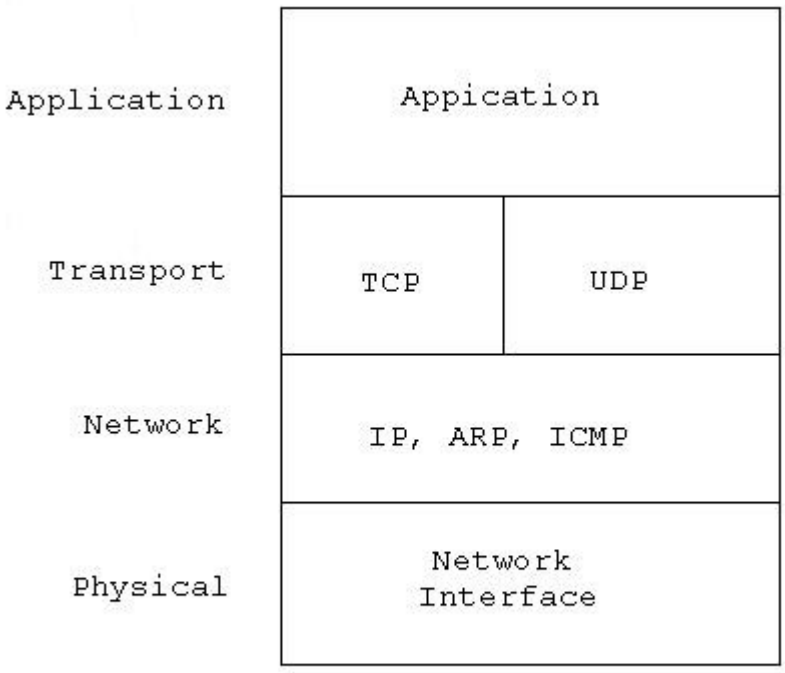
Compiler: GCC compiler

Tools : libcap 0.8.3 or higher version.

4.3 Design of SimpSniff

4.3.1 Introduction to TCP/IP

In the TCP/IP Architecture we only have 4 layers, contrast to the 7 layers of OSI Model



The TCP/IP Network Architecture

4.3.2 The Five Step Data Encapsulation

In order to have a better understanding of how the SimpSniff captures packets we first have to take a brief look at how packets actually travel in a TCP/IP model.

The only way of sending data from one machine to other using TCP/IP model, is by passing data from one layer to another by appending its corresponding header. This way of communication is known as " five step encapsulation", as described below:

The Five-step encapsulation includes the typical encapsulation by the transport, network, and data link layers as steps 2 through 4 in the process.

The first step is the application's creation of the data, and the last step is the physical layer's transmission of the bit stream.

Step 1 Create the data: This simply means that the application has data to send.

Step 2 Package the data for transport: In other words, the transport layer creates the transport header and places the data behind it.
The L4PDU is created here.
The OSI, TCP/IP, and NetWare Protocol Architectures 85

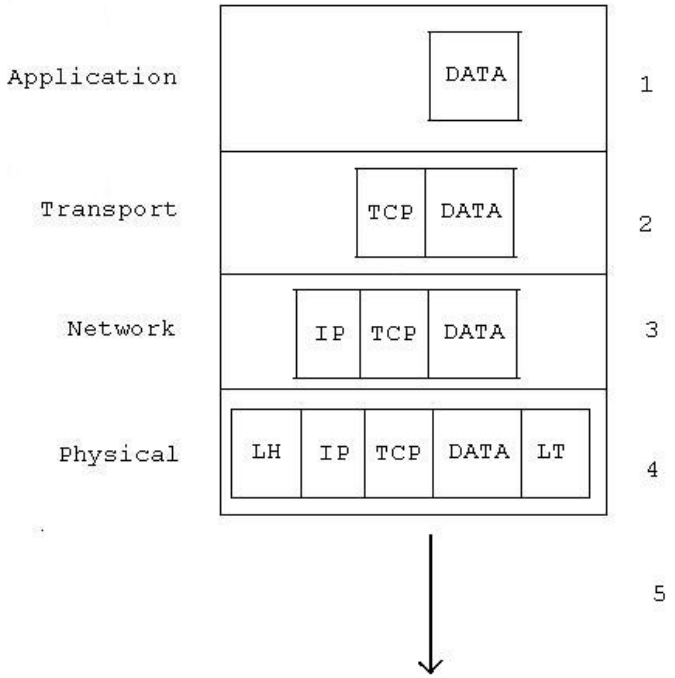
Step 3 Add the destination network layer address to the data: The network layer creates the network header, which includes the network layer address, and places the data (L4PDU) behind it. In other words, the L3PDU is created here.

Step 4 Add the destination data link address to the data: The data link layer creates the data link header, places the data (L3PDU) behind it, and places the data link trailer at the end. In other words, the L2PDU is created here.

Step 5 Transmit the bits: The physical layer encodes a signal onto the medium to transmit the frame.

This five-step process happens to match the TCP/IP network model very well.

The figure below depicts the concept; the numbers shown represent each of the five steps.



Five steps of Data encapsulation

4.3.3 The Flow of Program

There are infinite ways in which a packet sniffer can be customized to sniff packets traversing through the five layers of TCP/IP models, with each layer encapsulating the data by appending its own header.

Since one of the primary objective of SimpSniff is to provide a mechanism to demonstrate the working of a packet sniffer. Hence, we are deliberately limiting the functionality of the SimpSniff to only those protocols that are most widely used.

The various protocols considered at various layers are:

Network Interface Layer: Ethernet, ARP

Internet Layer: ICPMP, IP.

Transport Layer: TCP, UDP.

But, at any time the functionality can be increased to any protocol as we will soon observe.

4.4 Working of the project

The whole project is basically divided into three segments:

4.4.1 Segment 1: Pre-sniffing tasks

This part deals with the following objectives:

1. **Initializing various parameters required for a sniffing session.**

   Some of these parameters are:

   a. Interface or device: on which the sniffing session is to be created.
   b. Buffer size: to hold the packet that is to be analyzed.
   c. Promiscuous mode: yes if the user intends to capture all the packets trespassing through its machine, regardless of their destination, else no.
   d. Timeout: in milliseconds to provide the program the amount of time before quitting the sniffing session.
   e. Packet count: is the integer number of packets that is to be analyzed before quitting from the session.
   f. Log files: if 'yes' creates the unique log file for the session, else the output for the session is redirected to the 'stdout' stream, i.e. normally the terminal.

1. **Providing automatic and manual modes.**

   Each mode has been defined as to give different options to the user.

   Automatic mode: In this mode the program considers the best values for all the parameters for sniffing. Like:

```
bufsize = BUFSIZ;
sniffmode = 1;
timeout = 0;
```

However, some parameters like the “packet count” and “log-files” require input from the user. The main advantage of this mode is for users who lack deep technical knowledge about the various interfaces connected to the machine, or need for promiscuous mode, etc.
This mode just directly enters the sniffing session with least interaction.

Manual mode: After running the program if the user hits the ‘ENTER’ key before a specific interval of time, he enters the manual mode.

```
Press ENTER key to enter the manual mode ...
 5   4    3
```

The timer() modules calls the system call poll()

```
int poll(struct pollfd *fds,
          unsigned int nfds,int timeout_ms);
```

for ‘time’ interval of time, waiting for ‘POLLIN’ event to occur

```
pfd->events = POLLIN;
while(time && !poll(pfd,1,1000))
 {
  printf("%d ",time--);
  fflush(stdout);
 }
 ret = pfd->revents;
```

In contrast to the Auto-mode, manual is more like a deep question-answer session before starting the actual sniffing. The manual mode displays all possible devices found, over which the sniffing session can be created. Like,

```
S.No Device/Interface    Network Address Subnet Mask
1    eth0                60.243.244.0    255.255.240.0
2    eth1                10.0.0.1        255.255.255.0
3    lo                  127.0.0.1       0.0.0.0!
```

From these available devices the user can choose the device he wishes to sniff on.

```
 Enter the device or interface you want to sniff on …      :
```

Next the user is prompted with following querries:

```
Enter the size of the buffer to hold the pcakets\n"
"(recommended value is 8192)       :

Enter the mode of sniffing \n1: Promiscuous\n0: Non-Promiscuous :

Enter the read timeout in milliseconds
 0: Until an error occurs
-1: Indefinitely
(This value may get overwritten later on !!):
```

If everything goes well the sniffing session is created.

```
Opening sniffing session ....

Sniffing session created for device eth0
```

Next the user is prompted for total number of packets to be sniffed.

```
Enter the total no of packets to be captured
 (-ve value means until some error occurs)   :
```

2. Creation of log files.

Creating a log file is an optional part of sniffing session. The main advantage of creating log files is in events where the user is sniffing for troubleshooting purpose. Hence, the user can save the sessions file for afterwards analysis of the network traffic.

```
Do you want to create a log file of this session (y/n):
```

The various steps for creating a log file are:
Creating a filename: The first task is to create a file with unique filename. This is achieved by adding the time stamp to the filename. The timestamp consists of following entries:

1. Day, e.g. Mon, Tue, etc.
2. Date, e.g. Jun_02.
3. Time, e.g. 12_34_42. (HH:MM:SS)

This is achieved by help of time() function that is provided by the standard library as:

```
time_t fileid;      //include<time.h> id used for log file
time(&fileid);
sprintf(timestamp,"%s",ctime((time_t *)&fileid));
 for(p=timestamp;*p!='\0';p++)
  if(isspace(*p)|| *p==':')
   *p = '_';
 sprintf(fln,"%s%s%s","Sniff_",timestamp,".log");
```

eg of the filename:

```
Sniff_Sat_May_24_01_53_23_2008_.log Created ...\n
```

Adding attributes: The next step is to add attributes to the files. The attributes are added at the beginning. These attributes help in studying parameters that were defined for that particular session.

```
  log file opened ...
  Adding initial information to the log file ...

Device: eth0
Network Address: 255.255.240.0
Subnet Mask: 255.255.240.0
Buffer Size: 8192
Sniffing mode promiscuous ?: Y
Timeout :0
Packets Requested: 5
```

4. Closing the session

Closing the session is equally important as all other tasks. The main reason being that on some systems exiting without proper closing tasks, may lead to undesired results.

For proper exiting from the program three things need to be closed.

a. All opened files.
b. All opened devices.
c. All opened sessions.

```
Closing sniffing session ...

 Closing the log file …

 Freeing all devices ...
```

At the end the summary of the session is displayed as:

Total pcakets captured: 37

5. **Error handling**

Error handling is one of most important part of any program. For error handling we use following functions :

```
 pcap_perror(handle,errbuf);
```

4.4.2 Segment 2: The callback

This part is designed to handle each packet that is encountered by the machine.

```
void packet_found(u_char *filename,
     const struct pcap_pkthdr *header,
                    const u_char *packet)
```

The various actions taken by the callback function upon encountering a packet are:

1. **checking the output stream:**

```
Opening file ... Sniff_Sat_May_24_01_53_23_2008_.log
```

If the file can not be opened due to some error then the stream is redirected to 'stdout'.

```
  ERROR: The log file can not be opened !
  Changing stream to output ...
```

2. **displaying header information**:

```
Writing packet information to the file ...
```

```
packet no 0:

Packet Captured:
Total length of packet avilable = 54
Total length captured = 54
```

3. **calling subsequent functions to handle the protocol encountered**.

4.4.3 Segment 3: Analysis of the packet information

The various protocols header formats are handled by SimpSniff by using the fact that all the data contained inside a packet is actually carried in encapsulated form, i.e. every layers encapsulates the data by attaching its header at the beginning of the total payload and passes to the other layers.

In SimpSniff, the whole length of data is contained inside a large array that can be pointed by a pointer called as packet.

```
u_char *packet;
```

Hence, we can retrieve that information by simply typecasting the array to the desired header structure of the protocol encapsulating the data. As we go up-side to different layers of TCP/IP Architecture, we can typecast the packet to required structure header and pass the remaining data to "above" layers.

```
eth_hdr = (struct ether_header *)packet;
```

The structures for some of the popular headers are available at :

```
#include<netinet/ip.h>        //ip header
#include<netinet/tcp.h>       //tcp header
#include<netinet/udp.h>       //udp header
#include<net/if_arp.h>        //arp header
#include<netinet/if_ether.h>  //Ethernet header
```

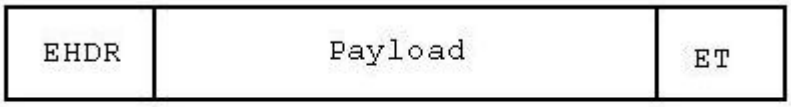
various protocol headers description is provided at the appendix.

Retrieving of data from various encapsulated headers:

1. The Ethernet frame:

```
10Mb/s ethernet header
struct ether_header
{
  u_int8_t  ether_dhost[ETH_ALEN];     destination eth addr
  u_int8_t  ether_shost[ETH_ALEN];       source ether addr
  u_int16_t ether_type;                packet type ID field
} __attribute__ ((__packed__));

 Ethernet protocol ID's
#define ETHERTYPE_PUP           0x0200           Xerox PUP
#define ETHERTYPE_IP            0x0800           IP
#define ETHERTYPE_ARP           0x0806   Address resolution
#define ETHERTYPE_REVARP        0x8035         Reverse ARP
```



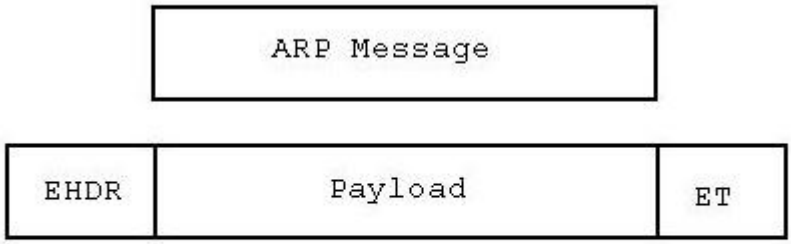
The Ethernet Frame

```
Printing the Ethernet information:
Source Address: 0:11:25:d8:ab:dd
Destination Address: 1:0:5e:0:0:16
Packet Type: IP
```

2. The encapsulation of ARP in Ethernet frame:

```
struct arphdr
  {
* Format of hardware address.  *
    unsigned short int ar_hrd;
* Format of protocol address.  *
    unsigned short int ar_pro;
* Length of hardware address.  *
    unsigned char ar_hln;
* Length of protocol address.  *
    unsigned char ar_pln;
* ARP opcode (command).  *
    unsigned short int ar_op;
#if 0
* Sender hardware address.  *
    unsigned char __ar_sha[ETH_ALEN];
* Sender IP address.  *
    unsigned char __ar_sip[4];
* Target hardware address.  *
    unsigned char __ar_tha[ETH_ALEN];
* Target IP address.  *
    unsigned char __ar_tip[4];
#endif
  };

hdr_covered = sizeof(struct ether_header);
arp_hdr = (struct arp *)(packet + hdr_covered);
```



ARP Encapsulated in an Ethernet frame

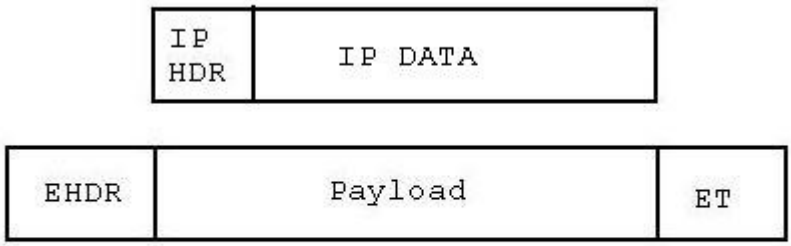
Printing the Ethernet information:
Source Address: 0:1b:24:63:c:16
Destination Address: ff:ff:ff:ff:ff:ff
Packet Type: ARP

ARP opcode: Undefined

3. **The encapsulation of IP packet in Ethernet frame:**

```
struct ip
  {
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int ip_hl:4;               * header length *
    unsigned int ip_v:4;                * version *
#endif
#if __BYTE_ORDER == __BIG_ENDIAN
    unsigned int ip_v:4;                * version *
    unsigned int ip_hl:4;               * header length *
#endif
    u_int8_t ip_tos;                    * type of service *
    u_short ip_len;                     * total length *
    u_short ip_id;                      * identification *
    u_short ip_off;                     * fragment offset field *
#define IP_RF 0x8000             * reserved fragment flag *
#define IP_DF 0x4000                 * dont fragment flag *
#define IP_MF 0x2000               * more fragments flag *
#define IP_OFFMASK 0x1fff     * mask for fragmenting bits *
    u_int8_t ip_ttl;                    * time to live *
    u_int8_t ip_p;                      * protocol *
    u_short ip_sum;                     * checksum *
    struct in_addr ip_src, ip_dst;* source & dest address *
  };

hdr_covered = sizeof(struct ether_header);
ip_hdr = (struct ip *)(packet + hdr_covered);
```



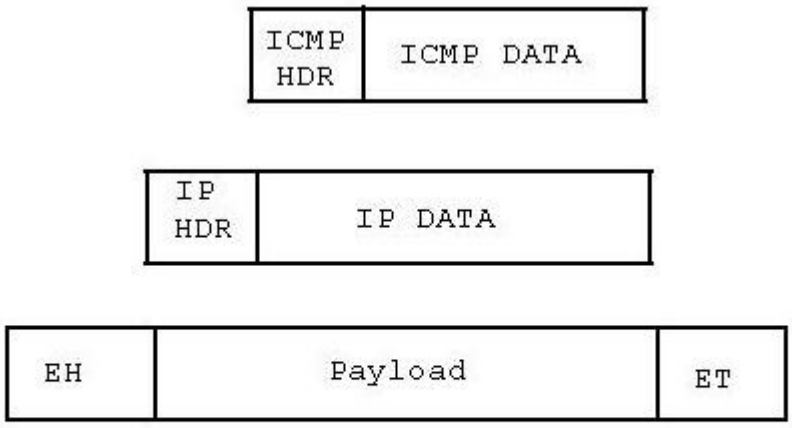
IP Encapsulated in an Ethernet frame

Printing the Ethernet information:
Source Address: 0:1b:24:63:c:16
Destination Address: 0:11:25:d8:ab:dd
Packet Type: IP

Printing the IP header information:
Source IP Address: 60.243.153.1
Destination IP Address: 224.0.0.22
Protocol: IGMP: Internet Group Management Protocol

4. **The encapsulation of ICMP in IP packet:**

```
hdr_covered = sizeof(struct ether_header)+
              sizeof(struct ip);
```



ICMP Encapsulation in an IP packet

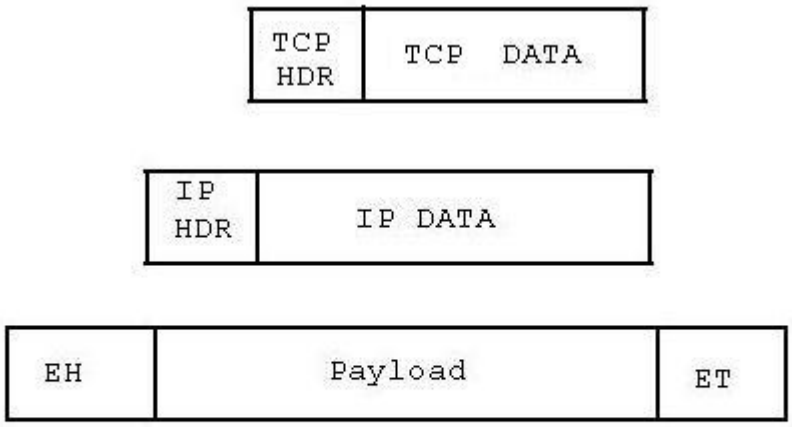
```
Printing the Ethernet information:
Source Address: 0:1b:24:63:c:16
Destination Address: 0:11:25:d8:ab:dd
Packet Type: IP

Printing the IP information:
Source IP Address: 60.243.153.135
Destination IP Address: 60.243.153.1
Protocol: ICMP: Internet Control Message Protocol
```

5. The encapsulation of TCP in IP packet:

```
struct tcphdr
  {
    u_int16_t th_sport;         * source port *
    u_int16_t th_dport;         * destination port *
    tcp_seq th_seq;             * sequence number *
    tcp_seq th_ack;             * acknowledgement number *
#  if __BYTE_ORDER == __LITTLE_ENDIAN
    u_int8_t th_x2:4;           * (unused) *
    u_int8_t th_off:4;          * data offset *
#  endif
#  if __BYTE_ORDER == __BIG_ENDIAN
    u_int8_t th_off:4;          * data offset *
    u_int8_t th_x2:4;           * (unused) *
#  endif
    u_int8_t th_flags;
#  define TH_FIN        0x01
#  define TH_SYN        0x02
#  define TH_RST        0x04
#  define TH_PUSH       0x08
#  define TH_ACK        0x10
#  define TH_URG        0x20
    u_int16_t th_win;           * window *
    u_int16_t th_sum;           * checksum *
    u_int16_t th_urp;           * urgent pointer *
};

hdr_covered = sizeof(struct ether_header)+
               sizeof(struct ip);
struct udphdr *tcp_hdr;
tcp_hdr = (struct tcphdr*)(packet + hdr_covered);
```



TCP Encapsulation in an IP packet

Printing the Ethernet information:
Source Address: 0:1b:24:63:c:16
Destination Address: 0:11:25:d8:ab:dd
Packet Type: IP

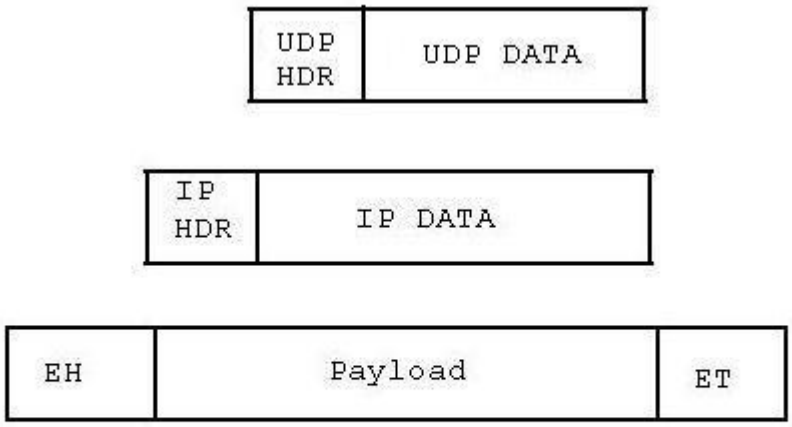
Printing the IP information:
Source IP Address: 60.243.153.135
Destination IP Address: 60.243.153.1
Protocol:TCP: Transmission Control Protocol

Printing the TCP information:
Source Port: 6660
Destination Port: 5632

6. The encapsulation of UDP in IP packet:

```
struct udphdr
{
  u_int16_t uh_sport;           * source port *
  u_int16_t uh_dport;           * destination port *
  u_int16_t uh_ulen;            * udp length *
  u_int16_t uh_sum;             * udp checksum *
};

hdr_covered = sizeof(struct ether_header)+
                sizeof(struct ip);
struct udphdr *udp_hdr;
udp_hdr = (struct udphdr*)(packet + hdr_covered);
```



UDP Encapsulation in an IP packet

```
Printing the Ethernet information:
Source Address: 0:11:25:d8:ab:dd
Destination Address: 1:0:5e:0:0:fb
Packet Type: IP
```

```
Printing the IP header information:
Source IP Address: 60.243.153.1
Destination IP Address: 224.0.0.251

Printing the UDP header information:
Source Port: 59668: user-defined service
Destination Port: 59668: user-defined service
```

4.5 Source Code

```
/*********************************************************
     Project: SimpSniff
     Filename: sniff.h
     Author: Sidharth Juyal
*********************************************************/

#include<pcap.h>
#include<stdio.h>
#include<netinet/if_ether.h>
/*for struct in_addr to convert ip to dot-notation & various
macros def.*/
#include<netinet/in.h>
/* for poll() syscall in timer()  */
#include<sys/poll.h>
#include<string.h>
#include<netinet/ip.h>        //ip header
#include<netinet/tcp.h>       //tcp header
#include<netinet/udp.h>       //udp header
#include<net/if_arp.h>        //arp header
#include<time.h>         //timestamp to log file

int timer(int time);           //ret: 1 timer interrupted; 0 timer
completes
void errors(char *);           //error handling
void packet_found(u_char *,const struct pcap_pkthdr *,const
                  u_char *);          //The Callback

/*
*    all these functions have following arguments:
*    const u_char *packet:    The packet sniffed
*    FILE *here:    The data stream where the output is to
                    be send
*    int hdr_covered:    The size of headers covered so far
                    starting from ethernet header
*/

/* if its ip header */
void its_ip(int ,const u_char *,FILE *,int);
/* tcp header */
```

```
void its_tcp(int ,const u_char *,FILE *,int);
/* udp header */
void its_udp(int ,const u_char *,FILE *,int);
/* arp header */
void its_arp(int ,const u_char *,FILE *,int);
void get_data(int ,const u_char *,FILE *,int);
/*ret: ptr to service of len SID_SNIFF_SER_MAX_LEN @
'callback.h'*/
char *porttoservice(int port_no,char *service);
/* creating log filename */
void createfln(char *);


/*********************************************************
     Project: SimpSniff
     Filename: callback.c
     Author: Sidharth Juyal
*********************************************************/

/*****************************************
     This is the callback function
*****************************************/

#include"sniff.h"
#define SID_SNIFF_SER_MAX_LEN 50

void packet_found(u_char *filename,const struct pcap_pkthdr
                    *header,const u_char *packet)
{
 int i;
 static count = 0;
 FILE *here;             //the data stream
 struct ether_header *eth_hdr;     // net/ethernet.h
 if(!filename)           //no filename provided
  here = stdout;
 else
 {
  if(count == 0)              //opening file first time
   printf("Opening file ... %s\n",filename);
  here = fopen(filename,"a"); //append in that file
  if(here == NULL)
  {
```

```
   fprintf(stderr,
          "ERROR: The log file can not be created !!\n");
   printf("Changing stream to output ...\n");
   here = stdout;
  }
  printf("Writting packet information to the file ...\n");
 }
 fprintf(here,"\n\npacket no %d:\n",count++);

 /*********************************************************
     packet's information:

   struct pcap_pkthdr {
     struct timeval ts;
     bpf_u_int32 caplen;
     bpf_u_int32 len;
     };
 *********************************************************/

 fprintf(here,"\nPacket Captured:\n");
 fprintf(here,
          "Total length of packet avilable = %d\n",
               header->len);
 fprintf(here,
          "Total length captured = %d\n",
               header->caplen);

/*********************************************************
     printing information inside the packet
      assuming ETHERNET interface:

This is a name for the 48 bit ethernet address available on many
systems.

struct ether_addr
{
  u_int8_t ether_addr_octet[ETH_ALEN];
} __attribute__ ((__packed__));

10Mb/s ethernet header
struct ether_header
{
```

```
  u_int8_t  ether_dhost[ETH_ALEN];     destination eth addr
  u_int8_t  ether_shost[ETH_ALEN];       source ether addr
  u_int16_t ether_type;                packet type ID field
} __attribute__ ((__packed__));

 Ethernet protocol ID's
#define ETHERTYPE_PUP           0x0200          Xerox PUP
#define ETHERTYPE_IP            0x0800          IP
#define ETHERTYPE_ARP           0x0806   Address resolution
#define ETHERTYPE_REVARP        0x8035         Reverse ARP
************************************************************/

/************************************************************
          Prototype of ether_ntoa:

     convert MAC Address to human readable:
     extern char *
          ether_ntoa(__const ether_addr *__addr) __THROW;
 ************************************************************/

 eth_hdr = (struct ether_header *)packet;
 fprintf(here,"\nPrinting the Ethernet information:\n");
 fprintf(here,
          "Source Address: %s\n",
                    ether_ntoa(eth_hdr->ether_shost));
 fprintf(here,
          "Destination Address:%s\n",
                    ether_ntoa(eth_hdr->ether_dhost));
 fprintf(here,"Packet Type: ");

/* i don't know why, but the orignals at <netinet/if_ether.h> are
not working for me */
#undef ETHERTYPE_IP
#define ETHERTYPE_IP 0x0008
#undef ETHERTYPE_ARP
#define ETHERTYPE_ARP 0x0608
#undef ETHERTYPE_REVARP
/* just reversing the bit-order, hope it works :( */
#define ETHERTYPE_REVARP 0x3580
switch(eth_hdr->ether_type)
 {
/* i'm not sure 'bout Xerox PUP
```

```
  case ETHERTYPE_PUP:
     printf("Xerox PUP\n");
     break;
*/
  case ETHERTYPE_IP:
     fprintf(here,"IP\n");
     its_ip(header->len,
          packet,here,sizeof(struct ether_header));
     break;
  case ETHERTYPE_ARP:
     fprintf(here,"ARP\n");
     its_arp(header->len,
          packet,here,sizeof(struct ether_header));
     break;
  case ETHERTYPE_REVARP:
     fprintf(here,"Reverse ARP\n");
     break;
  default:
     fprintf(here,"Unknown Type\n");
     /* its_ip(header->len,packet,
               here,sizeof(struct ether_header)); */
 }
 if(filename)       //don't close the stdout !!
  fclose(here);
}

void its_ip(int packet_len,
     const u_char *packet,FILE *here,int hdr_covered)
{
 fprintf(here,
     "\nPrinting the IP header information:\n");

struct ip *ip_hdr;

/*********************************************************
  Structure of the IP Header

struct ip
  {
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int ip_hl:4;               * header length *
    unsigned int ip_v:4;                * version *
```

```
#endif
#if __BYTE_ORDER == __BIG_ENDIAN
    unsigned int ip_v:4;                * version *
    unsigned int ip_hl:4;               * header length *
#endif
    u_int8_t ip_tos;                    * type of service *
    u_short ip_len;                     * total length *
    u_short ip_id;                      * identification *
    u_short ip_off;                     * fragment offset field *
#define IP_RF 0x8000             * reserved fragment flag *
#define IP_DF 0x4000                 * dont fragment flag *
#define IP_MF 0x2000                * more fragments flag *
#define IP_OFFMASK 0x1fff     * mask for fragmenting bits *
    u_int8_t ip_ttl;                    * time to live *
    u_int8_t ip_p;                      * protocol *
    u_short ip_sum;                     * checksum *
    struct in_addr ip_src, ip_dst;* source & dest address *
  };
*******************************************************/

 ip_hdr = (struct ip *)(packet + hdr_covered);
 hdr_covered += sizeof(struct ip);

 fprintf(here,
     "Source IP Address: %s\n",
                    inet_ntoa(ip_hdr->ip_src));
 fprintf(here,
     "Destination IP Address: %s\n",
                    inet_ntoa(ip_hdr->ip_dst));

 fprintf(here,"Protocol:");

 switch(ip_hdr->ip_p)
 {
  case IPPROTO_IP:
     fprintf(here,"Dummy TCP\n");
     break;
  case IPPROTO_ICMP:
     fprintf(here,
          "ICMP: Internet Control Message Protocol\n");
     break;
  case IPPROTO_IGMP:
```

```
      fprintf(here,
           "IGMP: Internet Group Management Protocol\n");
      break;
   case IPPROTO_IPIP:
      fprintf(here,"IPIP: IPIP Tunnels\n");
      break;
   case IPPROTO_TCP:
      fprintf(here,"TCP: Transmission Control Protocol\n");
      its_tcp(packet_len,packet,here,hdr_covered);
      break;
   case IPPROTO_EGP:
      fprintf(here,"EGP: Exterior Gateway Protocol\n");
      break;
   case IPPROTO_PUP:
      fprintf(here,"PUP\n");
      break;
   case IPPROTO_UDP:
      fprintf(here,"UDP: User Datagram Protocol\n");
      its_udp(packet_len,packet,here,hdr_covered);
      break;
   case IPPROTO_IDP:
      fprintf(here,"IDP: XNS IDP Protocol\n");
      break;
   case IPPROTO_TP:
      fprintf(here,"TP: SO Transport Protocol\n");
      break;
   case IPPROTO_IPV6:
      fprintf(here,"IPv6");
      break;
   case IPPROTO_RSVP:
      fprintf(here,"RSVP: Reservation Protocol\n");
      break;
   case IPPROTO_GRE:
      fprintf(here,"GRE: General Routing Encapsulation\n");
      break;
   case IPPROTO_ESP:
      fprintf(here,"ESP: Encapsulation Security Payload\n");
      break;
   case IPPROTO_AH:
      fprintf(here,"AH: Authentication Header\n");
      break;
   case IPPROTO_MTP:
```

```
     fprintf(here,"MTP: Multicast Transport Protocol\n");
     break;
  case IPPROTO_ENCAP:
     fprintf(here,"ENCAP: Encapsulation Header\n");
     break;
  case IPPROTO_PIM:
     fprintf(here,"PIM: Protocol Independent Multicasting\n");
     break;
  case IPPROTO_COMP:
     fprintf(here,"COMP: Compression Header Protocol\n");
     break;
  case IPPROTO_SCTP:
     fprintf(here,
          "SCTP: Stream Control Transmission Protocol\n");
     break;
  case IPPROTO_RAW:
     fprintf(here,"RAW: RAW IP Packet\n");
     break;
  default:
     fprintf(here,"Unknown\n");
 }
}

void its_tcp(int packet_len,
     const u_char *packet,FILE *here,int hdr_covered)
{
 fprintf(here,"\nPrinting the TCP header information:\n");

/*********************************************************
     TCP HEADER

# ifdef __FAVOR_BSD
typedef u_int32_t tcp_seq;

 * TCP header.
 * Per RFC 793, September, 1981.
 *
struct tcphdr
  {
    u_int16_t th_sport;         * source port *
    u_int16_t th_dport;         * destination port *
    tcp_seq th_seq;             * sequence number *
```

```
    tcp_seq th_ack;             * acknowledgement number *
#  if __BYTE_ORDER == __LITTLE_ENDIAN
    u_int8_t th_x2:4;           * (unused) *
    u_int8_t th_off:4;          * data offset *
#  endif
#  if __BYTE_ORDER == __BIG_ENDIAN
    u_int8_t th_off:4;          * data offset *
    u_int8_t th_x2:4;           * (unused) *
#  endif
    u_int8_t th_flags;
#  define TH_FIN        0x01
#  define TH_SYN        0x02
#  define TH_RST        0x04
#  define TH_PUSH       0x08
#  define TH_ACK        0x10
#  define TH_URG        0x20
    u_int16_t th_win;           * window *
    u_int16_t th_sum;           * checksum *
    u_int16_t th_urp;           * urgent pointer *
};

#else     * !__FAVOR_BSD *
struct tcphdr
  {
    u_int16_t source;
    u_int16_t dest;
    u_int32_t seq;
    u_int32_t ack_seq;
#  if __BYTE_ORDER == __LITTLE_ENDIAN
    u_int16_t res1:4;
    u_int16_t doff:4;
    u_int16_t fin:1;
    u_int16_t syn:1;
    u_int16_t rst:1;
    u_int16_t psh:1;
    u_int16_t ack:1;
    u_int16_t urg:1;
    u_int16_t res2:2;
#  elif __BYTE_ORDER == __BIG_ENDIAN
    u_int16_t doff:4;
    u_int16_t res1:4;
    u_int16_t res2:2;
```

```
    u_int16_t urg:1;
    u_int16_t ack:1;
    u_int16_t psh:1;
    u_int16_t rst:1;
    u_int16_t syn:1;
    u_int16_t fin:1;
#  else
#   error "Adjust your <bits/endian.h> defines"
#  endif
    u_int16_t window;
    u_int16_t check;
    u_int16_t urg_ptr;
};
# endif * __FAVOR_BSD *

*********************************************************/
 struct tcphdr *tcp_hdr;
 char ser_name[SID_SNIFF_SER_MAX_LEN];
 tcp_hdr = (struct tcphdr *)(packet + hdr_covered);
 hdr_covered += sizeof(struct tcphdr);
 fprintf(here,
     "Source Port: %u: %s\n",
          tcp_hdr->source,
          porttoservice(tcp_hdr->source,ser_name));
 fprintf(here,
     "Destination Port: %u: %s\n",
          tcp_hdr->dest,
          porttoservice(tcp_hdr->dest,ser_name));
 get_data(packet_len,packet,here,hdr_covered);
}

void its_udp(int packet_len,const u_char *packet,
                FILE *here, int hdr_covered)
{
 fprintf(here,"\nPrinting the UDP header information:\n");

/*******************************************************
     UDP Header:

 * UDP header as specified by RFC 768, August 1980. *
#ifdef __FAVOR_BSD
```

```
struct udphdr
{
  u_int16_t uh_sport;           * source port *
  u_int16_t uh_dport;           * destination port *
  u_int16_t uh_ulen;            * udp length *
  u_int16_t uh_sum;             * udp checksum *
};

#else

struct udphdr
{
  u_int16_t source;
  u_int16_t dest;
  u_int16_t len;
  u_int16_t check;
};
#endif

#define SOL_UDP            17     * sockopt level for UDP *

#endif * netinet/udp.h *
******************************************************/

 struct udphdr *udp_hdr;
 char ser_name[SID_SNIFF_SER_MAX_LEN];
 udp_hdr = (struct udphdr*)(packet + hdr_covered);
 hdr_covered += sizeof(struct udphdr);
 fprintf(here,
          "Source Port: %u: %s\n",
          udp_hdr->source,
          porttoservice(udp_hdr->source,ser_name));
 fprintf(here,
          "Destination Port: %u: %s\n",
          udp_hdr->dest,
          porttoservice(udp_hdr->dest,ser_name));
 get_data(packet_len,packet,here,hdr_covered);
}

void its_arp(int packet_len,const u_char *packet,
               FILE *here, int hdr_covered)
{
```

```
 fprintf(here,"\nPrinting the ARP header information:\n");

/*********************************************************
     ARP Header:

* Some internals from deep down in the kernel.  *
#define MAX_ADDR_LEN    7


* This structure defines an ethernet arp header.  *

* ARP protocol opcodes. *
#define ARPOP_REQUEST   1               * ARP request.  *
#define ARPOP_REPLY     2               * ARP reply.  *
#define ARPOP_RREQUEST  3               * RARP request.  *
#define ARPOP_RREPLY    4               * RARP reply.  *
#define ARPOP_InREQUEST 8               * InARP request.  *
#define ARPOP_InREPLY   9               * InARP reply.  *
#define ARPOP_NAK       10              * (ATM)ARP NAK.  *

* See RFC 826 for protocol description.  ARP packets are variable
   in size; the arphdr structure defines the fixed-length portion.
   Protocol type values are the same as those for 10 Mbs Ethernet.
   It is followed by the variable-sized fields ar_sha, arp_spa,
   arp_tha and arp_tpa in that order, according to the lengths
   specified.  Field names used correspond to RFC 826.  *

struct arphdr
  {
* Format of hardware address.  *
    unsigned short int ar_hrd;
* Format of protocol address.  *
    unsigned short int ar_pro;
* Length of hardware address.  *
    unsigned char ar_hln;
* Length of protocol address.  *
    unsigned char ar_pln;
* ARP opcode (command).  *
    unsigned short int ar_op;
#if 0
    * Ethernet looks like this : This bit is variable sized
       however...  *
```

```
* Sender hardware address.  *
    unsigned char __ar_sha[ETH_ALEN];
* Sender IP address.  *
    unsigned char __ar_sip[4];
* Target hardware address.  *
    unsigned char __ar_tha[ETH_ALEN];
* Target IP address.  *
    unsigned char __ar_tip[4];
#endif
  };
*****************************************************/

 struct arphdr *arp_hdr;
 arp_hdr = (struct arphdr *)(packet + hdr_covered);
 hdr_covered += sizeof(struct arphdr);
 fprintf(here,"ARP opcode: ");
 switch(arp_hdr->ar_op)
 {
  case ARPOP_REQUEST:
     fprintf(here,"ARP Request\n");
     break;
  case ARPOP_REPLY:
     fprintf(here,"ARP Reply\n");
     break;
  case ARPOP_RREQUEST:
     fprintf(here,"RARP Request\n");
     break;
  case ARPOP_RREPLY:
     fprintf(here,"RARP Reply\n");
     break;
  case ARPOP_InREQUEST:
     fprintf(here,"InARP Request\n");
     break;
  case ARPOP_InREPLY:
     fprintf(here,"InARP Reply\n");
     break;
  case ARPOP_NAK:
     fprintf(here,"ATM ARP NAK\n");
     break;
  default:
     fprintf(here,"Undefined\n");
 }
```

```
 get_data(packet_len,packet,here,hdr_covered);
}

/*********************************************************
     Print the Payload
*********************************************************/

void get_data(int packet_len,const u_char *packet,
               FILE *here,int hdr_covered)
{
#undef SID_SNIFF_ETH_TRAIL
#define SID_SNIFF_ETH_TRAIL 4
 int data_len =
      packet_len - hdr_covered - SID_SNIFF_ETH_TRAIL;
 fprintf(here,
     "\nPrinting the data remaining in the packet:\n");
 for(data_len = packet_len - hdr_covered;
               data_len > 0;
                    data_len--)
  fprintf(here,"%c ",*packet++);
}


char *porttoservice(int port,char *ser)
{
/*********************************************************
     * Standard well-known ports.  *
enum
  {
    IPPORT_ECHO = 7,            * Echo service.  *
    IPPORT_DISCARD = 9,         * Discard transmissions
                                  service.  *
    IPPORT_SYSTAT = 11,         * System status service.  *
    IPPORT_DAYTIME = 13,        * Time of day service.  *
    IPPORT_NETSTAT = 15,        * Network status service.
                                  *
    IPPORT_FTP = 21,            * File Transfer Protocol.
                                  *
    IPPORT_TELNET = 23,         * Telnet protocol.  *
    IPPORT_SMTP = 25,           * Simple Mail Transfer
                                  Protocol.  *
    IPPORT_TIMESERVER = 37,     * Timeserver service.  *
```

```
    IPPORT_NAMESERVER = 42,     * Domain Name Service.  *
    IPPORT_WHOIS = 43,          * Internet Whois service.
                                   *
    IPPORT_MTP = 57,

    IPPORT_TFTP = 69,           * Trivial File Transfer
                                   Protocol.  *
    IPPORT_RJE = 77,
    IPPORT_FINGER = 79,         * Finger service.  *
    IPPORT_TTYLINK = 87,
    IPPORT_SUPDUP = 95,         * SUPDUP protocol.  *


    IPPORT_EXECSERVER = 512,    * execd service.  *
    IPPORT_LOGINSERVER = 513,   * rlogind service.  *
    IPPORT_CMDSERVER = 514,
    IPPORT_EFSSERVER = 520,

    * UDP ports.  *
    IPPORT_BIFFUDP = 512,
    IPPORT_WHOSERVER = 513,
    IPPORT_ROUTESERVER = 520,

    * Ports less than this value are reserved for privileged
processes.  *
    IPPORT_RESERVED = 1024,

    * Ports greater this value are reserved for (non-privileged)
servers.  *
    IPPORT_USERRESERVED = 5000
  };
 ***********************************************************/
if(port < IPPORT_RESERVED)
 switch(port)
 {
  case 7: strcpy(ser,"Echo service.");
     break;
  case 9: strcpy(ser,"Discard transmissions service.");    break;
  case 11:    strcpy(ser,"System status service.");        break;
  case 13:    strcpy(ser,"Time of day service.");          break;
  case 15:    strcpy(ser,"Network status service." );      break;
  case 21:    strcpy(ser,"File Transfer Protocol.");       break;
```

```
  case 23:     strcpy(ser,"Telnet protocol.");              break;
  case 25:     strcpy(ser,
                    "Simple Mail Transfer Protocol.");
     break;
  case 37:     strcpy(ser,"Timeserver service.");      break;
  case 42:     strcpy(ser,"Domain Name Service.");          break;
  case 43:     strcpy(ser,"Internet Whois service.");       break;
  case 69:     strcpy(ser,
                    "Trivial File Transfer Protocol.");
     break;
  case 79:     strcpy(ser,"Finger service.");               break;
  case 80:     strcpy(ser,"HTTP service.");
     break;
  case 95:     strcpy(ser,"SUPDUP protocol.");              break;
  case 512:    strcpy(ser,"execd service.");           break;
  case 513:    strcpy(ser,"rlogind service.");              break;
  default:     strcpy(ser,"unknown system service");
 }
else
 strcpy(ser,"user-defined service");
return ser;
}



/*********************************************************
     Project: SimpSniff
     Filename: timer.c
     Author: Sidharth Juyal
*********************************************************/

/************************************************
 run the timer for specified no of seconds or some interrupts
occurs like pressing ENTER key
************************************************/

#include"sniff.h"
int timer(int time)
{
 int ret;
 /*****************************************
  struct pollfd {
```

```
     int fd;        file descriptor
     short events;  requested events
     short revents; returned events
     };
 ******************************************/
 struct pollfd *pfd =
          (struct pollfd *)malloc(sizeof(struct pollfd));
 pfd->fd = 0;
 pfd->events = POLLIN;
 /******************************************
 int poll(struct pollfd *fds,
          unsigned int nfds,int timeout_ms);
 ******************************************/
 while(time && !poll(pfd,1,1000))
 {
  printf("%d ",time--);
  fflush(stdout);
 }
 ret = pfd->revents;
 free(pfd);
return ret;
}

/*********************************************************
     Project: SimpSniff
     Filename: main.c
     Author: Sidharth Juyal
*********************************************************/

/*********************************************************
     SimpSniff : The Packet Sniffer
*********************************************************/

#include"sniff.h"

#define TIME 5
#define MAN_SIZE 15      //0.0.0.0 to 255.255.255.255


int main()
{
 int i;
```

```
 int mode;               //manual (1) or automatic (0)
 char *dev;              //device
 char netipdot[MAN_SIZE];//network address in dot-notation
 char maskdot[MAN_SIZE]; //mask in dot-notation
 char errbuf[PCAP_ERRBUF_SIZE];    //for error handling
 bpf_u_int32 netip;      //network addr
 bpf_u_int32 mask;       //subnet mask
 int ret;           //return code of pcap_lookupnet()
/* the network address @ "netinet/in.h" */
 struct in_addr netaddr,maskaddr;
 pcap_t *handle;         //session handle
 int timeout;       //read timeout for sniffing session
 int packet_count;       //total pcakets to be captured
 char log_choice;        //wanna create log file ??
 char file[25]={0};      //space for filename
 char *file_ptr;         //for sending filename to callback
 int packets_read;       //total no. of packets read

/******for manual override**************************/
 char manual[MAN_SIZE];  //for manual overriding purpose
 int bufsize;            //buffer size
 int sniffmode;               //promiscuous ?? ;)
 pcap_if_t *devlist;     //to hold the list of all devices
 int devno;              //device number
/*file pointer for adding initial information
      to the log file */
 FILE *f;

 /*********************************************************
     Setting up the interface
 *********************************************************/

 printf("Press ENTER key to enter the manual mode ...\n");
 mode = timer(TIME);

 if(mode)
 {
  printf("Interfaces found :\n");
  pcap_findalldevs(&devlist,errbuf);
  printf("S.No\tDevice/Interface
          \tNetwork Address\t\tSubnet Mask\n");
  while(devlist!=NULL)
```

```
  {
   printf("%d",++devno);
   printf("\t%s",devlist->name);
   pcap_lookupnet(devlist->name, &netip, &mask ,errbuf);
   netaddr.s_addr = netip;
   printf("\t\t\t%s",(char *)inet_ntoa(netaddr));
   maskaddr.s_addr = mask;
   printf(" \t\t%s",(char *)inet_ntoa(maskaddr));
   devlist = devlist->next;
   printf("\n");
  }
  printf("\nEnter the device or interface
                you want to sniff on ..\t:");
  scanf(" %s",manual);
  dev = manual;
 }

 else
 {
/*********************************************************
          Prototype of pcap_lookupdev:

      char *pcap_lookupdev(char *errbuf);
*********************************************************/
  dev = pcap_lookupdev(errbuf);
  if(dev==NULL)
   errors(errbuf);
  printf("Device: %s\n",dev);
 }
/*********************************************************
          Prototype of pcap_lookupnet:

 int pcap_lookupnet(const char *device, bpf_u_int32 *netp,
                         bpf_u_int32 *maskp, char *errbuf);
*********************************************************/
  ret = pcap_lookupnet(dev, &netip, &mask, errbuf);
  if(ret == -1)
   errors(errbuf);

  netaddr.s_addr = netip;
/*********************************************************
          Prototype of inet_ntoa:
```

```
      char *inet_ntoa(struct in_addr in);
*********************************************************/
  strcpy(netipdot,(char *)inet_ntoa(netaddr));
  if(inet_ntoa(netaddr)==0)
   errors("sniffer: error in obtaining IP Address\n");
  printf("Network Address: %s\n",netipdot);

  maskaddr.s_addr = mask;
  strcpy(maskdot,(char *)inet_ntoa(maskaddr));
  if(inet_ntoa(maskaddr)==0)
   errors("sniffer: error in obtaining Subnet Mask\n");
  printf("Subnet Mask: %s\n",maskdot);


 /*********************************************************
     Starting Sniffing Session
 *********************************************************/
 if(mode)
 {
  printf("\nEnter the size of the buffer
                     to hold the pcakets\n"
          "(recommended value is %d) \t:",BUFSIZ);
  scanf(" %d",&bufsize);
  printf("\nEnter the mode of sniffing
          \n1: Promiscuous\n0: Non-Promiscuous\t:");
  scanf(" %d",&sniffmode);
  printf("\nEnter the read timeout in milliseconds\n"

     " 0: Until an error occurs\n"
     "-1: Indefinitely\n"
     "(This value may get overwritten later on !!)\t:");
  scanf(" %d",&timeout);
 }
 else
 {
  bufsize = BUFSIZ;
  sniffmode = 1;
  timeout = 0;
 }
/*********************************************************
          Prototype of pcap_open_live:
```

```
     pcap_t *pcap_open_live(char *device, int snaplen,
                       int promisc, int to_ms, char *ebuf);
*********************************************************/
 printf("\nOpening sniffing session ....\n");
 handle = pcap_open_live(dev,bufsize,sniffmode,timeout,errbuf);
 if(handle == NULL)
  errors(errbuf);
 printf("Sniffing session created for device %s.\n",dev);

/*********************************************************
     Capturing Packets
*********************************************************/
 printf("\nEnter the total no of packets to be captured"
     " (-ve value means until some error occurs)\t:");
 scanf(" %d",&packet_count);

 printf("\nDo you want to create a log file
                       of this session (y/n)\t:");
 scanf(" %c",&log_choice);
 if(log_choice == 'y')
 {
  createfln(file);
  printf("%s Created ...\n",file);
  file_ptr = file;  //send filename
  /***********************************************
     Adding attributes to the log file
***********************************************/
  f = fopen(file,"w");
  if(f == NULL)
   errors("Unable to open log file");
  printf("log file opened ...\n");
  printf("Adding initial information
                to the log file ...\n");
  fprintf(f,"Device: %s\n",dev);
  fprintf(f,"Network Address: %s\n",netipdot);
  fprintf(f,"Subnet Mask: %s\n",maskdot);
  fprintf(f,"Buffer Size: %d\n",bufsize);
  fprintf(f,"Sniffing mode promiscuous ?: %c\n",
                          (sniffmode)?'Y':'N');
  fprintf(f,"Timeout :%d\n",timeout);
  fprintf(f,"Packets Requested: %d\n",packet_count);
```

```
  fclose(f);
 }
 else
  file_ptr = 0;          //send no filename
/*********************************************************
          Prototype of pcap_loop:

     int pcap_loop(pcap_t *handle, int count,
                pcap_handler callback, u_char *user);
     *********************************************************/
 packets_read =
      pcap_loop(handle,packet_count,
                packet_found,(u_char *)file_ptr);


/*********************************************************
     Closing Sniffing Session
*********************************************************/
 printf("\nClosing sniffing session ...\n");
 if(mode)
 {
  printf("Freeing all devices ...\n");
/* free al devices allocated by pcap_findalldevs() */
  pcap_freealldevs(devlist);
 }
// printf("Total pcakets captured: %d\n",packets_read);
 if(packets_read == -1)
 {
  pcap_perror(handle,errbuf);
  errors(errbuf);
 }
 if(packets_read == -2)
  errors("loop terminated due to
                call of pcap_breakloop() !");
 pcap_close(handle);
}

/*********************************************************
     Create Filename
*********************************************************/
void createfln(char *fln)
{
```

```
 char timestamp[25];          //timestamp on the file
 char *p;
 time_t fileid;               //include<time.h> id used for log
file
 time(&fileid);

 sprintf(timestamp,"%s",ctime((time_t *)&fileid));
 for(p=timestamp;*p!='\0';p++)
  if(isspace(*p)|| *p==':')
   *p = '_';
 sprintf(fln,"%s%s%s","Sniff_",timestamp,".log");
}
/*********************************************************
      For error handling
*********************************************************/

void errors(char *err)
{
 printf("\n%s\n%s\n%s\n",
          "ERROR: ",err,"Thanks for using SimpSniff");
 exit(1);
}
```

Conclusion

5. Conclusion

This introduction to sniffing methods is now over. After reviewing basic network concepts - topologies, standard models, protocols, we have introduced packet sniffing. First as a definition, then through sniffing methods we finally demonstrated with SimpSniff (a Project module) how to use pcap library to develop a practical sniffing tool. Far from being exhaustive in its description of sniffing methods, this paper gives the tools to understand how SimpSniff reaches its objectives in permitting the analysis of network protocols.

As discussed before the cruel irony in information security that many of the features that make using computers easier or more efficient and the tools used to protect and secure the network can also be used to exploit and compromise the same computers and networks.

If we need to maintain and monitor a network, we should become familiar with network monitors or packet sniffers. Learn what types of information can be discerned from the captured data and how we can put it to use to keep our network running smoothly. But, also be aware that some users on the network may be running rogue packet sniffers, either experimenting out of curiosity or with malicious intent, and that you should do what you can to make sure this does not happen.

By this we conclude our discussion on the network packet sniffing methods.

Appendix

A Network Protocols Headers

A.1 Ethernet Header

Definition:

Ethernet defines a 48-bit addressing scheme. Each computer attached to an Ethernet network is assigned a unique 48-bit number known as its Ethernet address. To assign an address, Ethernet hardware manufacturers purchase blocks of Ethernet addresses and assign them in sequence as they manufacture Ethernet interface hardware. Thus, no two hardware interfaces have the same Ethernet address.

Usually, the Ethernet address is fixed in machine readable form on the host interface hardware. Because each Ethernet address belongs to a hardware device, they are sometimes called hardware addresses, physical addresses, media access (MAC) addresses, or layer 2 addresses.

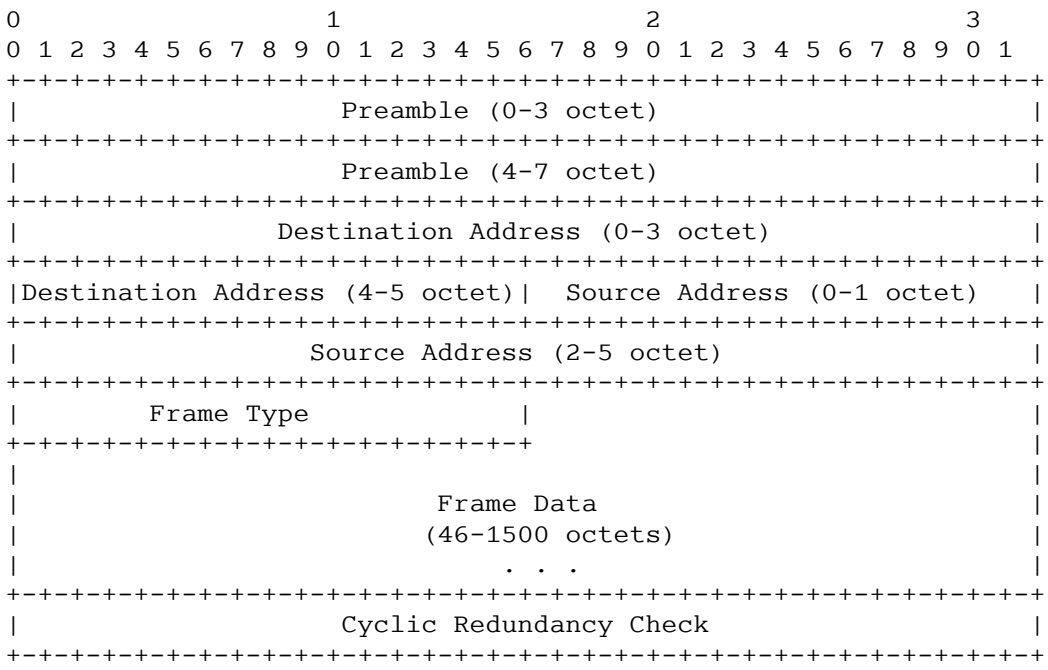
A 48-bit Ethernet address can do more than specify a single destination computer. An address can be one of three types:

1. The physical address of one network interface (a unicast address)
2. The network broadcast address
3. A multicast address

By convention, the broadcast address (all 1s) is reserved for sending to all stations simultaneously. Multicast addresses provide a limited fom~o f broadcast in which a subset of the computers on a network agree to listen to a given multicast address. The set of participating computers is called a multicast group. To join a multicast group, a computer must instruct its host interface to accept the group's multicast address. The advantage of multicasting lies in the ability to limit broadcasts: every computer in a multicast group can be reached with a single packet transmission, but computers that choose not to participate in a particular multicast group do not receive packets sent to the group.

Ethernet Frame Format:

Ethernet frames are of variable length, with no frame smaller than 64 octets* or larger than 1518 octets (header, data, and CRC). As in all packet-switched networks, each Ethernet frame contains a field that contains the address of its destination. Following is the Ethernet frame format contains the physical source address as well as the destination address.



Description:

In addition to identifying the source and destination, each frame transmitted across the Ethernet contains a preamble, type field, data field, and Cyclic Redundancy Check (CRC).
1. PREAMBLE: consists of 64 bits of alternating 0s and I s to help receiving interfaces synchronize.
2. DESTINATION & SOURCE ADDRESS: The 48-bit field is the actual hardware address of the interconnected machines.
3. CRC: This 32-bit field helps the interface detect transmission errors: the sender computes the CRC as a function of the data in the frame, and the receiver recomputes the CRC to verify that the packet has been received intact.
4. FRAME TYPE: this field contains a 16-bit integer that identifies the type of the data being carried in the frame. From the Internet point of view, the frame type field is essential because it means Ethernet frames are self-identifying. When a frame arrives at a given machine, the operating system uses the frame type to determine which protocol software module should process the frame. The chief advantages of self-identifying frames are that they allow multiple protocols to be used together on a single computer and they allow multiple protocols to be intermixed on the same physical network without interference.

A.2 Address Resolution Protocol (ARP)

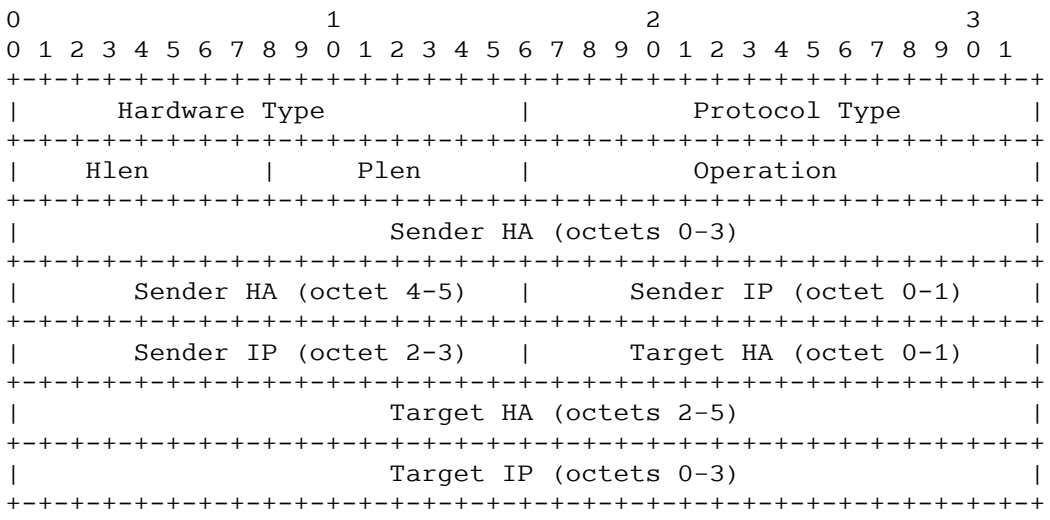
Definition:

ARP is a low-level protocol that hides the underlying network physical addressing, permitting one to assign an arbitrary IP address to every machine. We think of ARP as part of the physical network system, and not as part of the internet protocols.

ARP provides one possible mechanism to map from IP addresses to physical addresses. The point is that ARP would be completely unnecessary if we could make all network hardware recognize IP addresses. Thus, ARP merely imposes a new address scheme on top of whatever low-level address mechanism the hardware uses.

Unlike most protocols, the data in ARP packets does not have a fixed-format header. Instead, to make ARP useful for a variety of network technologies, the length of fields that contain addresses depend on the type of network. However, to make it possible to interpret an arbitrary ARP message, the header includes fixed fields near the beginning that speclfy the lengths of the addresses found in succeeding fields. In fact, the ARP message format is general enough to allow it to be used with arbitrary physical addresses and arbitrary protocol addresses. The following example shows the 28-octet ARP message format used on Ethernet hardware (where physical addresses are 48-bits or 6 octets long), when resolving IP protocol addresses (which are 4 octets long).

ARP Header Format:



Description:

1. HARDWARE TYPE: specifies a hardware interface type for which the sender

seeks an answer; it contains the value 1 for Ethernet.

2. PROTOCOL TYPE specifies the type of high-level protocol address the sender has supplied; it contains 0800, for IP addresses.

3. OPERATION: specifies an ARP request (I), ARP

response (2), RARPt request (3), or RARP response (4).

4. HLEN and PLEN allow ARP to be used with arbitrary networks because they speclfy the length of the hardware address and the length of the high-level protocol address.

5. Sender HA & IP : Senders Hardware and IP address if known.

6. Target HA & Target IP: When making a request, the sender also supplies the target hardware address (RARP) or target IP address (ARP), using fields TARGET HA or TARGET IP. Before the target machine responds, it fills in the missing addresses, swaps the target and sender pairs, and changes the operation to a reply. Thus, a reply carries the IP and hardware addresses of the original requester, as well as the IP and hardware addresses of the machine for which a binding was sought.

A.3 Internet Protocol

Definition:

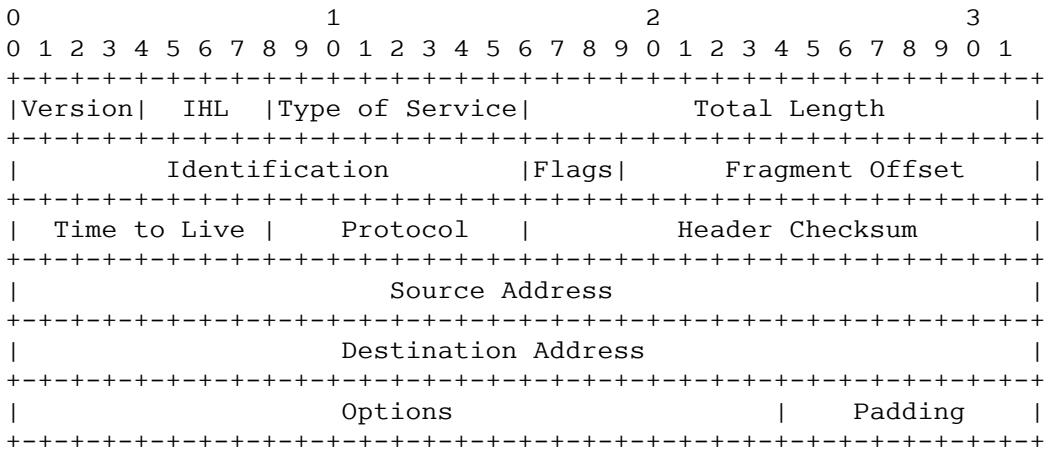
The Internet Protocol (IP) is software that performs a number of functions. It is responsible for how datagrams are created and moved across a network. IP performs one set of tasks when transmitting data to a computer and a different set of tasks when receiving data from another computer. Three key 32-bit fields (areas of information) within the IP software are integral to its operation:

- IP Address field—A unique 32-bit address assigned to a computer, or more accurately, to a node.
- Subnet Mask field—A 32-bit pattern of bits used to tell IP how to determine which part of the IP address is the network portion and which part is the host portion.
- Default Gateway field—An optional 32-bit address that, if present, identifies the address of a router. Datagrams destined for another network are sent to this address to be routed appropriately.

The Internet Protocol (IP) provides the first level of abstraction that provides a virtual view of the network where all nodes are treated as IP nodes. IP provides an abstract view of the network, a logical view wherein the network is viewed as an idealized network possessing properties that are described in this chapter.

Because of the abstraction layer that IP provides, protocols above the IP layers—such as the TCP and UDP protocols—can treat the network as an IP-only network. In reality, the network may not be an IP-only network; it may support other protocols besides TCP/IP. The IP node's network connections are identified by the 32-bit value known as the IP address. IP provides connectionless services to upper-layer services. The connectionless service is implemented using datagrams that contain the source and destination IP addresses and other parameters needed for IP operation.

IP Header Format:



Description:

1. Version: Indicates which version of IP is being used. The current version of IP is 4. The binary pattern is 0100.
2. Internet Header Length (IHL): The length of the IP header in 32-bit words. The minimum header length is five 32-bit words. The typical binary pattern for this field is 0101.
3. Type of Service: The Source IP can designate special routing information. The main choices include Low or Normal Delay, Normal or High Throughput, and Normal or High Reliability. There are seven other rarely used options.
4. Total Length: Identifies the length, in octets, of the IP datagram. This length includes the IP header and the data payload.
5. Identification: An incrementing sequenced number assigned to datagrams by the Source IP.
6. Flags:Flags indicate fragmentation possibilities. There are a total of three flags, the first of which is unused.
A DF (Don't Fragment) flag will signify whether fragmentation is allowed.
The MF (More Fragments) flag will signify that the datagram is a fragment. When set to 0, the MF flag indicates that no more fragments exist or that it never was fragmented.
Fragmentation occurs when the source of a TCP/IP datagram is on a network that allows large data payloads, such as token ring. For instance, a TCP/IP datagram could start with a data payload of 4,000 bytes. If the datagram is routed through an Ethernet network where the maximum IP data payload is 1,480 bytes, fragmentation must occur. If fragmentation is allowed, the router will break the large data payload into several smaller data fragments and send them through the Ethernet network as several datagrams. At the destination the fragmented datagrams are reassembled.
7. Fragment offset: A numeric value assigned to each successive fragment. IP at the destination uses the fragment offset to reassemble the fragments into the proper order.
8. Time to Live: Indicates the amount of time in seconds or router hops that the datagram can survive before being discarded. Every router examines and decrements this field by at least 1, or by the number of seconds the datagram is delayed inside the router. The datagram is discarded when this field reaches zero.
9. Protocol: This field holds the protocol address to which IP should deliver the data payload.
• Header checksum—This field holds a 16-bit calculated value to verify the validity of the header only. This field is recomputed in every router as the TTL field decrements.
10. Source IP address: This address is used by the Destination IP when sending a response.
11. Destination IP address: This address is used by the Destination IP to verify correct delivery.
12. IP data payload: This field typically contains data destined for delivery to TCP or UDP in the Transport layer, ICMP, or IGMP. The amount of data is variable but could include thousands of bytes.

A.4 Internet Control Message Protocol (ICMP)

Definition:

The Internet Control Message Protocol allows routers to send error or control messages to other routers or hosts.
ICMP provides communication between the Internet Protocol software on one machine and the Internet Protocol software on another.

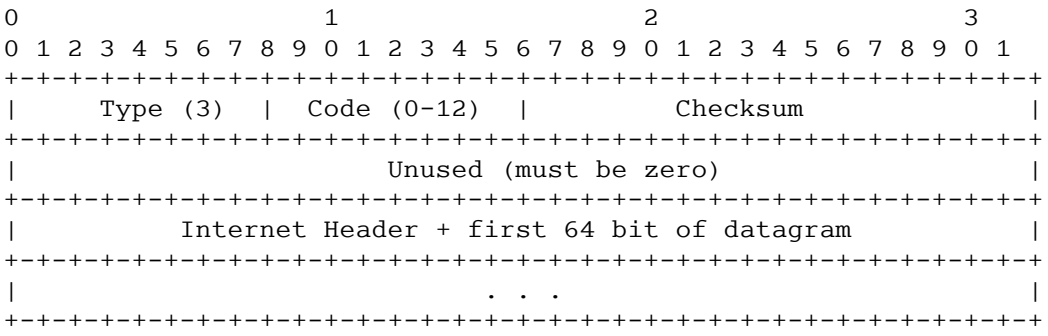
Technically, ICMP is an error reporting mechanism. It provides a way for routers that encounter an error to report the error to the original source. Although the protocol specification outlines intended uses of ICMP and suggests possible actions to take in response to error reports, ICMP does not fully specify the action to be taken for each possible error.

TCP/IP protocols provide facilities to help network managers or users identlfy network problems. One of the most frequently used debugging tools invokes the ICMP echo request and echo reply messages.

A host or router sends an ICMP echo request message to a specified destination. Any machine that receives an echo request formulates an echo reply and returns it to the original sender. The request contains an optional data area; the reply contains a copy of the data sent in the request. The echo request and associated reply can be used to test whether a destination is reachable and responding. Because both the request and reply travel in IP datagrams, successful receipt of a reply verifies that major pieces of the transport system work. First, IP software on the source computer must route the datagram. Second, intermediate routers between the source and destination must be operating and must route the datagram correctly. Third, the destination machine must be running (at least it must respond to interrupts), and both ICMP and IP software must be working. Finally, all routers along the return path must have correct routes.

On many systems, the command users invoke to send ICMP echo requests is named “ping”. Sophisticated versions of ping send a series of ICMP echo requests, capture responses, and provide statistics about datagram loss. They allow the user to specify the length of the data being sent and the interval between requests. Less sophisticated versions merely send one ICMP echo request and await a reply.

ICMP Header Format:



Description:

Although each ICMP message has its own format, they all begin with the same three fields:
1. TYPE: an 8-bit integer message field that identifies the message.
The ICMP TYPE field defines the meaning of the message as well as its format.
The types include:

Type Field	ICMP Message Type
0	Echo Reply
3	Destination Unreachable
4	Source Quench
5	Redirect (change a route)
8	Echo Request
9	Router Advertisement
10	Router Solicitation
11	Time Exceeded for a Datagram
12	Parameter Problem on a Datagram
13	Timestamp Request
14	Timestamp Reply
15	Information Request (obsolete)
16	Information Reply (obsolete)
17	Address Mask Request
18	Address Mask Reply

2. CODE: an 8-bit field that provides further information about the message type.

Code	Meaning
0	Network unreachable
1	Host unreachable
2	Protocol unreachable
3	Port unreachable
4	Fragmentation needed and DF set
5	Source route failed
6	Destination network unknown
7	Destination host unknown
8	Source host isolated
9	Communication with destination network administratively prohibited
10	Communication with destination host administratively prohibited
11	Network unreachable for type of service
12	Host unreachable for type of service

3. CHECKSUM: a 16-bit field (ICMP uses the same additive checksum algorithm as IP, but the ICMP checksum only covers the ICMP message). In addition, ICMP messages that report errors always include the header and first 64 data bits of the datagram causing the problem.
The reason for returning more than the datagram header alone is to allow the receiver to determine more precisely which protocol(s) and which application program were responsible for the datagram. As higher-level protocols in the
TCP/IP suite is designed so that crucial information is encoded in the fist 64 bits.

A.5 User Datagram Protocol (UDP)

Definition:

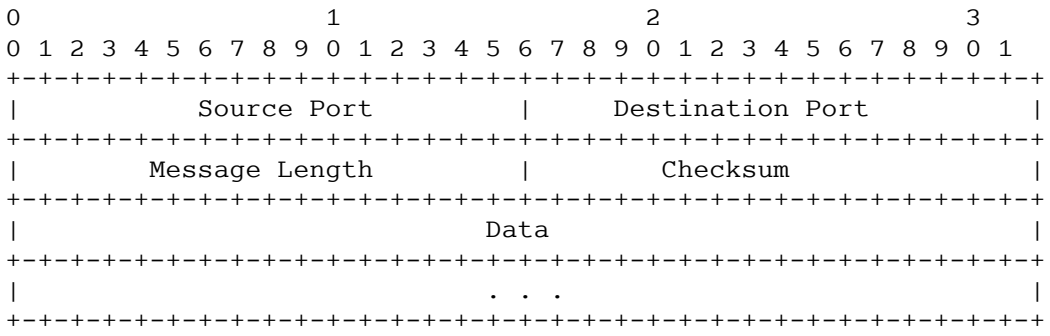
In the TCPDP protocol suite, the User Datagram Protocol or UDP provides the primary mechanism that application programs use to send datagrams to other application programs.

UDP provides protocol ports used to distinguish among multiple programs executing on a single machine. That is, in addition to the data sent, each UDP message contains both a destination port number and a source port number, making it possible for the UDP software at the destination to deliver the message to the correct recipient and for the recipient to send a reply.

UDP uses the underlying Internet Protocol to transport a message from one machine to another, and provides the same unreliable, connectionless datagram delivery semantics as IP. It does not use acknowledgements to make sure messages arrive, it does not order incoming messages, and it does not provide feedback to control the rate at which information flows between the machines.

Thus, UDP messages can be lost, duplicated, or arrive out of order. Furthermore, packets can arrive faster than the recipient can process them.

UDP Header Format:



Description:

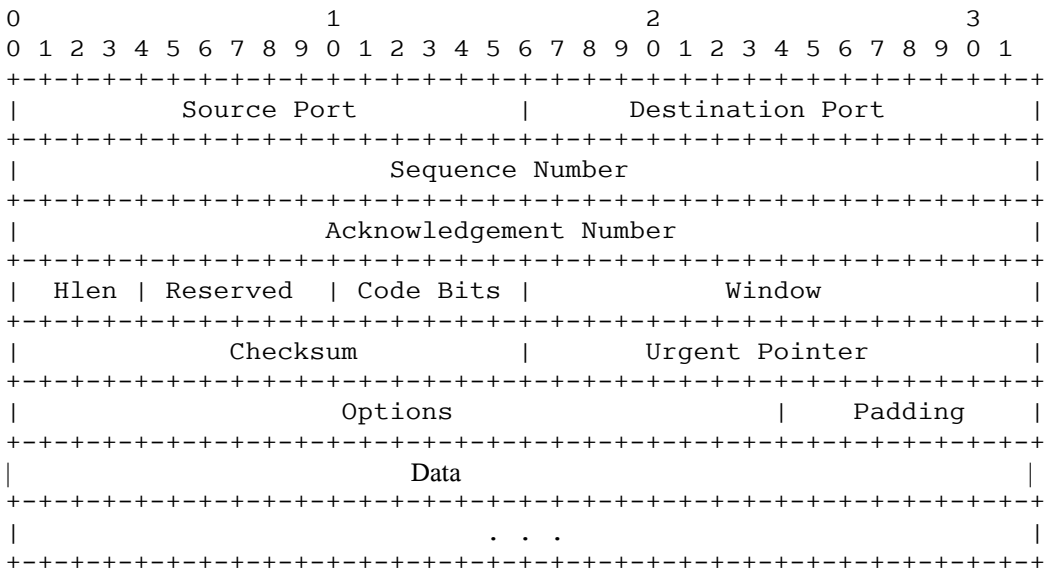
1. SOURCE PORT and DESTINATION PORT: fields contain the 16-bit UDP protocol port numbers used to demultiplex datagram among the processes waiting to receivethem.
The SOURCE PORT is optional. When used, it specifies the port to which replies should be sent; if not used, it should be zero.
2. LENGTH: field contains a count of octets in the UDP datagram, including the UDP header and the user data. Thus, the minimum value for LENGTH is eight, the length of the header alone.
3. CHECKSUM: is optional and need not be used at all; a value of zero in the CHECKSUM field means that the checksum has not been computed. The designers chose to make the checksum optional to allow implementations to operate with little computational overhead when using UDP across a highly reliable local area network.
However, IP does not compute a checksum on the data portion of an IP datagram. Thus, the UDP checksum provides the only way to guarantee that data has arrived intact and should be used.

A.6 Transmission Control Protocol (TCP)

Definition:

The TCP protocol provides a very important service in the TCP/IP protocol suite because it provides a standard general-purpose method for reliable delivery of data. Rather than inventing their own transport protocol, applications typically use TCP to provide reliable delivery of data because the TCP protocol has reached considerable maturity and many improvements have been made to the protocol to improve its performance and reliability.

TCP Header Format:



Description:

Each segment is divided into two parts, a header followed by data. The header,
known as the TCP header, carries the expected identification and control information.
1. SOURCE PORT and DESTINATION PORT: contain the TCP port numbers thatidentify the application programs at the ends of the connection.
2. SEQUENCE NUMBER: identifies the position in the sender's byte stream of the data in the segment.
3. ACKNOWLEDGEMENT NUMBER: identifies the number of the octet that the source expects to receive next. Note that the sequence number refers to the stream flowing in the same direction as the segment, while the acknowledgement number refers to the stream flowing in the opposite direction from the segment.
4. HLENS: contains an integer that specifies the length of the segment header measured in 32-bit multiples. It is needed because the OPTIONS field varies in length, depending on which options have been included. Thus, the size of the TCP header varies depending on the options selected.
5. RESERVED: This 6-bit field is reserved for future use.

6. CODE BITS: Some segments carry only an acknowledgement while some carry data. Others carry requests to establish or close a connection. TCP software uses this 6-bit field labeled to determine the purpose and contents of the segment. The six bits tell how to interpret other fields in the header according to the following table:

Bit (left to right)	Meaning if bit set to 1
URG	Urgent pointer field is valid
ACK	Acknowledgement field is valid
PSH	This segment requests a push
RST	Reset the connection
SYN	Synchronize sequence numbers
FIN	Sender has reached end of its byte stream

7. WINDOWS: TCP software advertises how much data it is willing to accept every time it sends a segment by specifying its buffer size in this field. The field contains a 16-bit unsigned integer in network-standard byte order. Window advertisements provide another example of piggybacking because they accompany all segments, including those carrying data as well as those carrying only an acknowledgement.
8. URGENT POINTER: To accommodate out of band signaling, TCP allows the sender to specify data as urgent, meaning that the receiving program should be notified of its arrival as quickly as possible, regardless of its position in the stream. The protocol specifies that when urgent data is found, the receiving TCP should notify whatever application program is associated with the connection to go into "urgent mode." After all urgent data has been consumed, TCP tells the application program to return to normal operation. The exact details of how TCP informs the application program about urgent data depend on the computer's operating system, of course. The

mechanism used to mark urgent data when transmitting it in a segment consists of the URG code bit and the URGENT
POINTER field. When the URG bit is set, the urgent pointer specifies the position
in the segment where urgent data ends

9. OPTION: Not all segments sent across a connection will be of the same size. However, both ends need to agree on a maximum segment they will transfer. TCP software uses this field to negotiate with the TCP software at the other end of the connection; one of the options allows TCP software to specify the maximum segment size (MSS) that it is willing to receive.

B Log Files

Session 1:

```
Device: eth0
Network Address: 255.255.240.0
Subnet Mask: 255.255.240.0
Buffer Size: 8192
Sniffing mode promiscuous ?: Y
Timeout :0
Packets Requested: 5


packet no 0:

Packet Captured:
Total length of packet avilable = 54
Total length captured = 54

Printing the Ethernet information:
Source Address: 0:11:25:d8:ab:dd
Destination Address: 1:0:5e:0:0:16
Packet Type: IP

Printing the IP header information:
Source IP Address: 60.243.153.1
Destination IP Address: 224.0.0.22
Protocol:IGMP: Internet Group Management Protocol


packet no 1:

Packet Captured:
Total length of packet avilable = 54
Total length captured = 54

Printing the Ethernet information:
Source Address: 0:11:25:d8:ab:dd
Destination Address: 1:0:5e:0:0:16
Packet Type: IP

Printing the IP header information:
Source IP Address: 60.243.153.1
Destination IP Address: 224.0.0.22
Protocol:IGMP: Internet Group Management Protocol


packet no 2:
```

```
Packet Captured:
Total length of packet avilable = 88
Total length captured = 88

Printing the Ethernet information:
Source Address: 0:11:25:d8:ab:dd
Destination Address: 1:0:5e:0:0:fb
Packet Type: IP

Printing the IP header information:
Source IP Address: 60.243.153.1
Destination IP Address: 224.0.0.251
Protocol:UDP: User Datagram Protocol

Printing the UDP header information:
Source Port: 59668: user-defined service
Destination Port: 59668: user-defined service

Printing the data remaining in the packet:
  124 ^ 124 124 û 124  % Ø « Ý  124 E 124 124 J 124 124 @ 124 ÿ  Ä ² < ó ™   à
124 124 û µ é µ é 124 6 ¿ j 124 124 124 124

packet no 3:

Packet Captured:
Total length of packet avilable = 88
Total length captured = 88

Printing the Ethernet information:
Source Address: 0:11:25:d8:ab:dd
Destination Address: 1:0:5e:0:0:fb
Packet Type: IP

Printing the IP header information:
Source IP Address: 60.243.153.1
Destination IP Address: 224.0.0.251
Protocol:UDP: User Datagram Protocol

Printing the UDP header information:
Source Port: 59668: user-defined service
Destination Port: 59668: user-defined service

Printing the data remaining in the packet:
  124 ^ 124 124 û 124  % Ø « Ý  124 E 124 124 J 124   @ 124 ÿ  Ä ± < ó ™   à 124
124 û µ é µ é 124 6 ? k 124 124 124 124

packet no 4:

Packet Captured:
```

```
Total length of packet avilable = 88
Total length captured = 88

Printing the Ethernet information:
Source Address: 0:11:25:d8:ab:dd
Destination Address: 1:0:5e:0:0:fb
Packet Type: IP

Printing the IP header information:
Source IP Address: 60.243.153.1
Destination IP Address: 224.0.0.251
Protocol:UDP: User Datagram Protocol

Printing the UDP header information:
Source Port: 59668: user-defined service
Destination Port: 59668: user-defined service

Printing the data remaining in the packet:
  125 ^ 125 125 û 125  % Ø « Ý  125 E 125 125 J 125  @ 125 ÿ  Ä ° < ó ™   à 125
125 û µ é µ é 125 6 ? k 125 125 125 125
```

Session 2:

```
packet no 0

The Packet Captured:
Total length of packet avilable = 60
Total length captured = 60

Printing the Ethernet information:
Source Address: 0:1b:24:63:c:16
Destination Address: ff:ff:ff:ff:ff:ff
Packet Type: ARP

ARP opcode: Undefined

packet no 1

The Packet Captured:
Total length of packet avilable = 42
Total length captured = 42

Printing the Ethernet information:
Source Address: 0:11:25:d8:ab:dd
Destination Address: 0:1b:24:63:c:16
Packet Type: ARP

ARP opcode: Undefined
```

```
packet no 2

The Packet Captured:
Total length of packet avilable = 106
Total length captured = 106

Printing the Ethernet information:
Source Address: 0:1b:24:63:c:16
Destination Address: 0:11:25:d8:ab:dd
Packet Type: IP

Printing the IP information:
Source IP Address: 60.243.153.135
Destination IP Address: 60.243.153.1
Protocol:TCP: Transmission Control Protocol

Source Port: 6660
Destination Port: 5632

packet no 3

The Packet Captured:
Total length of packet avilable = 54
Total length captured = 54

Printing the Ethernet information:
Source Address: 0:11:25:d8:ab:dd
Destination Address: 0:1b:24:63:c:16
Packet Type: IP

Printing the IP information:
Source IP Address: 60.243.153.1
Destination IP Address: 60.243.153.135
Protocol:TCP: Transmission Control Protocol

Source Port: 5632
Destination Port: 6660

packet no 4

The Packet Captured:
Total length of packet avilable = 106
Total length captured = 106

Printing the Ethernet information:
Source Address: 0:11:25:d8:ab:dd
Destination Address: 0:1b:24:63:c:16
Packet Type: IP

Printing the IP information:
```

```
Source IP Address: 60.243.153.1
Destination IP Address: 60.243.153.135
Protocol:TCP: Transmission Control Protocol

Source Port: 5632
Destination Port: 6660

packet no 5

The Packet Captured:
Total length of packet avilable = 106
Total length captured = 106

Printing the Ethernet information:
Source Address: 0:1b:24:63:c:16
Destination Address: 0:11:25:d8:ab:dd
Packet Type: IP

Printing the IP information:
Source IP Address: 60.243.153.135
Destination IP Address: 60.243.153.1
Protocol:TCP: Transmission Control Protocol

Source Port: 6660
Destination Port: 5632

packet no 6

The Packet Captured:
Total length of packet avilable = 54
Total length captured = 54

Printing the Ethernet information:
Source Address: 0:11:25:d8:ab:dd
Destination Address: 0:1b:24:63:c:16
Packet Type: IP

Printing the IP information:
Source IP Address: 60.243.153.1
Destination IP Address: 60.243.153.135
Protocol:TCP: Transmission Control Protocol

Source Port: 5632
Destination Port: 6660

packet no 7

The Packet Captured:
Total length of packet avilable = 106
Total length captured = 106
```

```
Printing the Ethernet information:
Source Address: 0:1b:24:63:c:16
Destination Address: 0:11:25:d8:ab:dd
Packet Type: IP

Printing the IP information:
Source IP Address: 60.243.153.135
Destination IP Address: 60.243.153.1
Protocol:TCP: Transmission Control Protocol

Source Port: 6660
Destination Port: 5632

packet no 8

The Packet Captured:
Total length of packet avilable = 54
Total length captured = 54

Printing the Ethernet information:
Source Address: 0:11:25:d8:ab:dd
Destination Address: 0:1b:24:63:c:16
Packet Type: IP

Printing the IP information:
Source IP Address: 60.243.153.1
Destination IP Address: 60.243.153.135
Protocol:TCP: Transmission Control Protocol

Source Port: 5632
Destination Port: 6660

packet no 9

The Packet Captured:
Total length of packet avilable = 106
Total length captured = 106

Printing the Ethernet information:
Source Address: 0:11:25:d8:ab:dd
Destination Address: 0:1b:24:63:c:16
Packet Type: IP

Printing the IP information:
Source IP Address: 60.243.153.1
Destination IP Address: 60.243.153.135
Protocol:TCP: Transmission Control Protocol

Source Port: 5632
```

```
Destination Port: 6660

packet no 10

The Packet Captured:
Total length of packet avilable = 106
Total length captured = 106

Printing the Ethernet information:
Source Address: 0:1b:24:63:c:16
Destination Address: 0:11:25:d8:ab:dd
Packet Type: IP

Printing the IP information:
Source IP Address: 60.243.153.135
Destination IP Address: 60.243.153.1
Protocol:TCP: Transmission Control Protocol

Source Port: 6660
Destination Port: 5632

packet no 11

The Packet Captured:
Total length of packet avilable = 54
Total length captured = 54

Printing the Ethernet information:
Source Address: 0:11:25:d8:ab:dd
Destination Address: 0:1b:24:63:c:16
Packet Type: IP

Printing the IP information:
Source IP Address: 60.243.153.1
Destination IP Address: 60.243.153.135
Protocol:TCP: Transmission Control Protocol

Source Port: 5632
Destination Port: 6660
```

References:

[1] Tim Carstens. Programming with pcap. http://www.tcpdump.org/pcap.htm.

[2] Martin Casado. Packet Capture with libcap and other Low Level Network Tricks. http://www.cet.nau.edu/~mc8/Socket/Tutorials/

[3] pcap man page

[4] Richard Stevens. Unix Networking Programming. Prentice Hall.

[5] Douglas E. Comer. Internetworking with TCP/IP: Principles Protocols and Architectures, 4th Edition. Prentice Hall.

[6] Wendell Odom. Cisco CCNA Exam #640-507 Certification Guide. Cisco Press

[7] Karanjit Siyan. Inside TCP/IP, 3rd Edition. Macmillan Computer Publishing.

[8] Matthew G. Naugle. Illustrated TCP/IP. John Wiley & Sons, Inc.

[9] Joe Casad, Bob Willsey. Sams Teach Yourself TCP/IP in 24 Hours. Macmillian Computer Publishing.

[10] Amit K. Saha. Coding a Simple Packet Sniffer. http://www.geocities.com/amit_saha_works/

[11] Brian W. Kernighan Dennis M. Ritchie. The C Programming Language, 2nd Edition. Prentice Hall