Instrumenting Linux to Collect the Traces of An Individual Communication Packet

Submitted to the

Department of Computer Science

College of Computing Sciences

New Jersey Institute of Technology

In partial fulfillment of

The requirements of the degree of

Master of Science

By

Gyan Shanker Phalahari

APPROVALS

Proposal Number : 210-77-707

Approved By : _____

(Dr. Teunis J. Ott)

Date Submitted : 08/04/2005

ACKNOWLEDGEMENT

I would like to express my gratitude and thanks to Dr. Teunis Ott, for his guidance and invaluable help without which this project would not have been possible.

I also appreciate the help provided by Mr. Rahul Jain during the project. The discussions we had during the project gave me further insight into the Linux Networking Code.

Abstract

In this project, we study how IP packets flow through the Linux TCP/IP protocol stack. The Linux variant used is Red Hat 9 and the kernel is 2.4.20-8.

We use two tools to study the flow of packets.

1. Tcpdump
2. Run Time Kernel Function Hijacking using Loadable Kernel Modules.

"tcpdump" is an existing tool used for packet sniffing by system administrators.

"Kernel Function Hijacking" is a method suggested by Silvio Cesare, November 1999[4]. Using this method, we have developed an extensive tool, which intercepts the Linux Kernel functions and prints the traces of packets for our study.

We use both the tools mentioned above, in parallel, to capture details of the packets. Our tool records the details in a log file. We record time, source address, destination address and few more details as packets are seen by the kernel functions. The log file is cleaned to make it suitable for presentation using a Perl script.

We discuss a general handling of packets by the kernel functions and then verify our learning by looking at the log files. The protocols covered in this project are IP, TCP and UDP.

TABLE OF CONTENTS

List of Diagrams

1.0 Introduction

1.1 Problem Statement

The Linux network protocol stack is part of the kernel and is embedded in the kernel. The protocol stack forms a carrier and a pipeline of the data from one host to another. It is modular in design and we can interact with the different layers separately. We are always curious to know what happens inside these individual layers. How packets are handled? What are the important functions that implement a layer? How much time does a packet spend in a particular function? Answer to these questions are important if we want to improve the working of the protocol stack, design a new protocol stack, eliminate weaknesses of the present protocol stack and make the present implementation more secure.

"Instrumenting Linux to collect the traces of an individual communication packet" is a project to study the TCP/IP implementation of Linux. It is a small step in this direction. This project also records the time spent by the individual packets at various kernel functions. This gives us an idea of resources consumed by a function. The protocols covered in this project are Transport layer protocols TCP & UDP and network layer protocol IP.

We use two tools to see the packets that flow from one machine to another during data transmission.

1. "tcpdump", an existing tool. Refer to man pages of the tcpdump for further details.

2. The second tool is our own tool. It intercepts the packets at pre-decided kernel functions; records the time and packet details. A detailed explanation of the tool is given in the section "Loadable Kernel Module".

1.2 Glossary of Terms

SKB: Socket Buffer, A kernel representation of network packets

IP: Internet Protocol

TCP: Transmission Control Protocol

UDP: User Datagram Protocol

Host: A computer, which may be working as a standalone machine or may be connected to some network.

Router: A host, which has at least two active network interfaces and has the capability of forwarding IP datagram.

Source: A host, which initiates the TCP/IP connection.

Destination: A host, which participates in the TCP/IP connection initiated by the Source.

Internet: The vast collection of interconnected networks that all use the TCP/IP protocols and that evolved from the ARPANET of the late 60s.

DNS: Domain Name Server

GOME: GNU Network Object Module Environment

MTU: Maximum Transfer Unit

RTT: Round Trip Time

DMA: Direct Memory Access

LKMs: Loadable Kernel Modules

ELF : Executable and Linkable Format

2.0 Design

2.1 Choice of Alternatives

Red Hat Linux 9 and kernel 2.4.20-8 was used for the project. It is free and a stable version of the Linux kernel . It is available at the Internet lab, NJIT.

There were two possible approaches to the project, which we could have taken.

1. Implant probes within various networking function in the kernel, recompile the kernel and collect readings.
2. Use kernel modules and collect readings.

Recompiling the kernel after changing approximately 30 kernel functions and collecting the traces of a packet is a big task. More so, if a small change is required the whole process has to be redone. Taking into account that one compilation takes approximately 45 minutes of time, this was bound to become very difficult or even unmanageable task. In addition to that, we do not have the flexibility of inserting and removing the probes at runtime.

All these disadvantages led to the consideration of the second alternative, which was using Loadable Kernel Modules(LKMs).

Using the LKMs, our code could be inserted or removed at will and any changes in the design could be brought about easily even at a later stage of development. Moreover, compilation of LKMs takes very little time, as the whole kernel need not be re-compiled. (A few seconds as compared to 45 minutes of kernel compilation.)

The second approach was not free of disadvantages. The limitation was that we could not instrument any kernel function that was declared as “static inline” in the code. The reason for this will be stated when we describe our instrumentation methodology later in the project.

The measurement of time will not be as accurate as it would have been, had we considered the first approach. In our methodology, we record the time just before the kernel function is entered whereas in the first method, we could have recorded the time after entering the kernel function. Nevertheless, the accuracy gained in LKMs approach was good enough for the purpose of this study and we shall live with it.

As stated earlier, since we could not attach our timer device to “static inline “ functions, the second approach did not work in totality and specially in TCP. Many functions in TCP implementation are “static inline.” We had to change the kernel source code to intercept these functions. We removed the “static inline” from the beginning of some of the kernel functions and recompiled the kernel to suit our requirement.

Thus instead of relying totally on the 2nd approach, we used a hybrid method of changing the kernel code and using the LKMs in this project.

2.2 Functional Specifications

2.2.1 General Methodology

Runtime kernel function hijacking using Loadable kernel module was the method adopted to collect the traces of packets across various networking functions of the kernel. Details of how the tool is used for data collection are available in section **Loadable Kernel Module**. Only an overall picture of data transmission through a Linux kernel and our tool's interaction with the kernel is being presented.

Using our tool, we actually fool the kernel to call "**our function**" whenever it wants to call a kernel function, in the following example tcp_v4_rcv or __tcp_v4_lookup. Our function prints the time, packet details and allows the kernel function tcp_v4_rcv or __tcp_v4_lookup to continue its processing. Once these kernel functions finish their processing, the control is returned back to our function and we once again record the time and the packet details. Then we transfer the control back to the kernel to continue its working. The nesting of the lines in the log tells us the function call mechanism of the Kernel.

Once we use our tool, for every function instrumented we have packet's entry and exit time apart from other details. If we know how much additional "time cost" our measurement procedure introduces, we can tell what is the time consumed by any intercepted kernel function for processing a packet. Of course, this time will vary from one computer to another and on the same computer, it will be based on number and nature of other processing tasks the computer is handling. Most of our readings are taken

on a computer not running any other user processes apart from the ones required to take the readings.

Packets traverse the functions of Linux network protocol stack in two ways:

1. Successive calls of function.

What this means is that, the first function calls the second function and the second function calls the third function and so on. Then all the functions return the control to their caller.

In order to see the function calls, we have indented the log file suitably so that how one function calls the other is clearly visible.

In the following example, tcp_v4_rcv function calls __tcp_v4_lookup, thus there is an indentation before __tcp_v4_lookup. All the functions on the same level are vertically in the same column. This is very intuitive and merely a look at the log file shall give the idea how it is organized.

2. Deposit in a Queue.

A function deposits the packet in a queue and then based on the protocol's processing stage, the next function is called, or the scheduler schedules the next function when appropriate. All such places have been explained while explaining the code. Refer to section on **queuing** for further details.

A sample log of functions visited by a packet is :

```
35    19:26:16:525001 128.235.204.81:21 192.168.1.20:33056 tcp_v4_rcv B P 2291929752 ack 516203532
36        19:26:16:525109 128.235.204.81:21 192.168.1.20 __tcp_v4_lookup B
37        19:26:16:525180 128.235.204.81:21 192.168.1.20:33056 __tcp_v4_lookup E
38    19:26:16:525265 128.235.204.81:21 192.168.1.20:33056 tcp_v4_rcv E P 2291929752 ack 516203532
      Time : 264
```

2.2.2 Organization of the Log File

Let us discuss organization of the log file. The first column of the log file is the line number and the second is the time. The third column is the source IP address and the source port. The fourth column is the destination IP address and the destination port. The fifth column is the function name (line 35 and 38 **tcp_v4_rcv**). Sixth column is either B or E.

B stands for beginning of a function. Actually, it is just before a Kernel function is entered i.e. just before we call a kernel function.

E stands for end of a function or Just after a kernel function's execution is over i.e. just after the control returns to our function, which fooled the kernel.

We have added our own network layer protocol using LKMs. The details will be explained later in the IP section of this report. When we see a packet at the **netfiler hooks** or at our **own protocol**, we do not have B/E at the 6th column. Instead we have '-' as no processing is done at these places and there is no beginning and end of a kernel function. Details about netfilter hooks can be found in the IP section of this report.

These six columns are printed for all the protocols that we study in this report. There are more columns if the packet being handled belongs to TCP. The seventh Column gives the status of TCP flags SYN/FIN/PUSH by letters S/F/P. In case none of the flags are set, it is represented by a '.'. The eighth column gives TCP sending sequence number and the 9th Column gives TCP acknowledgment sequence number. These columns are printed even if we are working at the IP layer, as long as the transport layer protocol is TCP.

When we intercept a kernel functions that belongs to the IP layer, we have a 10th Column giving the IP identifier number and a 11th column giving the transport layer protocol by name. Only TCP, UDP & ICMP are recorded in the 11th column and "unknown protocol", if none of them.

Time is very important in our readings, so we have considered various approaches to obtain the time in a Linux kernel. Refer to the section **Time in Linux Kernel** for further details. The time printed in the log file is in the format hour: minute: second: microsecond.

After a group of functions is traversed by a packet, we write the total time elapsed in microseconds. In the above example, it is 264 microseconds.

2.3 Recording of Packet Details

2.3.1 Generation of Network Traffic

We require a method to generate network traffic. In general, we use FTP to transfer a small file from one computer to another. This generates the necessary TCP traffic so that we can take our readings.

While intercepting the UDP kernel functions, we needed a way to send a UDP datagram. We wrote customized client and server programs, which will generate UDP traffic. The workings of the programs are explained in the UDP section of this report. The programs are udp_client.c and udp_server.c.

2.3.2 Network Topology Used

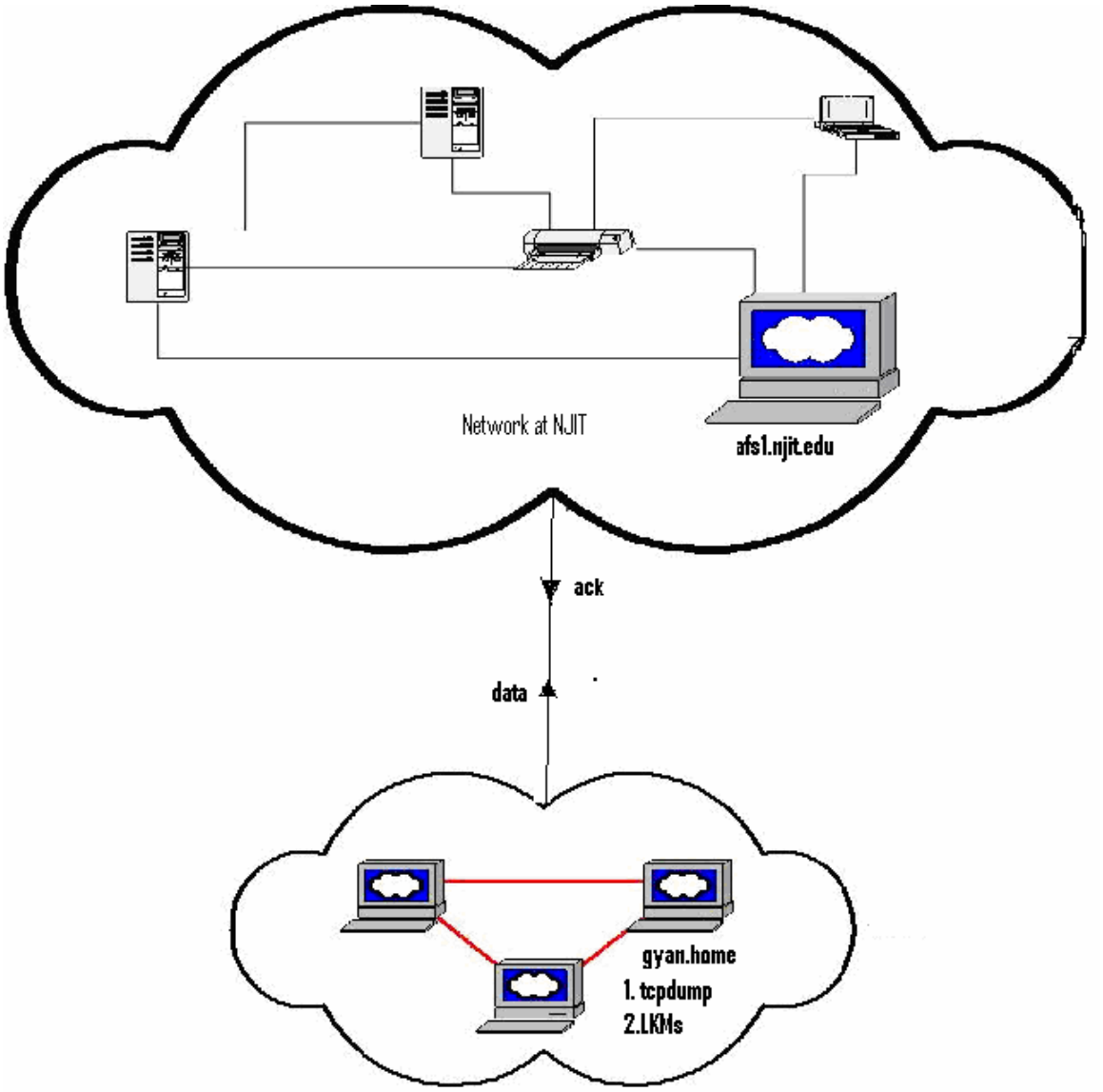


Figure – 1: Transmission of Data from gyan.home to afs1.njit.edu

The computers used for data transmission and produce the log file were inside the Internet Lab and mostly 'Franklin ' as the source where the instrumented kernel was

running. My home computer "gyan.home" was also used as source for many readings so that the real world Internet traffic could be intercepted. While working with the TCP layer, we experienced that the logging mechanism started losing the packets. This happened more when the transfer of a file was done inside the lab. This was primarily because of the speed with which successive packets arrived. When my home computer and actual Internet traffic was used, the speed of transmission was reduced and the loss of packets became negligible.

Figure -2 : Computers inside the Internet Lab, NJIT with Franklin as source.

2.3.3 Cleaning of Log Files

All the modules give a raw output in the file /tmp/packet.log which must be cleaned for more readability. We add the indentation and line numbers using a perl script. All the log files seen in this project are cleaned log files. The perl script is included in the appendix along with other source code.

2.4 Command Line Parameters to the Kernel Modules

Loading of LKMs in the kernel required command line parameters to be passed to the programs. These parameters are essentially memory addresses of kernel functions intercepted in LKMs, in hexadecimal notation. They are available in System.map file which shall be explained later in the **Loadable Kernel Module** section.

Passing the parameters manually is an alternative, but writing a shell script can automate the whole process of passing the parameters and loading the LKMs. We have written shell scripts to extract the requisite information from the System.map file, pass the parameters to the LKM and load the modules. We call these shell scripts as "module loader" in this project.

2.5 Organization of the Project Source Code

We have written LKMs to intercept the packets and produce a log file. The LKMs are different for TCP, IP and UDP. Even inside an individual protocols say IP, there are multiple LKMs. Every LKM resides in its own file. The layer wise organization of the files is being explained.

All "x.c" files contain LKMs, and "x.sh" files are corresponding LKM loaders.

General LKM:

1. tcpip.c contains a module which must be loaded for the entire duration of taking the readings. It provides the necessary memory buffer, where all LKMs will record the details of the packets seen by them. See the Taking Reading Section for further detail.

TCP layer :

2. synfin.c and corresponding module loader synfin.sh
3. tcpin.c and tcpin.sh
4. tcpout.c and tcpout.sh
5. tcp_prot.c and tcp_prot.sh

UDP layer:

6. udpio.c and udpio.sh

IP layer:

7. myip_send.c and myip_send.sh
8. myip_rcv.c and myip_rcv.sh
9. myip_forward.c and myip_forward.sh

Miscelaneous:

10. Makefile to compile all the above 'x.c' file and convert them to corresponding 'x.o' file for loading .

11. Perl script called ‘format’ to clean the log files.
12. udp_client.c and udp_server.c for generating UDP traffic.
13. module_header.h is a header file which provides the definition of constants used in the project. It also contains definitions of the common utility functions.

All the above 22 files are required to run the project efficiently. All the files must be in the same directory except perl script 'format' which can be in any directory where the user desires the cleaned log file.

3.0 Taking the Readings

Procedure used for generating the Log files discussed in this project is being presented.

1. We compile all “x.c” files using the Makefile which produces “x.o” files. Here the ‘x’ can be replaced by any of the ‘.c’ files described in the previous section.
2. Load the tcpip.o file by 'insmod tcpip.o'. This is a must for all the other modules to work. It provides big_buffer variable which is a global variable and is seen by all other modules as long as tcpip.o is loaded in the kernel. This is where all other modules store the information they want to log.
3. Load the desired module by executing the shell script associated with the modules e.g., if we want to load myip_send.o, we will give the command “./myip_send.sh”.
4. Start the tcpdump with –w option to record the packet details in a file.
5. FTP a file from the present computer to some other computer if TCP traffic is desired. Otherwise, use UDP client and server programs if the UDP traffic is desired. I have not tested extensively but I feel while working with IP layer we could also use PING to generate ICMP traffic and study the activities of kernel functions in the IP layer.
6. Unload the previously loaded modules by giving a “rmmod” command say 'rmmod myip_send'. Please note that while giving the rmmod command, we do not include '.o' extension of the file. All the loaded modules will have to be unloaded separately and tcpip.o must be last module to be unloaded.
7. As soon as a module is unloaded /tmp/packet.log file is created or if the file already exists, the results are appended.
8. Clean the log file by running the perl script 'format'. Give command './format >clean_log'.

The format script knows that it has to clean /tmp/packet.log file.

The cleaned log file will be available in the file with name clean_log as a result of above command.

9. Stop tcpdump and read from the file produced by the tcpdump.

4.0 Linux Network Protocol Stack Overview

4.1 The Protocol Stack

An Overview of the network protocol packet of Linux is being presented. Explaining the functionality of a Linux kernel protocol stack in totality and in detail is beyond the scope of the project and not possible within the stipulated time. This is just an overview and not the complete picture as many books are written alone on this subject. The short explanation that follows will leave out many details. Nevertheless, the details presented here are sufficient for a basic understanding of the working of a protocol stack in Linux. Many details are further explained in TCP/UDP/IP sections, which follow. This overview assumes basic understanding of network communication. The protocols themselves are not explained. Familiarity with basic socket programming is assumed and the details related to socket programming are not explained.

Network Devices constitute the bottom layer of the protocol stack. They use a Link layer protocol for communication with other devices. The link layer protocol we work with in this project is Ethernet.

Device drivers are a family of software present in the kernel, generally as loadable module, to manage various devices attached to the computer including the Ethernet cards. The packets arriving at the interface of a card from the external world are copied by the device drivers. After performing some error checks, the packets are transferred to the network layer, in our case IP. Similarly, outgoing packets received from the network layer (read IP) are sent over the physical medium by the device drivers after performing an error check. We do not discuss the link layer functionality of a kernel in this project.

Refer to Figure -3 for details. The figure uses TCP over IP as an example.

The main functionality of IP, our network layer protocol, is to provide routing information to the individual packets. It checks every incoming packet to see if they are meant for the host computer, else the packets need to be forwarded to the right computer. It de-fragments the incoming packets and passes them to the transport protocol. For all the outgoing packets, IP maintains a dynamic database of routes. It addresses the packets and fragments them if necessary, before handing over to the link layer.

TCP and UDP are the transport layer protocols used in the project. UDP simply provides a mechanism for addressing the packets to the ports within a computer. TCP allows more complex connection based operations, which facilitate recovery of lost packets as well as traffic management. TCP copies the data from user address space to kernel address space for transporting. For UDP data is copied from user address space to kernel space by IP, with the help of UDP function as function pointer. We shall see them while discussing the individual protocols.

Moving up the transport layer is the INET layer. It is the intermediate layer between the application sockets and transport layer. The INET layer implements the sockets owned by the applications for the kernel. All socket specific operations are implemented here.

In order to provide abstraction to the application programmer, another interface is provided with the help of BSD sockets. BSD sockets are abstract data structures containing INET sockets. Application's request to connect, read, write through socket identifier is converted to INET operations by BSD. Refer to “Unix Network

Programming" by Richard Stevens[4] for further details. We represent BSD sockets as Socket layer in this project.

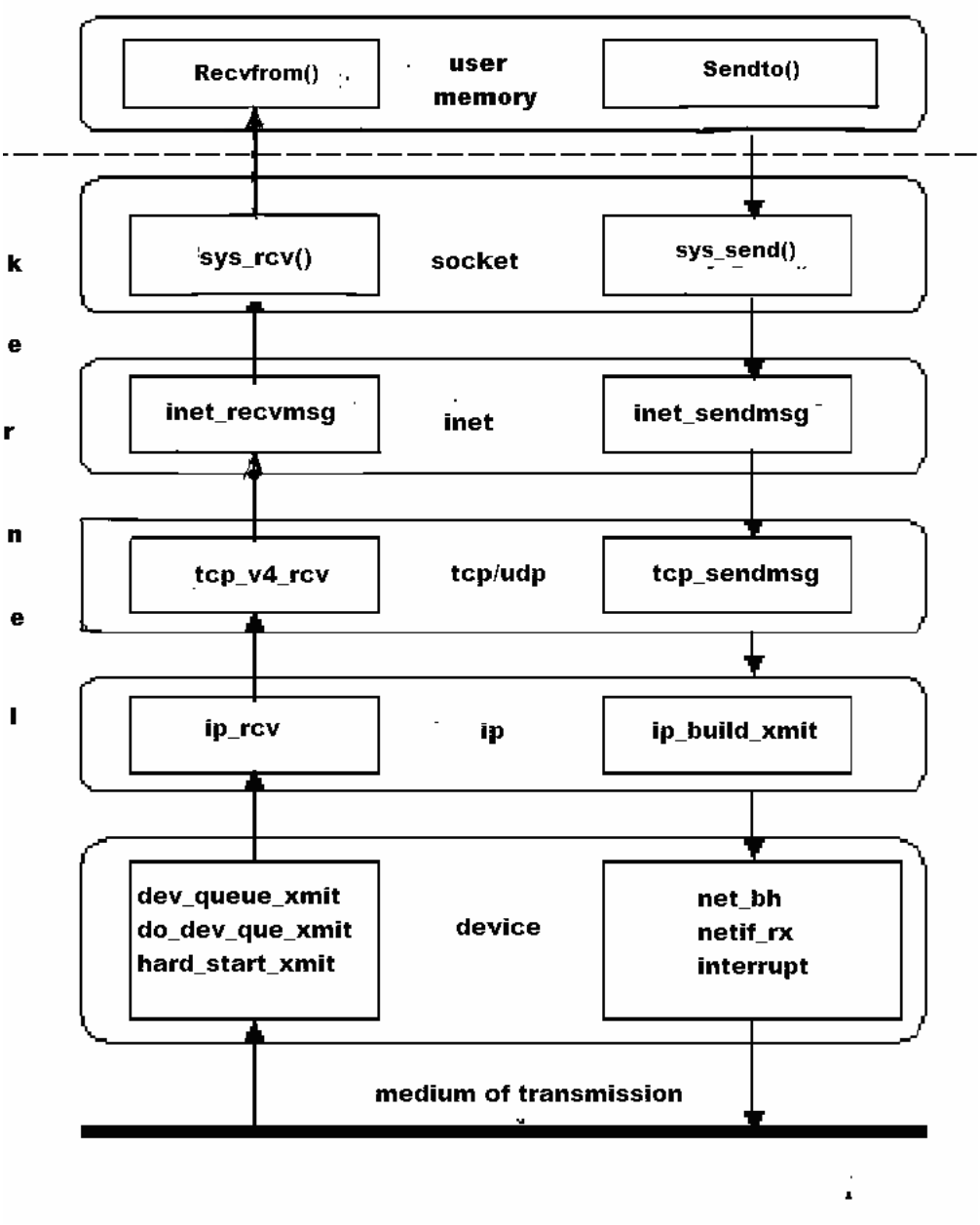


Figure -3 : The Linux Network Protocol Stack
(Transport layer considered is TCP)

(source: Analysis and Evaluation of TCP/IP Protocol stack of Linux, Institute of Communication Engineering, China[7])

4.2 Socket Buffer

Socket Buffer (also referred to as skb in this report) is the representation of a packet inside the Linux kernel. It is actually a struct from the C programming perspective and is defined in /include/linix/skbuff.h. A packet is stored in a contiguous memory and skb contains various pieces of information including the pointers to the various places of the packet. In a layered protocol as TCP/IP the head of the upper layer is added to the data passed from the upper layer to the lower layer. The head of the lower layer is stripped off when data is passed from lower layer to upper layer. The method used in Linux is to calculate and allocate maximum amount of memory needed to represent a packet including its various headers and data. When a packet is passed from one layer to another, skb is passed and the head or tail pointers of the skb are moved.

The data structure sk_buff has following important parameters:

head : points at the beginning of the packet

end : points at the end of the packet

data : pointer at the beginning of the valid data of the packet. This pointer changes as we move from layer to layer.

tail : pointer to the end of valid data

There are also functions for manipulating the skb.

skb_push : puts the data at the beginning of the packet pointed by the skb.

skb_put : puts the data at the end of the packet pointed by the skb

skb_pull : removes data from start of a packet pointed by the skb. A pointer to the next data in the buffer is returned. Once data has been pulled future skb_push will overwrite the old data.

The figure below shows how the socket buffers are arranged in a linked list and important data structures contained within the skb. Please refer to "The Linux Network Architecture[6] for further details.

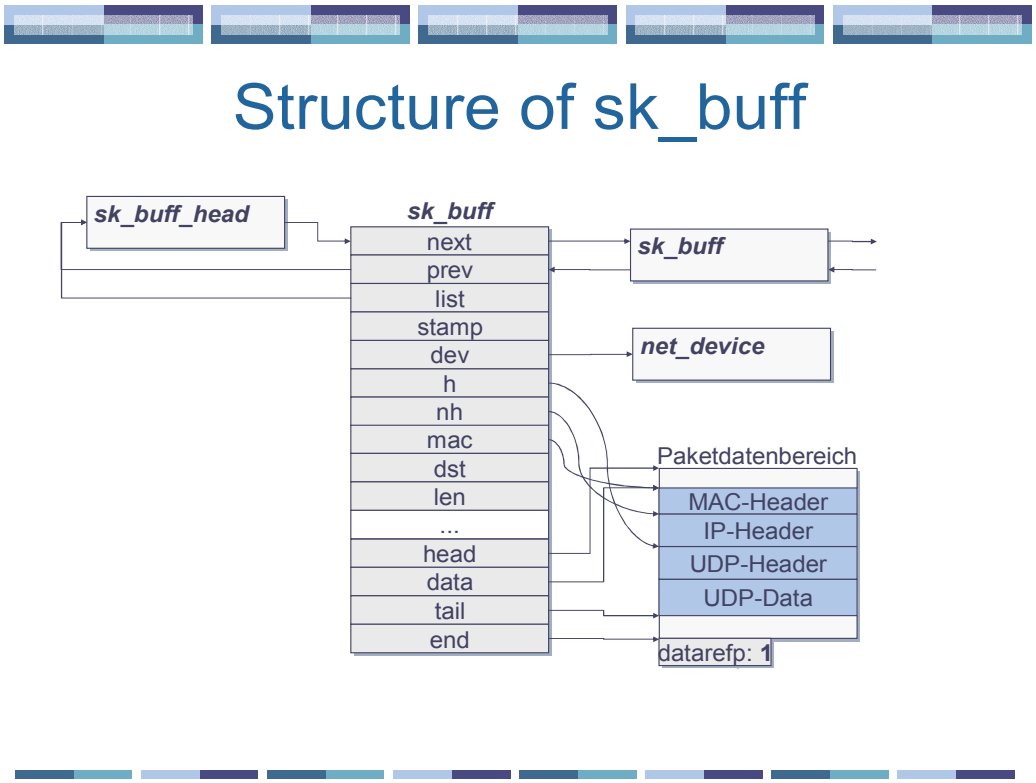
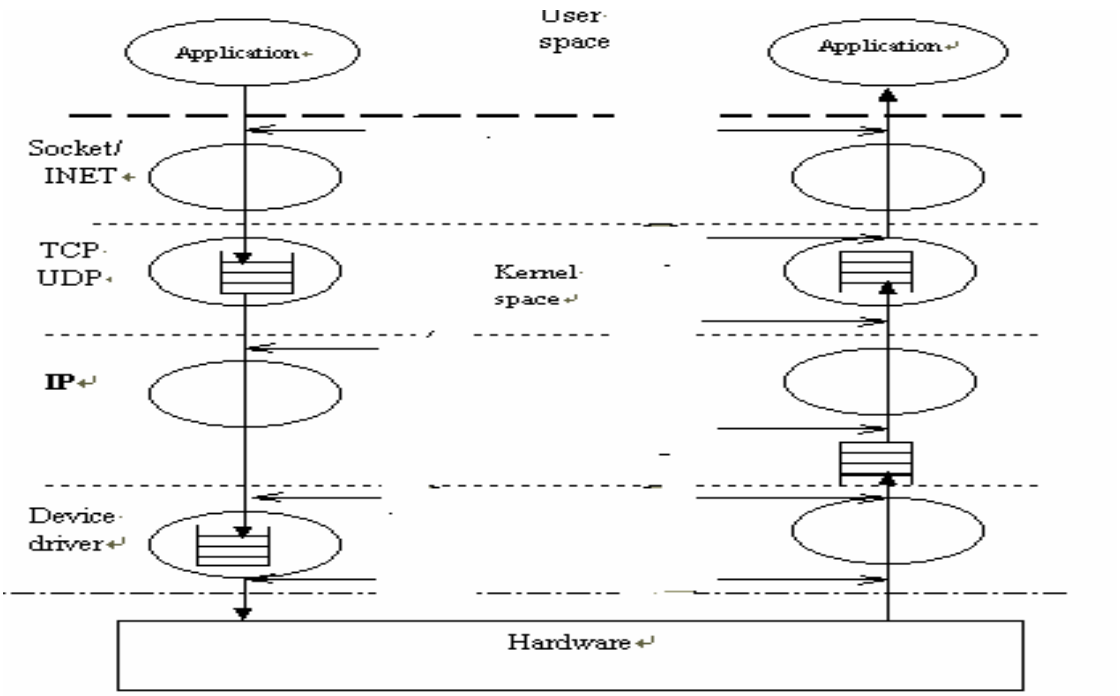


Figure - 4: Structure of a Socket buffer in a Linux Kernel (source: University of Illinois, Dept of CS[5])

4.3 Queuing in the Kernel Protocol Stack

TCP/IP is a store and forward protocol. There are queues in the protocol stack implementation. The location of the queue, its allowable maximum length, default treatment of overflowing packets and the methods used in queuing and de-queuing the packets greatly impact the efficiency of the protocol. Queue inside kernel protocol stack exists in two places in sending and receiving path while handling TCP packets. On the sending path, queue exists at the TCP layer and the device layer. On the receiving path, the queue exists at the IP layer and at the TCP layer. The queues at the TCP layer are based on the total number of bytes and the queues at the IP and device layers are based on the total number of packets.



Fugure - 5: Queues in TCP/IP protocol implementation
(source: Analysis and Evaluation of TCP/IP Protocol stack of Linux, Institute of Communication Engineering, China[7])

The queues at the TCP layer are for buffering and flow control. The queue at the sending side of the device layer is for buffering the data when the packet generation of IP exceeds the sending speed of the physical device.

At the receiving side, all the packets received are placed in backlog queue waiting for IP processing.

The default queuing system of the protocol stack is first come first serve. It discards all the packets irrespective of the content when maximum length of the queue is reached.

5.0 Loadable Kernel Modules

5.1 Structure of a Kernel Module

A Linux loadable kernel module is a set of functions and data types, which is compiled independent of a kernel and then dynamically loaded in a running kernel. A Kernel typically will have many modules implemented and a list of modules presently loaded can be obtained by typing *lsmod* at the command prompt in a Linux machine.

Loadable Kernel Modules (LKMs) are generally used to implement device drivers but can also be used to do various other tasks requiring a kernel mode of execution. We have implemented all our probes to the various kernel functions in TCP/IP stack of Linux using LKMs. An understanding of LKM is a prerequisite to understand the methodology adopted in this project, hence a brief about LKMs is being presented. For a detailed understanding please refer to The Linux Kernel Module Programming Guide[8] .

A minimum module interface is to have two functions that a kernel can call, when the module is loaded, via init_module() and unloaded via exit_module(). The init_module() function is executed while loading the module and exit_module() is executed while unloading. A typical module has the following structure:

```
#define  MODULE
# define   __KERNEL__
#include < linux/kernel.h >  /*Needed for all modules */
#include < linux/module.h > /* Needed for KERN_ALERT */
                ...
int init_module() {
   /* code to init the module */
return 0;
}
   .              ..
void cleanup_module() {
/* code to close the module */
}
```

There can be other functions in a module and these functions are either called from within the init_module() or from within the cleanup_module(). These functions are generally present in the same file that contains the init_module() and the cleanup_module().

Let us look at the above code line by line.

The first define #define MODULE tells header files to give appropriate definitions for module. The second # define __KERNEL__ symbol tells the header files that the code will be run in a kernel mode rather than as a user process.

The first include lines state that it is a kernel module and all the modules need to have this. The file is also needed for macro expansion if we use a "printk" command. The "printk" is a printing command used in a kernel module programming as against the "printf" used in a user space C program. The syntax of the printk is

int x=1;
printk("<1> The integer is %d " ,x);

This syntax of printk is a familiar concept to those who know the C style printf, but with a small variation. To understand this let us consider how user interacts with a linux machine. There are two possible ways

1. Using Virtual Console(or virtual terminal as they are sometimes called). These consoles are managed by the kernel. Each virtual terminal is connected to a corresponding device file represented by /dev/ttyt[0-9].

2. The second method is X window system commonly called 'X'. This graphical user interface comes with various Linux distributions. All command line tools and most applications that can run in a console can also run in the X. There are various applications specifically written for the X. X windows can be started from a console by giving the

command “startx”. KDE is an example of a desktop environment built over the X-windows that supply users with a modern look and feel.

The output of a printk print statement in a module will not be visible if user is interacting with the Linux machine using a X windows. The output of printk will only be visible if the user is interacting with the Linux using a console. It is always better to do kernel programming using a console. We can switch to the console mode of interaction by pressing <alt> <del> <f[1-6]> from the X. All the printk messages can be viewed in this mode.

printk is actually used for logging the kernel activities and <1> in the printk represents the priority of logging. There are eight priorities defined in linux/kernel.h. We would be comfortable to use phrases like “KERN ALERT” instead of numbers like <1> and these phrases are translated to the numbers by definitions stored in the kernel.h file.

5.2 Compilation of the Modules

All modules in this project are compiled using the following make file:

```
WARN    := -W
INCLUDE := -isystem /lib/modules/`uname -r`/build/include
CFLAGS  := -O6  $(WARN) $(INCLUDE)
CC      := gcc
OBJS    := $(patsubst %.c, %.o, $(wildcard *.c))

all: $(OBJS)

.PHONY: clean

clean:
        rm  *.o
```

Explanation:

-W : The compiler warnings at the time of compilation is turned on

-isystem /lib/modules/`uname -r`/build/include : The modules must be compiled against kernel headers of the kernel being compiled against. If we go to the /lib/modules sub directory we can see kernel headers of all the kernels we have on machine in separate subdirectories. uname -r gives the name of kernel presently on and thus helps us to choose the right header files. We should not use /usr/include/linux in compilation of modules.

-O6 : the compiler optimization must be turned on while compiling modules

$(patsubst %.c, %.o, $(wildcard *.c)) : Wildcard expansion takes place automatically in the rules of a make file but it does not happen when variables are assigned or inside the arguments of a function. Since we want all the .c files to be compiled, we are using wildcard function $(wildcard pattern....) as an argument to the function ***patsubst***. The string substitution function *patsubst* has the format *$(patsubst pattern*, *replacement*, *text)*. What it does is that it finds white space separated words in the *text* that match *pattern* and replaces them with *replacement*.

Finally, the implicit rule of a make file is invoked to compile a C program.`x.o' is made automatically from `x.c' with an implicit command of the form `$(CC) -c $(CPPFLAGS) $(CFLAGS)'

5.3 Installing and Uninstalling Modules

When we run the make file above, we will have all ***'x.c'*** file compiled and made a corresponding ***'x.o'*** file, ready to be inserted into the kernel.

The modules are installed using insmod command e.g. insmod abc.o. As soon as modules are installed, the init_module() function is executed if installation is successful.

The modules are uninstalled using rmmod command e.g. rmmod abc where abc.o is the previously loaded module. As soon as we issue this command, the exit_module() function is executed and the module unloaded.

5.4 Passing Command line Parameters

We can also pass command line parameters at the time of loading the modules as we can do while executing any executable file designed to accept command line parameters. The parameters are not argc and argv as in a standard C program. We use the macro *MODULE_PARM(xxx,"i")* inside the module and while passing the parameters we do *insmod abc.o xxx=12*.

Inside the module we have following global declaration:

```
int xxx=0;
MODULE_PARM(xxx , "i");
```

The point to be noted is that the variable name inside the module and the passing parameter are both xxx and that is how insmod resolves which parameter is meant for which module variable, as multiple module parameters can be passed. "i" in the MODULE_PARM macro stands for integer. We can also pass strings, arrays and other types of arguments. In our LKMs we have used array variable of the form

```
int myarray[4];
```

MODULE_PARM(myarray, "2-4i");

The i of "2-4i" says that the array is of integers . The prefix 2-4 before i means that we will supply a minimum of two command line arguments and a maximum of 4. We do not use this feature and always supply all the parameters to the module in this project. Following example is only for the demonstration of the above concept.

insmod abc.o myarray=2,4

insmod abc.o myarray=2,4,5

insmod abc.o myarray=2,4,5,6

All the above are valid insmod as we have to supply minimum of two and maximum of 4 parameters. In first case myarray[0] will be assigned 2 and myarray[1] will be assigned 4. In the second case apart from previous assignments myarray[2] will be assigned 5 and in the last case apart from all the previous assignments, myarray[3] will be assigned 6.

5.5 Modules and Symbols

In the context of programming symbols are building blocks of a program. Variable names and function names are symbols of a program. Similar to other programs, a kernel also has symbols. Only the difference is that the kernel being complicated piece of program, has many global symbols.

Modules are designed and compiled in the user space separate from compilation of a kernel, thus cannot use all the kernel symbols directly. Instead they can access kernel symbols which are exported using EXPORT_SYMBOL directive in the kernel code. A list of symbols exported is contained in the file /proc/ksyms. This file also tells us the syntax of exporting a symbol from the kernel code. If we cat the file we can see all the exported symbols.

As a result of the export of symbols in the kernel, we can access the symbols like tcp_prot(a data structure) or ip_rcv (a function) directly inside a LKM by defining them as extern in the code and using them.

*extern int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt);*

Now ip_rcv can be used directly in the module code like any other function locally defined. "**insmod"** acts as a linker and takes care of linking the LKM code to the piece of the kernel code that gives the definition of ip_rcv.

If we see /proc/ksyms file closely, we can also see the memory addresses in hex where the symbols are located in the kernel. A typical listing will look like this

```
c0248190 get_option_Rb0e10781
c0248200 get_options_R0fbff9b9
c0248340 rb_insert_color_Raa2b5a22
c02485f0 rb_erase_Rda226a80
.....
.....
```

The first column is the address of a symbol and the second column is the symbol name. Attached to the symbol names in the second column is the kernel version information. This information is added to all the symbols when kernel is compiled with CONFIG_MODVERSIONS set. Refer to Kernel Module Programming Guide[8] for further information on kernel modules and MODVERSIONS.

Not all the symbols present in the kernel are exported and if we want to see a complete list, we will have to look at another file namely System.map. This file is created every time a kernel is compiled and is present in /boot directory. Every compilation normally leads to change in symbol's address so kernel compilations results in new System.map file in /boot directory having name say System.map-2.4.20-8 . The /boot/System.map file is a symbolic link to this file. When we boot a particular kernel, System.map file is

automatically linked to the correct System.map-X.X.XX. file.

System.map is a "map" of the kernel. It contains info about the entry points of the functions compiled into the kernel and the de-bug information. The kernel itself knows the addresses and entry-points, but this file is needed for other programs, which need info about the kernel.

What if System.map is not available? This file is an absolute necessity for this project and we are using the file extensively. Without this file, the concept of the project cannot be taken forward. We can create the System.map file using a 'nm' command from the uncompressed kernel image file

nm -a /boot/vmlinux-2.4.20-8 >System.map

Details about 'nm' can be obtained from the man files of a Linux machine.

All the functions and the variables defined in a kernel module are by default exported and available for other modules as long as the original module is loaded. This is a default behavior of a kernel module. In case we want to alter this behavior, we can give the directive EXPORT_NO_SYMBOLS inside the kernel module. This prevents the default exporting of symbols by a kernel module.

6.0 Kernel function Hijacking

6.1 General Methodology

All our modules of the project are based on the above concept and all the previous discussions regarding the LKMs will be used while explaining this methodology.

When a Linux kernel is loaded, we have only loader in the memory and it is always loaded in big contiguous area of real memory whose virtual address is equal to the real address. Thus when we cat /boot/System.map file all the address shown are the places in the kernel memory where the symbols are present. What if we modify something at a particular memory address? The Symbol present at that address is effected and if done randomly, it may result in the kernel crash. It is important to understand that a module has same rights and responsibilities as a base kernel. A crash in a module is a crash of kernel, nothing less.

If modification of a symbol is done judiciously, we can use the method to our advantage and play around with any of the kernel functions, barring a few ones.

6.2 Typical Kernel Function Hijacking Code

Let us assume that we are interested in changing the behavior of a function ip_rcv whose System.map entry obtained by giving : 'cat /boot/System.map | grep -w ip_rcv' is as follows:

c020a020 T ip_rcv

The function ip_rcv has the following signature in the kernel code and is defined in net/ipv4/ip_input.c

*int ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt);*

The general format of our modules in this project is being presented. In a normal kernel module, we have to include all the necessary files which are required to define a kernel module, as well as kernel header files which define the various structs like struct sk_buff, struct net_device, struct packet_type etc which we shall be using. For the sake of convenience, we have included all these files and all other global definitions in a specially created file called module_header.h. We shall use this file in this project.

```
/*demo.c  a demo file for concept presentation */

# include "module_header.h"

static int allFunAddr[1] = { 0} ;
MODULE_PARM(allFunAddr, "1-1i");


static unsigned char pr_jump[NUM_BYTES]="\xb8\x00\x00\x00\x00" /* movl $0,%eax
*/
                                "\xff\xe0"; // jmp *eax
static unsigned char pr_save[NUM_BYTES];
static int (*pr)( struct sk_buff *, struct net_device *, struct packet_type *);

char big_buffer[BUF_SIZE];

 int init_module() {

      int slock_flags;

      pr=(int (*)(struct sk_buff *, struct net_device *, struct
packet_type*))allFunAddr[0];

      *(unsigned int *)(pr_jump+1)=(unsigned int)changed_ip_rcv;

      LOCK_KERN;

      _memcpy(pr_save,pr,NUM_BYTES);
      _memcpy(pr,pr_jump,NUM_BYTES);

      UNLOCK_KERN;
      return 0;
}
```

```
void cleanup_module(){
        int slock_flags;
         LOCK_KERN;
        _memcpy(pr, pr_save,NUM_BYTES);
       UNLOCK_KERN;

        if(strlen(big_buffer)>0){
                print_buffer(FILE_NAME,big_buffer,strlen(big_buffer));
                big_buffer[0]='\0';
        }

}



int changed_ip_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt)
{
        int slock_flags;
        int retval;
        char store_time[20];

        my_time(store_time);
        strcat(big_buffer,store_time);
        strcat(big_buffer," ");
        strcat(big_buffer,"ip_rcv");
        strcat(big_buffer," ");
        strcat(big_buffer , in_ntoa(skb->nh.iph->saddr));
        strcat(big_buffer," ");

          LOCK_KERN;
        _memcpy(pr, pr_save,NUM_BYTES);
        UNLOCK_KERN;

        retval=pr(skb,dev,pt);

        LOCK_KERN;
        _memcpy(pr, pr_jump,NUM_BYTES);
        UNLOCK_KERN;


        return retval;
}
```

6.3 Explanation of the Code

6.3.1 Global Declarations

static int allFunAddr[1] = { 0} ;

We declare an array of integers of length 1 and call it allFunAddr. We make it static so that this can be addressed only from within this module file and its memory for storage is allocated only once.

The default characteristic of an LKM is to exports all its symbols to be used by other modules unless otherwise we make a symbol static.

MODULE_PARM(allFunAddr, "1-1i");
The command line argument given to the module at the time of installing the module is assigned to variable allFuncAddr[0];

The syntax of installing a module with parameters is :

insmod xyz.o allFunAddr=0xc020a020

After successful installation the variable allFunAddr[0] will be assigned 0xc020a020, the address of ip_rcv which we got from System.map.

static unsigned char pr_jump[NUM_BYTES]="\xb8\x00\x00\x00\x00 \xff\xe0" ;

This is in reality two assembly instructions assigned to the variable pr_jump which is an array of 7 byte. (NUM_BYTES is 7 and is defined in the file module_header.h.)

The first instruction is

movl $0,%eax

which when translated to hex is "\xb8\x00\x00\x00\x00"

The second instruction is *jmp *eax*, which when translated to hex is "\xff\xe0"

This effectively says

1. move the address which is the argument of movl command to the register eax which is a general purpose register in the Intel architecture
2. jump to the address pointed by the register eax .

So while executing the code, when the kernel comes across these two instructions, the execution at the present memory location is stopped and the execution of the code at the address stored in the eax register is started.

How do we know that the hex translations of the instructions we are using is correct? More importantly, how to obtain such information from the resources made available to us by the Linux.

We do a small experiment wherein we write a small C program with inline assembly code The assembly instructions used are random, as the program does not do anything by itself, but the program contains two assembly instructions we need to understand.

```
/* test.c file */
#include <asm/page.h> /* to include assembly code */

main(){
 asm ("movl $0,%eax\n\t"
                "inc %eax\n\t"
                "nop \n\t"
                "dec %eax\n\t"
                "jmp *%eax");
}
```

We have given \n\t after every assembly instruction so that when we see the output of compiling the above program, each instruction is printed in a separate line, suitably

formatted by the tab.

Now we compile this C program using the *gcc -S tcst.c*

A file name **test.s** is produced which contains assembly instructions of the code above.

The file **test.s** is as follows:

```
file            "test.c"
                .text
.globl main
                .type       main,@function
main:
                pushl       %ebp
                movl        %esp, %ebp
                subl        $8, %esp
                andl        $-16, %esp
                movl        $0, %eax
                subl        %eax, %esp
#APP
                movl $0,%eax
                inc %eax
                nop
                dec %eax
                jmp *%eax
#NO_APP
                leave
                ret
.Lfe1:
                .size       main,.Lfe1-main
                .ident      "GCC: (GNU) 3.2.2 20030222 (Red Hat Linux 3.2.2-5)"
```

We can see that the **test.s** does contain assembly instructions of our interest namely movl $0 %eax and jmp*eax.

Now we compile the same file **test.c** with *gcc -c test.c* to produce **test.o**.

To reconstruct the assembly code from an object code , we can use “objdump” which is freely available on the Linux platforms. Only the text portion of the object file, which is in a ELF format in the Linux, is being expanded using the following command

objdump -ld test.o;

The output produced is :

test.o: file format elf32-i386

Disassembly of section .text:

```
00000000 <main>:
main():
  0:        55                      push   %ebp
  1:        89 e5                   mov    %esp,%ebp
  3:        83 ec 08                sub    $0x8,%esp
  6:        83 e4 f0                and    $0xfffffff0,%esp
  9:        b8 00 00 00 00          mov    $0x0,%eax
  e:        29 c4                   sub    %eax,%esp
 10:        b8 00 00 00 00          mov    $0x0,%eax
 15:        40                      inc    %eax
 16:        90                      nop
 17:        48                      dec    %eax
 18:        ff e0                   jmp    *%eax
 1a:        c9                      leave
 1b:        c3                      ret
```

We can see that the line 9 and the line 18 are of our interest. They tell us the hex equivalent of mov and jmp command we will be using in our project.

Now we are certain that we have the right instructions. Line 9 tells us that

mov $0x0, %eax is represented by b8 00 00 00 00

Line 18 tells us that

jmp *%eax is represented by ff e0.

Thus our following instruction makes sense.(\x is added to tell the compiler that we are using hexadecimal numbers).

static unsigned char pr_jump[NUM_BYTES]="\xb8\x00\x00\x00\x00 \xff\xe0"

Back to our demo.c.

Our next line of the code is

static unsigned char pr_save[NUM_BYTES];

Here we create another variable to be able to store the content of pr_jump when needed.

static int (*pr)(struct sk_buff *, struct net_device *, struct packet_type *);

In the above line we create a function pointer which has two characteristics:

1. The return type is same as that of the return type of the function ip_rcv().

2. The parameters are same as that of the parameters of the function ip_rcv().

6.3.2 Function: init_module()

init_module() should always be of type int and should return 0 on the successful installation of a module.

int slock_flags;

Spin locks are the mechanism by which we can control the undesirable effects of concurrency on the critical sections of a code. When we execute a critical section in a code, we hold a lock or exclusive right to execute the part of the code and no other process on the same or different processor can execute the code so long as we hold the spin lock. Once we are done with the critical section, we release the lock. The next process, which is waiting for the lock, can hold the lock and enter the execution of the critical section.

We also disable the interrupts while we are holding the lock. If we do not disable the interrupts, imagine the situation that our code is preempted after we have taken the spin lock by an interrupt. The interrupt handler also wants to execute the same code. Since we are holding the spin lock (excusive right to the section of the code) the interrupt handler

can not execute the code and we do not have the processor to execute the code. The result is a deadlock. Thus, it is always advisable to disable the interrupts while holding the spin lock.

int slock_flag; defines the spin lock variable. When we do the memory copy in our modules, we hold a spin lock and disable the interrupt.

The syntax of using a spin lock is as follows:

```
static spinlock_t kern_lock = SPIN_LOCK_UNLOCKED; // defination of a  spin lock
                                                     variable

#define LOCK_KERN spin_lock_irqsave(&kern_lock,slock_flags)
#define UNLOCK_KERN spin_unlock_irqrestore(&kern_lock,slock_flags)
```

The function spin_lock_irqsave(&kern_lock,slock_flags) disables the interrupt and takes the lock.

The function spin_unlock_irqrestore(&kern_lock,slock_flags) re enables the interrupt and releases the lock.

pr=(int (*)(struct sk_buff *, struct net_device *, struct packet_type*))allFunAddr[0];

pr is a pointer which can store memory address. Our allfunAddr[0] has the address of the function ip_rcv passed via the command line parameter. We assign this address to the pr after proper pointer casting i.e. same return type as ip_rcv and same parameters as ip_rcv. The effect of the assignment is that we have a pointer to a function pointing to address 0xc020a020.

****(unsigned int *)(pr_jump+1)=(unsigned int)changed_ip_rcv;***

We have declared a function changed_ip_rcv, which has the same return type as ip_rcv and takes the same parameters as the function ip_rcv. We know from our programming

knowledge that name of a function is a pointer to the function. We store this address in pr_jump. If we remember pr_jump is an array of 7 and as of now each byte of the array contains in hexadecimal notation the code “mov,0, 0, 0, 0, jmp, eax”

Thus the bytes 1,2,3 and 4 which contains 0 (array index starts from 0) are changed to the address of the function changed_ip_rcv using the above assignment.

Now pr_jump variable contains the instructions which says

1. move the address of the function changed_ip_rcv to register the eax
2. jump the execution to the address pointed by eax and start executing from this address.

LOCK_KERN;

we take the spin lock and disable the interrupts.

_memcpy(pr_save,pr,NUM_BYTES);

_memcpy is a function defined in our header file “module_header.h”. We could have used the memcpy of C. To be absolutely sure of what we are doing, we write our own memcpy function and name it _memcpy. The code of the function _memcpy can be seen in the file module_header.h.

The function has the structure: _memcpy(*dest, *src, number_of_bytes)

This function copies num_of_bytes from the memory location pointed to by the src pointer to the memory location pointed to by the dest pointer.

We know in our program above, pr points to the memory address 0xc020a020. _memcpy copies 7 bytes from this memory location and stores it in a variable pr_save which is a pointer to an array. Since we are accessing the kernel memory directly, we do not want to be disturbed and we also take a spin lock before this operaton.

_memcpy(pr,pr_jump,NUM_BYTES);

This piece of the code copies the "move" and "jump instructions from the address pointed by pr_jump to the address pointed by pr. The net result is "move" and "jump" instruction being stored at the address pointed by pr.

UNLOCK_KERN;

we release the spin lock.

The result of the assignments is that we have stored a move and jump instruction at the address pointed by ip_rcv after storing the original 7 bytes of that address in the variable pr_save.

When the kernel needs to execute the ip_rcv function, the fist thing it will encounter at the address of ip_rcv is our move and jump instruction, pointing to the changed_ip_rcv function. The kernel will act accordingly i.e. start executing changed_ip_rcv. We know that the changed_ip_rcv has the same return type and the functions parameters as that of the ip_rcv, it will conveniently receive the parameters originally intended for ip_rcv as if it was the original function.

We finish our init_module with a return of 0 which will signal the successful installation of a module to the kernel.

6.3.3 Function: changed_ip_rcv()

my_time(store_time);

We record the time of our arrival in this function using the function my_time declared and defined in the header file module_header.h.

The value is stored in the variable store_time.

We do not write directly to the disk file every time we see a packet. One of the reasons is speed. Access to the memory is faster than the disk access, thus saves us the valuable packet processing time.(If the kernel takes too much time to process a packet, it will eventually start dropping the packets as it will not be able to meet the speed of packet reception/generation). To solve this problem we have declared a big buffer memory variable called big_buffer[BUF_SIZE] where we store all we need to record. There is another reason for this behavior from the programming perspective. While handling of the incoming packets, we are executing in the bottom half context of the interrupt. In the interrupt context of execution, we cannot store our information directly to the disk files. The reason for such a limitation is because, the file accessing routines can block while waiting for the file input/output to be over. Blocking while executing in the interrupt context is not allowed. The method adopted is we store everything we need to record in the big_buffer variable and then while unloading the module we write to the disk.

```
strcat(big_buffer,store_time);
  strcat(big_buffer," ");
  strcat(big_buffer,"ip_rcv");
  strcat(big_buffer," ");
  strcat(big_buffer , in_ntoa(skb->nh.iph->saddr));
  strcat(big_buffer," ");
```

We stored the time we entered the ip_rcv function to the accuracy of micro seconds.

As we normally do while instrumenting functions, we store the name of the function.

Also as a demonstration of what we can do with the parameters inherited, we store the source address of the packet for which ip_rcv has been called.

```
LOCK_KERN;
      _memcpy(pr, pr_save,NUM_BYTES);
UNLOCK_KERN;
```

We need the networking functions of the kernel to continue the processing of the packet as we are not here to stop the packet processing and crash the kernel. We have to find a way to let the ip_rcv do what it is normally supposed to do with a packet. We reverse the changes we have done to the kernel by the above piece of code. We know that the the first seven bytes of the ip_rcv function is stored in the pr_save. We copy the original bytes to the address pointed by pr, which is the original address of the function ip_rcv .

retval=pr(skb,dev,pt);

We now execute the original ip_rcv by using the function pointer pr which points to this function and collect the return value in a variable retval which is of the same type as that of the return value of the function ip_rcv.

LOCK_KERN;
_memcpy(pr, pr_jump,NUM_BYTES);
UNLOCK_KERN;

After this we gear up for the next packet and again bring about changes so that changed_ip_rcv will be executed for the next packet instead of ip_rcv.

We can again do the book keeping of storing the time i.e. the time we finished processing of ip_rcv kernel function. We can also store the current values of the other variables if we desire.

return retval;

Finally, we return the retval the value ip_rcv was supposed to return so that the next function in the kernel can continue the processing of the packet.

6.3.4 Function:cleanup_module()

```
void cleanup_module(){

  _memcpy(pr, pr_save,NUM_BYTES);
    .....
}
```

In the cleanup module, we reverse the changes brought about by the init module ie. store the original 7 bytes of the ip_rcv in the address pointed by the pr pointer.

```
if(strlen(big_buffer)>0){
      print_buffer(FILE_NAME,big_buffer,strlen(big_buffer));
      big_buffer[0]='\0';
```

In case we have recorded anything in the variable big_buffer, we print it a the disk file /tmp/packet .log which is defined by the FILE_NAME in our header file .

For further details on kernel function hijacking see Kernel Function Hijacking [4].

We have many utility functions defined in the header file which are used to format the data, record the time and print the big_buffer on to a disk file. The functions are mostly self explanatory so we shall explain only few of them in the following section of the report.

6.3.5 Time in a Linux kernel

A timer interrupt is the way used by the kernel for keeping track of time. Interrupts are fired by the external hardware and when this happens, the CPU is interrupted in its current activity and it executes a special code called Interrupt Service Routine (ISR) to serve the interrupt. A timer interrupt is generated by the systems timing hardware at a regular interval and this interval is set by the value of a HZ variable defined in the file <linux/param.h>.This value is architecture dependent. In most of the i386 architecture, it

is set to 100 which gives a resolution of approximately 10 milliseconds.

Jiffies is the variable which records number of timer interrupt occurred since the system startup. The value of the jiffies is initialized to 0 at the system boot time .It is declared in <linux/sched.h>. jiffies is *unsigned long volatile* type and will possibly overflow in case of 16 months of continuous system operation.

We have an option of tracking our time using the jiffies, but the resolution offered is inadequate as we will see later in this project that the networking functions are traversed by a packet in few microseconds.

We could also not use time_of_the_day() functions as this uses another variable xtime which is initialized by the kernel at the booting time by reading the RTC.(Real Time Clock) . RTC keeps track of the time when a system is powered down by use of a small battery on the motherboard .The resolution offered by xtime is inadequate for our purpose and it is updated less frequently than we would need it in our measurements. xtime is declared in the file /include/linux/sched.h

Architecture Specific methods

When the time is to be measured minutely we could resort to architecture Specific methods. Most of the architectures have a time counter register called TSC (timestamp counter) and they can be read from both the user and the kernel space. This register increments its value after every CPU clock cycles and the following macro can be used to read the register.(#include <asm/msr.h>)

rdtsc(low,high) // catches 64 bit value in two 32 bit variables
rdtscl(low) // Catches lower 32 bit value

The 64 bit registers are not present in all the architectures. If we catch only lower 32 bit value for a 500 MHZ machine the 32 bit counter will overflow in every 8.5 seconds . As our networking measurements may last more than this interval, the overflow of register may lead to a readings which can be a potential pitfall while calculating the various timings. These register are architecture specific and their use may impair code portability to other architectures. Thus, we sacrifice the accuracy of the measurements to the level of system clock and resort to gettimeofday() function which gives us the accuracy of microseconds.

The function my_time (char *p_time) in our header file takes a pointer to a string where it will store its result. The most notable line of the function is

do_gettimeofday(&tv);

do_gettimeoftheday is function available in kernel mode which takes as its argument a variable of type struct timeval defined in time.h of following type

```
struct timeval {
        time_t          tv_sec;         /* seconds */
        suseconds_t     tv_usec;        /* microseconds */
};
```

The function returns with tv_sec containing seconds elasped since epoch (January 1, 1970 UTC) and tv_usec containing the microseconds.

Further formatting is required to convert this information into the present time in Hour,Minute,Seconds and Microseconds after taking into account the time zone we are presently in. my_time() function does this formatting and returns the present time .

6.3.6 Printing to a Disk File from the Kernel Module

Normally when we use a kernel module, the proc file system is used for recording the output. We do not take this approach. Our approach is that we store the information in a memory buffer large enough to store all the information we will generate and then print the buffer directly from inside the kernel to a disk file /tmp/packet.log. The function which does this job is print_buffer(). We have defined this function in the header file module_header.h. The procedure is conceptually similar to opening, writing and closing the file as we do from a user space program but the interface functions used are slightly different since we do this in a kernel mode of execution. The code is self- explanatory. All the log files produced are enclosed in the Appendix section of this report.

6.3.7 Macro Definations in the module_header.h

In the file module_header.h, we have the following macro definitions

MODULE_AUTHOR("Gyan"); //Who wrote the code ?

MODULE_DESCRIPTION("Tcp Ip in Linux Kernel"); //what does module do

MODULE_LICENSE("GPL"); //License

In case we do not define the MODULE_LICENCE macro,the kernel will issue a warning that the module will taint the kernel . To keep away from the warning declare the module code under GPL.

6.3.8 Other Support Functions

As stated earlier, we have created our own include file called “module_header.h” for this project, which contains all the necessary header files and support utility functions. We shall not discuss them as they are self-explanatory and can be seen in the source code of this report.

The whole project is modularized with different kernel modules for different layers of the TCP/IP protocol. The transport layer protocols considered are TCP, UDP and the network Layer protocol considered is IP. We start our discussions with the TCP layer.

7. 0 TCP Implementation in Linux

We have written four LKMs which change the behavior of the TCP functions of Linux. All the four LKMs are in four different files and the files are:

a) **tcpout.c** – Contains modifications to the kernel functions mostly involved in the handling of the outgoing data packets. The functions are :

1. tcp_sendmsg
2. tcp_push
3. __tcp_push_pending_frames
4. tcp_write_xmit
5. tcp_transmit_skb
6. tcp_retransmit_skb

b) **tcpin.c** Contains modifications to the functions involved in the handling of the incoming data packets.The functions are :

1. tcp_v4_rcv
2. __tcp_v4_lookup
3. tcp_v4_do_rcv
4. tcp_rcv_established
5. tcp_ack
6. tcp_event_data_recv
7. __tcp_data_snd_check
8. __tcp_ack_snd_check
9. tcp_data_queue

c) **tcp_prot.c** - Contains modifications to the functions involved in the window management of the TCP and other protocol related book keepings. The kernel functions are :

1. __tcp_select_window
2. tcp_receive_window
3. tcp_cong_avoid
4. tcp_enter_loss
5. tcp_recalc_ssthresh

d) **synfin.c** - Contains modifications of function involved in handling connection establishment and connection closing aspect of TCP. List of functions are:

1. tcp_v4_init_sock
2. tcp_setsockopt
3. tcp_connect
4. tcp_v4_connect
5. tcp_rcv_synsent_state_process
6. tcp_send_ack
7. tcp_rcv_state_process
8. tcp_create_openreq_child
9. tcp_v4_conn_request
10. tcp_v4_send_synack
11. tcp_close
12. tcp_send_fin
13. tcp_close_state
14. tcp_fin

15. tcp_send_delayed_ack

16. tcp_time_wait

17. tcp_set_state

Any combination of the above four modules can be loaded and studied, giving us the advantage of modularity in design.

Certain functions in the TCP are ***static inline*** defined in the files tcp.h, tcp.c & tcp_input.c which could not be instrumented without kernel recompiling. The code of inline functions when compiled with compiler optimization on, is attached to the calling function and thus there is no memory address where we could attach our jump code. We recompiled the kernel after moving the functions from tcp.h to tcp.c and removing static __inline__ directive before all the functions of our interest. This resulted in the address of the function becoming visible in the System.map and the functions themselves available for instrumentation.

We shall discuss the log file produced after loading all the four modules and running our usual procedure of sending a file using ftp from a source machine A to a destination machine B.

All the four modules write to the same big_buffer as they see the packets and when we unload the modules, we get a dump of the big_buffer to a disk file /tmp/packet.log, using our print_buffer function.

This log file is then run through a perl program to format and suitably present the file.

In order to understand the log file, we must understand what these kernel functions do and how they are called one after the other. A brief discussion on the various transport layer kernel functions is being presented.

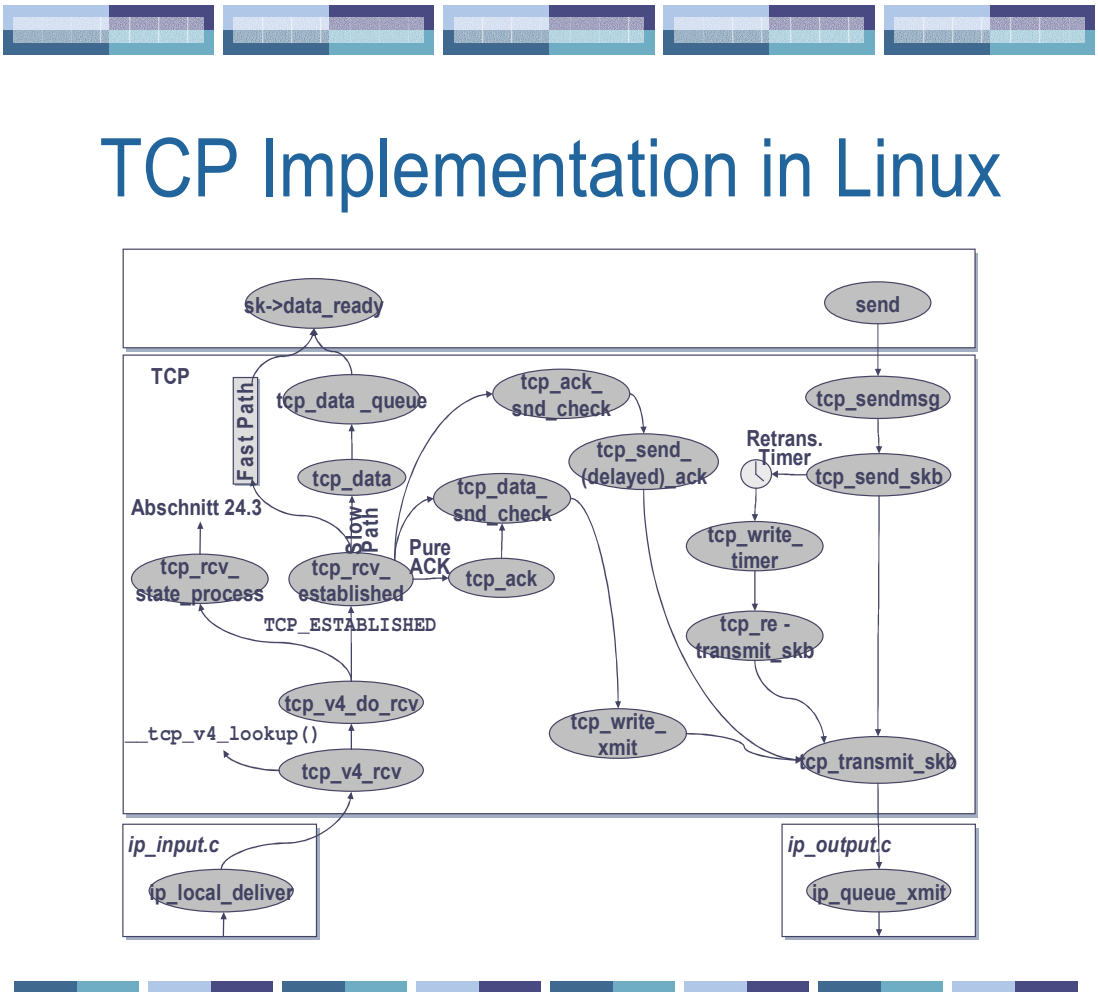


Figure – 6: TCP Implementation In Linux
(source: University of Illinois, Dept of CS)

7.1 Handling an incoming Segment

All important functions used in the TCP for handling an incoming segment are instrumented in file tcpin.c. Only data flow is considered for simplicity.

To begin with, how does IP know what is the transport layer function it should hand over the further processing of a packet? All the transport protocols are managed in a hash table

inet_protos. This hash table is built at the time of initialization by the function inet_proto_init() using the function inet_add_protocol(). In fact we can write a module and register our own transport layer protocol using inet_add_protocol() and de-register the protocol using inet_del_protocol().

The data structure which is attached to the inet_protos[MAX_INET_PROTOS] is prot structue of type inet_protocol. Refer to the figure – 7:

1. handler() is a function pointer to the entry function of the transport protocol which is in case of the tcp is tcp_v4_rcv and in case of the UDP is udp_rcv.
2. id is the protocol id . If an IP packet with this identifier in protocol field of the IP header is received, then it is passed to the handling routine .If there are more than one protocols with the same id registered, then a copy of the skb is passed to each of the protocols.
3. The copy bit specifies if there are more than one protocol with the same id

The ip_local_deliver_finish() calls ip_run_ipprot() which runs though the the hash table and returns 1 if it successfully finds the transport protocol. It hands over a copy of the skb to the handler function using the following line of code ipprot->handler(skb2);

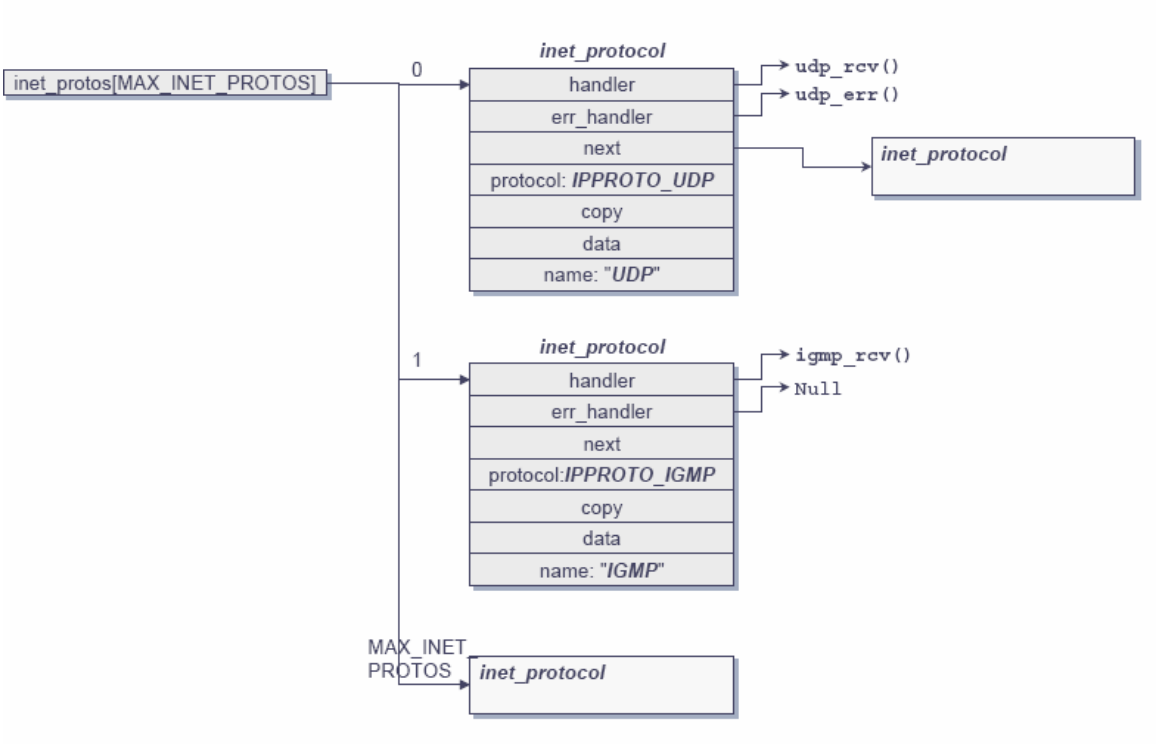


Figure : 7 Transport Protocols in a Hash
(source: University of Illinois, Dept of CS)

Thus the ip_local_deliver_finish hands over the packet to tcp_v4_rcv in case of TCP. Data flow in the TCP is as follows. Indenting depicts the function call for clarity purpose and B,E stands for before entry and after exit of a kernel function, respectively.

tcp_v4_rcv B
  __tcp_v4_lookup B
  __tcp_v4_lookup E
tcp_v4_rcv E

tcp_v4_rcv

does some sanity checking and removes the ip header from the data .Then normally adds the packet to a backlog queue using the function sk_add_backlog(sk, skb) once the corresponding sock structure of the packet has been found.

tcp_v4_lookup

Searches the hash table for an active socket or presence of sock structure corresponding to the present packet.

tcp_v4_do_rcv

Removes the packet from the backlog queue and checks for sk->state?

In case the state is TCP_ESTABLISHED, then the further processing is taken care by the function tcp_rcv_established() otherwise tcp_rcv_state_process() is called.(this is explained while explaining synfin module)

We continue with *tcp_rcv_established* . Here a packet processing can follow two paths fast path and slow path. We shall not get into details further.

Now a check is done to see if the incoming packet contains only ack or it contains data + ack. The check determines the next function to be called.

If the incoming packet contains only ack, then the following path is followed:

```
tcp_v4_do_rcv B
   tcp_rcv_established B
      tcp_ack B
      tcp_ack E
   tcp_rcv_established E
tcp_v4_do_rcv E
```

The incoming ack is processed in tcp_ack . In fact tcp_ack handles all the tasks of receiving and acknowledging a packet with a valid ACK numbers. The tasks could be changing the receive window, cleaning of retransmission queue, adapting the congestion window etc.

If the incoming packet contains ack + data then the following path is followed:

```
tcp_v4_do_rcv B
   tcp_rcv_established B
      tcp_ack B
      tcp_ack E
      tcp_data_queue B
         tcp_event_data_recv B
         tcp_event_data_recv E
      tcp_data_queue E
      __tcp_ack_snd_check B
      __tcp_ack_snd_check E
   tcp_rcv_established E
tcp_v4_do_rcv E
```

tcp_data_queue() processes the payload and inserts the skb into the receive queue of the socket.

tcp_event_data_recv() handles all the management work required for receiving the payload like initializing the delayed ack engine, increasing the value of slow start thresh hold variable in the tcp_opt data structure etc.

7.2 Handling an outgoing Segment

TCP uses a send system call at the socket level to send the payload which causes the tcp_sendmsg function to be invoked. This function is present as an handling routine in the tcp_prot structure.

A typical data transmission path followed by the tcp segment is

```
tcp_sendmsg B
   tcp_push B
      __tcp_push_pending_frames B
         tcp_write_xmit B
            tcp_transmit_skb B
            tcp_transmit_skb E
         tcp_write_xmit E
      __tcp_push_pending_frames E
   tcp_push E
tcp_sendmsg E
```

The **tcp_sendmsg()** copies the payload from the user address to the kernel address space.

tcp_push pushes the packet with push or urgent flag set.

tcp_push_pending_frames checks if there are segments for transmissions and if there are, then passes the control to tcp_write_xmit.

tcp_write_xmit() continues to send the frames as long as it is allowed by the function **tcp_snd_test()**. It also cheeks whether the condition of the tcp algorithms are maintained e.g. slow start, congestion-control etc.

tcp_transmit_skb is responsible for completing the TCP segment and passing it to the function pointed by tp->af_specific->queue_xmit and in case of IP it is ip_queue_xmit().

7.3 Handling of Connection Management

a) tcp_rcv_state_process()

This function handles the TCP/IP state transitions and management works for a connection.

b) tcp_v4_init_sock()

This function runs the various initialization functions like initialization of queues, timers, slow start, maximum segment size .

c) tcp_setsockopt()

This function sets the customizable options of TCP like TCP_MAXSEG, TCP_NODELAY.

d) tcp_connect()

This function initializes the outgoing connection. It reserves the memory for a connection and initializes the sliding window variables.

e) Transition from CLOSED to SYN_SENT

Transition from CLOSED to SYN_SENT state happens when an application calls connect at the socket interface. This invokes the function tcp_v4_connect() which in turn calls the tcp_connect(), which changes the state from CLOSED to SYN_SENT, using the tcp_set_state() function. This can be seen in the following lines of the log.

tcp_v4_connect B

tcp_time_wait B

tcp_time_wait E

tcp_connect B

tcp_connect E

tcp_v4_connect E

f) Transition from SYN_SENT to ESTABLISHED state

After receiving a SYN and an ACK, the client tcp sends an ack to the server and changes from SYN_SENT state to the ESTABLISHED state. The following lines of the log file explain this event.

tcp_send_ack B

tcp_rcv_synsent_state_process B

tcp_time_wait B

tcp_time_wait E

tcp_send_ack B

tcp_send_ack E

tcp_rcv_synsent_state_process E

tcp_send_ack E

g) Connection Tearing Down process

When both computers client A and server B is in the established state and the computer A initiates the connection closing by sending a packet with a FIN flag set, TCP state of the computer A which was in the established state switches to the FIN_WAIT_1 state. The function tcp_close_state() switches TCP from the Established to the FIN_WAIT_1 state.

tcp_close B

tcp_close_state B

tcp_set_state B

tcp_set_state E

tcp_close_state E

tcp_send_fin B

tcp_send_fin E

tcp_close E

tcp_set_state is used by the tcp to transition to a desired state.

tcp_send_fin is used by the tcp when it wants to send a FIN

h) Transition from FIN_WAIT_1 To FIN_WAIT_2 state

As soon as the computer A receives an ACK from the computer B without a FIN, it changes the state from FIN_WAIT_1 To FIN_WAIT_2 using tcp_set_state().

i) Transition from FIN_WAIT_2 to TIME_WAIT

As soon as TCP in the computer A receives a FIN, it sends an ack and changes the state from FIN_WAIT_2 to the TIME_WAIT using the function tcp_time_wait().

tcp_send_ack B

tcp_send_ack E

tcp_rcv_state_process B

  tcp_set_state B

  tcp_set_state E

  tcp_time_wait B

    tcp_set_state B

    tcp_set_state E

  tcp_time_wait E

tcp_rcv_state_process E

7.4 Handling of the Protocol Data

Important functions handling the protocol related data are:

a) tcp_select_window()

This function is invoked in tcp_transmit_skb() when a tcp segment is sent (except syn and syn-ack) to specify the size of the advertised window .

b) **tcp_receive_window()**

This function specifies the current advertised window and **__tcp_select_window()** is used to see how much buffer space is available which forms the basis of a new transmit credit to be offered to the partner TCP instance.

c) tcp_cong_avoid()

This function implements a congestion window growth in the slow-start and congestion-avoidance algorithms. tcp_cong_avoid() is invoked when an incoming TCP segment with a valid ACK is handled in the tcp_ack().

d) tcp_enter_loss()

This function is invoked in the handling of a retransmission timer. If the timer goes off and the data has not been acknowledged yet by the partner TCP instance, it is treated as one of the symptoms of a congestion and **tcp_recalc_ssthresh()** is used to recalculate the new value of Slow Start Threshold variable.

7.5 Discussion on the LOG File

We shall discuss all the TCP functions in action for transmission of a small file from the source computer gyan.home having an ip 192.168.1.20 to the destination computer afs1.njit.edu ,which is an alias for alizarin.njit.edu having an ip address 128.235.204.81, using the FTP protocol. We also start the tcpdump in parallel, capturing the packets so that we can co-relate the activities as seen by the tcpdump and the log produced by intercepting the kernel TCP functions, using our tool.

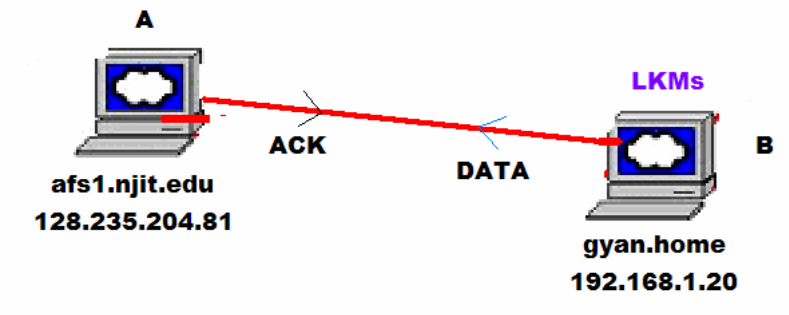


Figure - 8: Topology for capturing of TCP Data

The log file produced by our tool is LOG_TCP and the corresponding tcpdump file is the DUMP_TCP. Both the files are there in Appendix section of this report.

The format of the file LOG_TCP has already been explained but is being presented once again to refresh ourselves.

1st column Line number
2nd column Time
3nd column source ip:source port

4th column destination ip : destination port
5th column name of the function
6th column position where we see the packet
B stands for the beginning of the function
E stands for end of the function

Additional columns if present are

7th column S/P/F/.
S in case SYN flag is set
P in case PUSH flag is set
F in case FIN flag is set
. in case none of the above
8th column TCP source starting sequence number

9th column ack word

10th Column TCP ack sequence Number

All the columns are white space separated and every record of the log file is suitably indented to make visible the entry and exit of a function , by running the log file through a cleaning perl program. After exit from a block of functions, total time taken for the block in microseconds is printed.

DUMP_TCP file has also been modified to include line numbers so that the lines can be easily referenced in our discussions.

In the discussion that follows we are discussing LOG_TCP file in the appendix thus the line numbers always refers to the LOG_TCP file and whenever we reference the DUMP_TCP's line numbers, we shall specifically mention it.

line 1 and 2 depicts the socket initialization process.

Lines 3 to 10

tcp_v4_connect() changes the state from CLOSED to SYN_SENT and an SKB is transmitted with SYN flag set and a TCP Sequence Number *516203531*

This can be verified from the Line 7 of the DUMP_TCP.

Lines 11 to 32

We can see from Line 8 of the DUMP_TCP, afs1.njit.edu sends a SYN and ACK to the gyan.home computer.

In ***Line 11*** we see that the gyan.home receives a packet. In tcp_v4_rcv. SYN flag is set and the SYN number matches with SYN number seen in the DUMP_TCP file.

The socket of this connection is searched by using the __tcp_v4_lookup . Here we see that the tcp_v4_do_rcv() is called from within tcp_v4_rcv() so as to speed up the processing of a packet. The protocol implementation knows that this is the 1st packet for the connection, thus no point in queuing the packet in the incoming queue for later retrieval by tcp_v4_do_rcv(). This is an excellent example of optimization in the TCP/IP stack of Linux. tcp_v4_do_rcv() continues the processing of the packet by calling tcp_rcv_state_process() as TCP is not in the CONNECTED_STATE. tcp_rcv_state () process calls tcp_rcv_synsent_state_process() as gyan.home is in the SYN_SENT state. As the incoming packet also contains acknowledgment, all the processing related to acknowledgment is handled in the tcp_ack() function.

gyan.home has received ack so the state must change from SYN_SENT to the CONNECTED state. This state change is brought about by the function tcp_set_state(). Processing of an incoming segment by TCP is complete at this stage.

As gyan.home has received a syn and ack it must send an ack. The process of sending an ack is initiated by calling tcp_send_ack(). This function calls tcp_transmit_skb() which does the task of completing the segment. Before sending the segment tcp_transmit_skb() must know what is the present window size. This is determined by calling the

tcp_receive_window(), which returns current advertised window size of 5840 bytes. The function also checks the present status of receive buffers by calling __tcp_select_window() to ascertain what window size it must advertise to the partner tcp. Armed with these information a segment is completed and transmitted with appropriate window size of 5840 and ACK sequence Number of 2291929752. We can see on line number 27 as well as Line number 9 of DUMP_TCP.

Line 35 to 62 of LOG_TCP

These lines corresponds to the line 10 of DUMP_TCP where alizarin.njit.edu sends 64 bytes of data and ACK to gyan.home.

In Line 35 to 38 we see that tcp_v4_rcv() receives the data, finds the corresponding socket and queues the data for a later processing by the tcp_v4_do_rcv().

In ***line 39*** we see that tcp_v4_do_rcv() starts the processing of incoming queues and finds this packet. The rest of the processing is same as in the previous case till the Line 45, where the TCP discovers that the data is received and it queues the data using tcp_data_queue() and the receiving path followed is that of a packet having both data and ACK. In line 48 we see that the tcp_event_data_recv() is called.

Now TCP checks if it can send any ACK using tcp_ack_snd_check(). As data has been received, ack could be sent and ack sequence number 2291929816 is sent acknowledging all the incoming data. Line 58 of the LOG_TCP tells us this.

Line 67 to 80

Line 12 of DUMP_TCP shows the transmission of data from gyan.home to alizarin.njit.edu and corresponds to the above lines. We can see that the Push flag is set so

the tcp_push() is called by tcp_sendmsg(). tcp_push() calls tcp_push_pending_frames() which in turn calls tcp_write_xmit() which determines with the help of tcp_send_test(), how much data could be sent. This function is not invoked if only ACK is sent by the sending TCP as sending only ACK does not involve any change of transmission window in the sliding window mechanism.

Line 271 to 278

We can see that the data flow of TCP starts here. gyan.home sends a SYN to alizarin.njit.edu. Again the similar steps as that of the above are repeated for this connection too, till we reach line 389-390 where we see that the congestion window is increased from 2 to 3 by calling tcp_cong_avoid().

Line 371 to 380

We see that the tcp_close() function is executed and a fin is sent. This function even though recorded here is actually executed only after Line 425 . I fell that because of the network processing load, out of sequence recording takes place.This requires further investigation.

After sending the FIN, TCP of gyan.home switches to FIN_WAIT1 state.

Line 431 to 450

We see that after receiving an ack from alizarin.njit.edu, gyan.home TCP changes the state from FIN_WAIT_1 to FIN_WAIT_2.

Line 451 to 454

We see that gyan.home receives a FIN from alizarin.njit.edu and moves from the FIN_WAIT_2 to the TIME_WAIT state.

8.0 IP Implementation in Linux

8.1 General Methodology

The Internet Protocol (IP) is the only network layer protocol we consider in our discussions. The main contribution of the IP is to facilitate routing between the two communicating computers. It is an unreliable protocol like UDP.

The packets handled by IP can be generated by, or destined to, any transport layer protocol. In addition, when the Linux box is working as a router, the packets are received and routed by IP. IP may itself generate a packets, an example would be fragmentation of large packets or ICMP packets.

In our experimentation, we transfer a file using the FTP protocol from my home computer having IP address 192.168.1.20 to afs1.njit.edu and observe the handling of the packets by various functions of IP. The measurement software is loaded on the home computer.

The instrumentation is conducted by separating the functions of IP involved in handling of the incoming packets, the outgoing packets and forwarding of the packets. The corresponding files are myip_rcv.c, myip_send.c, myip_forward.c.

8.2 Our Own Network Layer Protocol

Packets arrive on an interface of a computer and are stored in the input queue of the respective CPU. Packets are handled by the device layer and then passed on to the suitable network layer. Without getting into much detail, we shall look at handling of the packets by the device layer. A Device driver mainly relies on two kernel functions:

1. dev_allock_skb() : This function allocates a sk_buff of appropriate size prior to the transfer of packets from the device memory i.e. NIC buffers to the kernel memory using DMA(Direct Memory Allocation).

2. netif_rx() : This function is used to pass the sk_buff to the generic device layer, as soon as the packet reception is completed at the device interface. It typically runs in the context of a hardware interrupt that signaled the completion of DMA transfer. It queues the sk_buff for further processing and schedules a bottom half for further processing. Bulk of the processing is done in the context of the "bottom half" by net_rx_action().

The "Interrupt Handling Mechanism" for the interrupt generated by the Network Interface Card(NIC) on reception of a packet is split into two, top half and a bottom half. Absolutely minimal necessary activities are performed in the top half with interrupts disabled. Once the top half completes, the bottom half is scheduled to run with the interrupts enabled. This ensures that the kernel does not loose the incoming packets, as it is able to handle the interrupt generated by the NIC card on the reception of the packets, while executing the bottom half.

Protocol handlers register themselves by filling a struct packet_type and passing it to the function dev_add_pack() where the structure is put in a chain to facilitate an easy retrieval.

```
struct packet_type

{
   unsigned short   type;   /* This is really htons(ether_type).    */
   struct net_device *dev;   /* NULL is wildcarded here          */
   int (*func) (struct sk_buff *, struct net_device *,
        struct packet_type *);
   void        *data;  /* Private to the packet type          */
    struct packet_type      *next;
};
```

The first member of the above structure is "type" and in the case of IP it is the protocol present in the Ethernet header namely ETH_P_IP. When the scheduler calls the net_rx_action(), it passes the packet to the protocol function pointed by the "func" member of the structure. In the case of IP, it is ip_rcv().

Protocols, which wish to receive all the incoming packets, are linked in a list pointed by the ptype_all pointer. They register themselves with the type as ETH_P_ALL and they are processed first before considering the protocols that consume only a specific packet type. We have added our own protocol to demonstrate the concept and a brief description is being presented.

In the init_module() of the code of the file ip_rcv.c, which is our module file containing LKM , we have added our protocol with the following line

dev_add_pack(&my_ip_protocol);

"my_ip_protocol" is global variable of type struct packet_type. With the declaration

packet_type my_ip_protocol;

Thus, we create a variable "my_ip_protocol" of the above type and give appropriate values to the members of the structure as follows:

```
static struct packet_type my_ip_protocol = {
__constant_htons(ETH_P_ALL),
NULL,
my_pack_rcv,
(void *) 1,
NULL
};
```

Our handler function is called my_pack_rcv(), which is given a copy of skb for all the incoming packets. (The corresponding function in the kernel code is ip_rcv() to receive all packets of type ETH_P_IP.)

We will see in the log files that our protocol is always handed the packet first and then the ip_rcv() receives the packet since we have registered out protocol with the type as ETH_P_ALL.

The interface between the device layer and the network layer once again demonstrates the modularity of the TCP/IP implementation in the Linux kernel.

8.3 Netfilter Hooks

These hooks are places in the kernel code and can register a functions to be called at a specific event say "reception of a packet". We will not discuss about netfilter hooks and its functionality in details. The code explains the method of registering the functions with the various hooks . There are five of them in total in ipv4 for handling of the packets in the different transmission/reception stages.

NF_IP_PRE_ROUTING :
In the path of the incoming packets after sanity checks and before routing decisions

NF_IP_LOCAL_IN:
After the routing decisions if the packet is for this host

NF_IP_FORWARD
If the packet is destined for another interface

NF_IP_LOCAL_OUT
For packets coming from local processes on their way out.

NF_IP_POST_ROUTING
Just before outbound packets are transferred to the device layer.

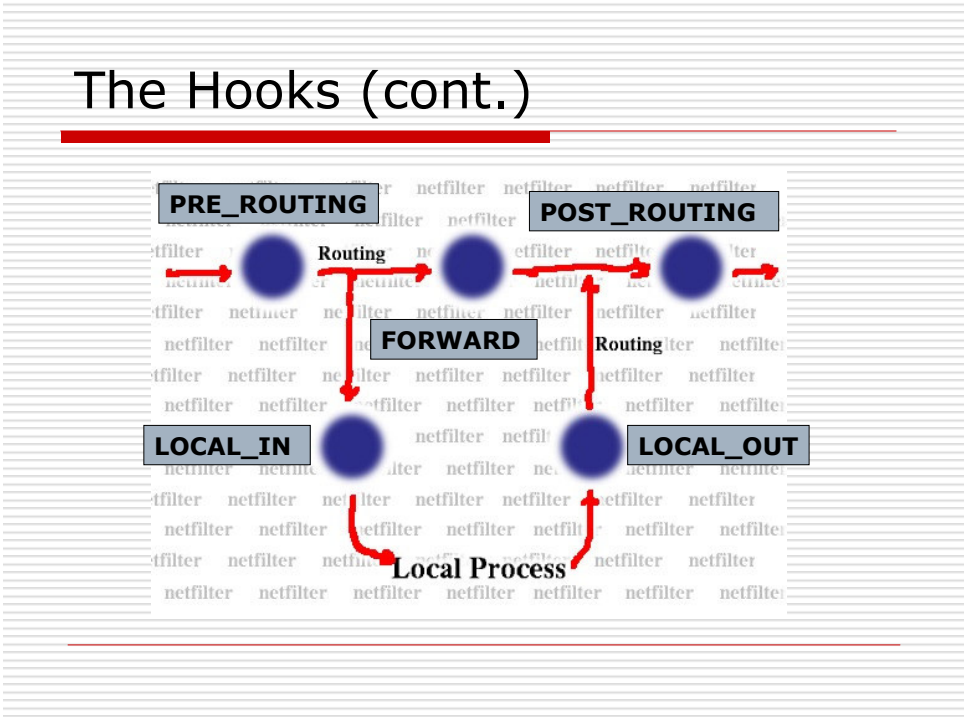


Figure – 9: Netfilter Hooks in the Linux IP
(source: University of Illinois, Dept of CS)

8.4 IP Functions Intercepted

Study of IP implementation in Linux is divided in three parts based on the path followed by a packet. Following are the kernel functions where the packets were intercepted using our tool.

1. In myip_rcv.c file: (The functions that handle an incoming IP packet)

ip_rcv
ip_rcv_finish
ip_route_input
ip_local_deliver
ip_defrag
ip_local_deliver_finish

In addition to the above, as soon as the packets are received by the device layer, our new protocol is handed over the packets. We also see the movement of the packets at the netfileter hooks in the IP layer.

2. In myip_send.c. (The functions that handle outgoing IP packets)

ip_queue_xmit
ip_queue_xmit2
ip_output
ip_finish_output2
ip_finish_output

3. In myip_forward.c. (The functions that handle forwarding of the IP packets)

ip_forward
ip_forward_finish
ip_send

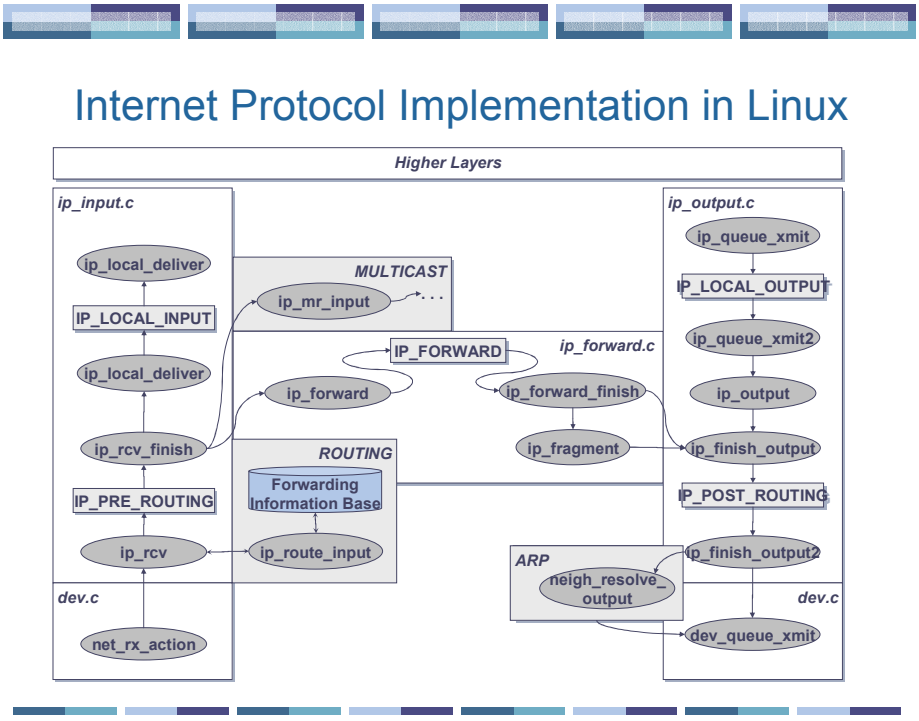


Figure – 10: IP Implementation in Linux
(source: University of Illinois, Dept of CS)

8.5 Handling of Incoming Packets by IP

ip_rcv()

As explained earlier, during the course of receiving an incoming packet, netif_rx_action() hands over control of packet processing to the ip_rcv() function. This function rejects the packet not addressed to the local computer. For example, packets received in promiscuous mode. Then basic correctness checks are performed like

1. Does the packet have minimal size of an acceptable IP packet?
2. Is the IP version 4?
3. Is the checksum correct?
4. Does the packet have a right length?

Once the above checks have been passed by the packet, the netfilter hook NF_IP_PRE_ROUTING is handed over the packet. A netfilter hook is invoked by a macro and function following the hook is passed to this macro. The line of code used to pass the packet to the above hook is

```
return NF_HOOK(PF_INET, NF_IP_PRE_ROUTING, skb, dev, NULL,
               ip_rcv_finish);
```

We can compile a kernel without the netfilter hook support and this macro will ensure that the packet is passed directly to the follow up function. If the kernel is compiled with the netfileter support, the functions registered with the hook is passed the packet and eventually the control will be passed to the followup function.

ip_rcv_finish()

The function **ip_route_input()** is invoked to determine the route of a packet . The skb->dst pointer of the skb is set to an entry in routing cache. Routing details are cached by the kernel to speed up the process of routing. Routing cache is the area where the details of the destination at the IP level are cached. If the ip_route_input() cannot determine the route, the packet is discarded.

Then handling of IP options is done by allocating ip_options structure.

Finally the processing of packet has reached a stage where it must be determined if the packet is destined for the local computer or it must be forwarded. The information for the further path is stored in the skb->dst pointer. Function pointers are used to determine which function handles the packet next . Skb->dst->input() points to the function which will be used to handle the packet next, based on the various possible scenarios.

1. ip_local_deliver() is pointed to, if the packet is for the local computer
2 . ip_forward() is pointed to if the unicast packet is meant to be forwarded.
3. ip_mr_input() is pointed to in case of the multicast packets which need to be forwarded.

In our discussion we do not consider multi cast packets and we are tracing the path of incoming packets to local computer so the next function of our interest is ip_local_deliver().

ip_local_deliver()

The most important task of this function is to reassemble the fragmented packets using **ip_defrag()**. The details of de-fragmentation will not be looked into, in detail. The process mainly consists of collecting the fragments of packets over a period of time until all the fragments of a datagram have arrived so that they can be passed on for further handling as a whole. The netfilter hook NF_IP_LOCAL_IN is passed on the packet next.

ip_local_deliver_finish()

The NF_IP_LOCAL_IN passes the packet to the ip_local_deliver_finish().The packet has reached the end of the network layer processing and now the next course of action is to be determined.

1. Is the packet meant for RAW-IP socket ?
2. Else what is the Transport protocol?

All transport protocols are managed in ip_prot hash table in Linux. The de-multiplexing of the IP layer is explained while explaining the TCP. Please refer to the TCP section for

the further details. The most common handling routines are

1. tcp_v4_rcv()
2. up_rcv()
3. icmp_rcv()
4. igmp_rcv()

In case no transport protocol is found, the packet is passed to a RAW socket(if there is one) or it is dropped and ICMP Destination Unreachable message is returned to the sender.

8.6 Handling of Outgoing Packets by IP

There are many IP functions available for transport layer to send the data created locally like

ip_queue_xmit() , **ip_build_and_send_pkt(), ip_build_xmit().** Each of these functions are specialized and optimized for a specific use.

1. ip_queue_xmit () - used for data packets from tcp
2. ip_build_and_send_packet() - used for SYN or ACK packet that do not contain data.
3. ip_build_xmit () - used for UDP packets containing data.

We shall discuss ip_queue_xmit() which is the one normally used for the data packets.

ip_queue_xmit()

This function first checks whether the socket structure sk->dst includes a pointer to an entry in the routing cache and if so whether the pointer is presently valid. The route of a packet is kept in the socket structure which is referenced by the skb i.e. skb->sk has the

route details as all the packets of a socket go to the same destination. This ensures that the expensive search for the route can be avoided, wherever possible.

If no route is present, then the ip_route_output() function is used to choose a route. The routing cache is constantly updated by the kernel and it has a limited space. Once the route has been entered in the routing cache, its reference count is incremented by one to ensure that the route is not inadvertently deleted as long as there is a skb referencing it.
The fields of the IP header are then filled . Then ip_options_build() handles the options if it is present. Then the netfilter hook NF_IP_LOCAL_OUTPUT is invoked.

ip_queue_xmit2()

The function sets the network device as specified by the routing cache. It also checks the MTU used by the device. It may happen that a packet is created for device 1 and its route has changed and it is sent over the device 2 which can handle smaller MTU. Thus the packet is checked for fragmentation and checksum is calculated.
Here the locally generated packet meets the path of a forwarded packet. The function pointer dst->output() which is set during the routing process causes the ip_output() function to be invoked().

ip_output()

This function calls ip_finish_output()

ip_finish_output()

This function sets skb-> device to the outgoing network device. The skb->protocol is set to layer 2 packet type ETH_P_IP .Then function calls NF_IP_POST_ROUTING hook

which in turn calls **ip_finish_output2().**

ip_finish_output2()

The Ethernet header is added to the skb and packet is sent.

8.7 Packet Forwarding by IP

If a computes has many network adapters and if IP forwarding is enabled, then the packets addressed to other computers are handled by the **ip_forward()** function. The Linux allows us to enable and disable the packet-forwarding mechanism at run time, provided the kernel was configured as router at the time of creating it. The directory /proc/sys/net/ipv4 includes a virtual file ip_forward. If 0 is written on to the file, the packet forwarding mechanism is disabled. To activate the packet forwarding mechanism we can give the command

echo '1' > /proc/sys/net/ipv4/ip_forward

ip_forward()

The main task of routing has already been handled in the ip_input() while determining if the packet belongs to this machine or it must be forwarded. First thing ip_forward() does is that packet marked with pkt_type == PACKET_HOST are deleted. Now if TTL field is one before decrementing, then the packet is deleted and an ICMP packet has to be returned to the sender informing ICMP_TIME_EXCEEDED.

For all confirming packets, after ensuring that there is sufficient headroom for the MAC header, TTL field is decremented by one.

Then packet size is compared against the MTU of the leaving device and if skb->len > MTU and do not fragment bit is set, packet is discarded and ICMP_FRAGMENT_NEEDED is sent to the sender. This is an early check for Do Not Fragment flag so that a Do Not Fragment candidate packet does not traverse the entire protocol stack only to be dropped eventually. If the packet passes the test, the fragmentation does not happen here. The packet is next handed over to netfilter hook NF_IP_FORWARD.

ip_forward_finish()

This function does not have much functionality except processing IP options, if any with the help of ip_forward_options(). ip_send() is invoked next.

ip_send()

ip_send() determines if the fragmentation is needed or packet should be passed to the ip_finish_outout().If the fragmentation is needed ip_fragment() is called.

Then the packet is passed to ip_finish_output(). Here the forwarded packet path meets the path of outgoing packets produced locally.

8.8 Method for producing log files

Our Standard method of producing the output is followed.

Figure - 11: Topology for capturing of IP Data

The kernel module in file myip_send.c is loaded on computer "Franklin" having an IP address 10.13.0.1 . Using the FTP protocol, we transfer a small file from the Franklin to Hawking, which has IP address 10.14.0.1 .

tcpdump is also run in parallel to record the packets . The output produced from the module is cleaned using a perl script and it produces output called LOG_IP_SEND. The tcpdump output produced is LOG_DUMP_SEND. Once we get the log file we unload the module.

Similarly the module in the file myip_rcv() is loaded and entire process repeated to produce LOG_IP_RCV and LOG_DUMP_RCV. The only difference being this time using FTP, we get the file from Hawking to Franklin.

8.8.1 Discussions on LOG_IP_SEND

LOG_IP_SEND : lines 1 to 6

These are the ARP packets seen by the IP and correspond to line 4 of LOG_DUMP_SEND the tcpdump log. We can see that ip_queue_xmit is not called in the case of UDP packets as explained earlier.

LOG_IP_SEND : lines 7 to 16

This is a syn packet and corresponds to the line 6 of tcpdump output LOG_DUMP_SEND .

The packet is first seen by the ip_queue_xmit that handles the routing for the packets and prepares some fields of the IP header as explained earlier. The packet is passed to IP_LOCAL_OUT_HOOK.(Line 8).

Then the packet is seen by the ip_queue_xmit2 where the packet is checked for the fragmentation. The checksum is also calculated here.

Then the packet is passed to the ip_output where the ip_finish output is called. ip_finish_output() function is not recorded in instrumentation as it is inline function and cannot be instrumented. After adding the device and protocol information in the skb this function passes, the packet to the NF_IP_POST_ROUTING hook (line 11).

Then the function ip_finish output2 handles the packet where an Ethernet header is added to the packet.

The entire log file is similar to above sequence thus not discussed further.

8.8.2 Discussion on LOG_IP_RCV

LOG_IP_RCV : lines 1

The packet is seen first by our new protocol . This corresponds to the Line 3 of the LOG_DUMP_RCV, the tcpdump output. The packet is an ARP packet.

It has been observed that if there is do not fragment bit set, IP does not give IP identification number to the packets.

LOG_IP_RCV : lines 14 to 26

This is syn packet received from hawking . It corresponds to line 5 of tcpdump output. ip_rcv() receives the packet first and after some sanity checking, passes the packet to the netfilter hook nf_ip_pre_routing . The packet is then received by the ip_rcv_finish().The route of the packet is determined here and the ip_route input is called. It was ascertained that the packet is for local delivery so ip_local_deliver() was called. This passes the packet to the NF_IP_LOCAL_IN hook . Then all the functions exit.

Entire log file follows this sequence thus not discussed further.

8.8.3 Forwarding of an IP Packet

The setup used for forwarding of the packets is that our module in file myip_forward.c is loaded in the computer B called Erwin, which acts as router.

File is transferred from computer A called Franklin having an IP address 10.13.0.1 to the computer C Gandalf having an IP 10.10.0.2. The acknowledgments from the Gandalf also follow the same path, in reverse.

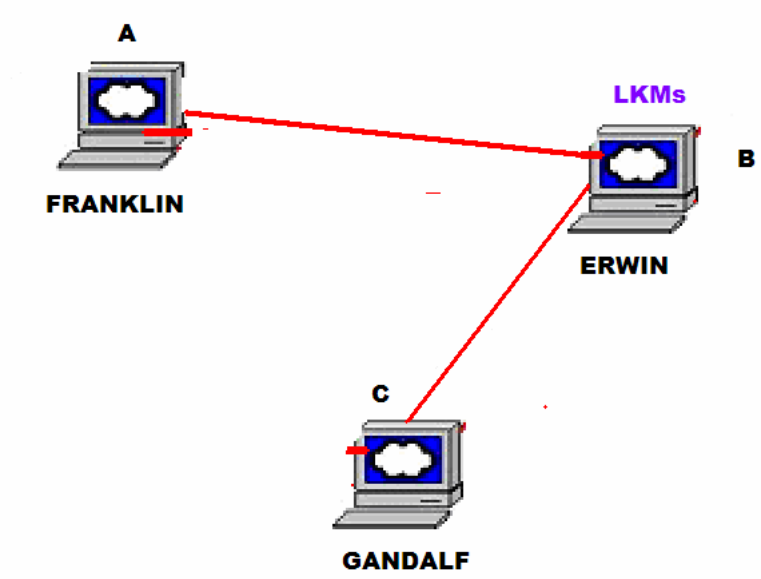


Figure - 12: Topology for capturing of IP Data forwarded by Linux

LOG_IP_FORWARD: LINES 1 to 7

We see as per our expectation the ip_forward is called first, which passes the packet to the netfilter hook NF_IP_FORWARD . Then the packet is passed on to the ip_forward finish function. Finally, ip_send hands over the packet to ip_finish_output. The functionality of these functions have already been discussed.

Entire log file follows the same pattern as above thus not discussed further.

9.0 UDP Implementation in Linux

9.1 General Methodology

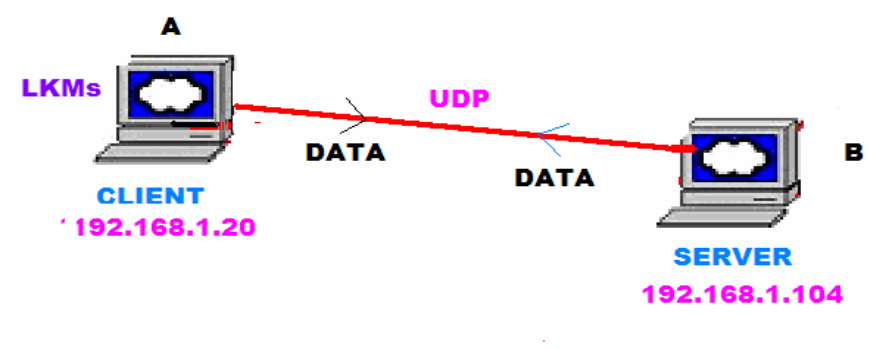


Figure -13: Topology for capturing of UDP Data

UDP packets are sent from the "Machine A" having an IP address 192.168.1.20 to the "Machine B" with an IP Address 192.168.1.104, over a LAN.

Specially written Client and Sever programs are used for data transmission between the two machines using UDP. Our Machine A runs the Client program and the Machine B runs the Server program.

The flow of data is as follows:

The Server is started on B and is waiting to receive a UDP datagram on PORT 30091. As soon as B receives a UDP datagram, it re-sends the datagram back to the sender using the incoming IP address and port. The transport layer protocol used is UDP.

A runs the Client program. The client program is started by providing the IP address of server as a parameter. It opens a socket and waits for an input from the standard input. As soon as a user inputs any line, it sends the text user entered using UDP protocol to the computer B and waits for a reply from B. On receipt of reply from B it prints the reply on to the screen.

The programs are enclosed called UDP Client and UDP Server. We shall not explain the socket programming. Refer to "The Unix Network Programming" [9] for further details.

9.2 UDP Functions Intercepted

UDP is a connectionless and unreliable protocol like IP. Apart from what IP does for the communication, UDP provides additional mechanism to address the ports and checksum the data. This makes UDP a very simple protocol and not many functions are involved in transmission and receiving of a datatagram. We have only one file, which instruments the Linux UDP functions. It intercepts the functions responsible for both transmission and receiving of data and is called udpio.c. Following are the UDP functions intercepted:

1.udp_close

2.udp_connect

3.udp_sendmsg

4.udp_recvmsg

5.udp_queue_rcv_skb

6.udp_v4_hash

7.udp_v4_unhash

8.udp_v4_get_port

9.udp_rcv

10.udp_getfrag

Our general methodology of producing output in /tmp/packet.log is followed. This log file is further cleaned for presentation using a perl script called format. The final Output

is called LOG_UDP. tcpdump is run in parallel while transmitting and receiving the data and the corresponding file produced by the tcpdump is called DUMP_UDP.

9.3 Handling of an Outgoing Datagram

The transmission of an UDP packet starts from a system call at the socket interface and runs all the way until a completed packet is added to the input queue of the network interface, all in one pass. Important functions that handle the packets are described below:

udp_v4_get_port()

When a socket is created, using a socket system call, one local port has to be assigned to the socket. This function is invoked by the PF_INET implementation.

udp_sendmsg()

When some data is to be sent, this function is invoked. It takes the parameters as socket, msg structure which specifies the destination, payload and the payload length in bytes.

There are two ways in which the details are stored in a sock structure:

a) If a programmer issues a command that leads to udp_connect system call, the destination address and the port is stored in a sock structure and in every send it need not be given.

b) In case the programmer does not use udp_connect() system call, then the destination address and the destination port is stored in msg_name element of the msg structure. In our client and the server programs we follow this method. Please refer to the programs for further details.

ip_route_output() is used to get the routing details of a packet.

Finally a datagram is passed over to the IP for transmission. IP function ip_build_xmit() is called. One of the parameters passed to the ip_build_xmit() is a pointer to the function udp_getfrag or udp_getfrag_nosum(), based on whether a checksum is needed or not .

udp_getfrag()

The functions udp_getfrag() and udp_getfrag_nosum() do the same thing except as the name indicates, the former is invoked when the checksum computation is required while the later when the checksum is not desired.

For every fragment generated by ip_build_xmit(), udp_getfrag() is called to get the required payload from the user space to the kernel space and also calculate the checksum. csum_partial_copy_fromiovecend() defined in net/core/iovec.c does much of this work.

udp_close()

Invoked during the closing of UDP sockets. It calls the function inet_sock_release() which in turn calls udp_v4_unhash().

All the UDP sockets are arranged in a hash table so that it is easier to locate them. Searching for a socket is required many times in the networking code, say when a datagram is received. If we do not have the right socket open for the datagram, it must be dropped. This warrants searching for a socket. The Hash table is called struct sock*udp_hash[UDP_HASHTABLE_SIZE]. Port number modulo UDP_HASHTABLE_SIZE is used as a hashing function.

udp_v4_unhash()

udp_v4_unhash() function is invoked to remove the socket from a hash table. This happens when a user wants to stop udp communication by issuing a udp_close() directive.

udp_v4_hash()

This can be used to enter the socket into a hashing table. In practice udp_v4_get_port() completes the process of entering the socket in a hash table, apart from assigning the local port number. Thus, this function is never called.

9.4 Handling of an Incoming Datagram

Receiving of UDP packets requires two separate passes. udp_rcv() receives the packet from the IP and deposits it into the socket's receive queue. A user program fetches the packet using a system call that is mapped to the udp_recvmsg() function. Important UDP functions that handle the receiving packets are being described here.

udp_rcv()

IP passes a suitable receiving packet to udp_rcv() . udp_rcv() calls udp_v4_lookup() to get the right socket associated with the packet.

udp_queue_rcv_skb()

This inserts the SKB in the socket's receive queue .

udp_recvmsg()

When we execute the recvfrom() in our socket program of the client/server at socket interface, the operating system maps the function call to the udp_recvmsg(). This function remmoves the SKB from the receive queue of the socket and passes the payload

contained along with the necessary header information in the form of entry in the msghdr structure, passed by reference.

In case the socket receive queue is empty, then either the process is put to waiting or the call is terminated based on the socket programming specifications.

There are generic functions to handle retrieval of a sk_buff and copying the sk_buff data which are used not only by the UDP but at other places too. These functions are in the file net/core/datagram.c. The functions are like skb_recv_datagram(), skb_copy_datagram_iovec(), skb_free_datagram() which we shall not look into in this project.

9.5 Discussion on LOG_UDP

The line numbers in the following discussion always refer to the LOG_UDP and whenever we refer to DUMP_UDP we shall say so specifically.

Line 1,2

local port is selected for transmission of the datagram.

Line 3 to 6

Datagram is sent using the udp_sendmsg() and we can see that the ip_build_xmit() calls udp_getfrag() to copy the data from the user_spcae to the kernel space.

Lines 7-12

We can see that the function call at the user land translates to the udp_recvmsg which calls udp_rcv. udp_queue_rcv_skb() is called by the udp_rcv() which queues the datagram in the appropriate socket's receive queue.

We can see that the IP numbers in many cases are 0 because the UDP is connectionless protocol and the destination ip numbers are not stored in sock structure unless otherwise we do so specifically using socket programming..

Rest of the log file is similar to whatever we have already discussed thus not discussed further.

Lines 27-30

We close the socket and the udp_close() is executed which calls udp_v4_unhash to drop the socket from the hash table.

The tcpdump file showing transmission of two packets is DUMP_TCP and confims the above discussion.

1.0.0 Conclusion

"Instrumenting Linux to Collect the Traces of Individual Communication Packets" is the project that uncovers the internal handling of the packets by the Linux Kernel. Redhat Linux 9 and kernel 2.4.20-8 was used for experimentation.

The protocols covered in the project were TCP, IP and UDP. A detailed analysis of the protocol stack as well the technique used to study the protocol stack was presented.

The project also attempted to uncover the details of kernel module programming and how to utilize the knowledge of the module programming for Kernel Function Hijacking.

This was my first exposure to the kernel programming and when I look back, I feel I could have done many things differently. In particular, I should have included the device layer and the INET layers in the discussions. I should also have written one generalized function to print the packet details instead of the multiple functions which do the task at present. I should have used architecture specific could method to get more resolution for measurement of time.

These are the tasks which require improvements.

This project could be very easily extended further to include the following:

1. Include the other layers of the protocol and even other protocols.
2. Collect and analyze the resource consumption by the various functions/protocol layers.
3. Study the implementations of various queues in the protocol implementation.

Linux kernel is very complex piece of software and the details covered in the project are the tip of the iceberg. Nevertheless, I have made the beginning and shall continue the journey of uncovering the mysteries of the kernel.

11.0 References

[1] Herald Welte (10/2000) The journey of a packet through Linux 2.4 network stack. http://gnumonks.org/ftp/pub/doc/packet-journey-2.4.html

[2] Glenn Herrin (May 31,200) Linux IP Networking, A guide to implementation and Modification of Linux Protocol Stack.

http://kernelnewbies.org/documents/ipnetworking/linuxipnetworking.html

[3] M. Rio et al. (March 31, 2004) A map of the Networking Code in Linux Kernel 2.4.20. http://www.labunix.uqam.ca/~jpmf/papers/tr-datatag-2004-1.pdf

[4] Silvio Cesare. (November 1999). Kernel Function Hijacking http://www.rfxnetworks.com/docs/kernel-hijack.txt

[5] Jennifer Hou, Department of Computer Science, University of Illinois at Urbana-Champaign. *Lectures of CS 498.* http://www-courses.cs.uiuc.edu/~cs498hou/lectures/

[6] Klaus Wehrle, Frank Pahlke, Hartmut Ritter, Daniel Muller, Marc Bechler. (2005) *The Linux Networking Architecture, Design and Implementation of Network Protocols in the Linux Kernel*

Prentice Hall

[7] Guo huanxiong, and Zheng Shaoren . Analysis and Evaluation of the TCP/IP Protocol Stack of LINUX. www.ifip.or.at/con2000/icct2000/icct452.pdf

[8] Peter Jay Salzman and Ori Pomerantz.(2001) The Linux Kernel Module Programming Guide. http://www.faqs.org/docs/kernel/

[9] W. Richard Stevens (1998) *Unix Network Programming, Volume 1, Second Edition: Networking APIs: Socket and XTL.* Prentice Hall

12.0 Appendix