“Once Only” Drop Capability in the Linux Routing Software

Submitted to the
Department of Computer Science
College of Computing Sciences
New Jersey Institute of Technology

in Partial Fulfillment of
the Requirements for the Degree of
Master of Science

By
Viraj Ajgaonkar

APPROVALS

Proposal Number:-___________________________________

Approved by:

____

(Dr. Teunis J. Ott)

Date Submitted:

____________________________________-

Abstract

This Project adds a capability to the existing Linux Routing Software. The new capability is to give a number n and a specific pattern to the software and make the software drop the packet if the packet contains the pattern and is within the first n packets of the flow having the pattern. Here n is known as limit of the flow. Subsequent packets of that flow containing the same pattern need not be dropped.

e.g.:- Let the pattern be "yikes" and the limit be set to 3. Then for each TCP/IP flow the first 3 packets containing "yikes" are dropped. The subsequent packets containing "yikes" are not dropped.

For testing and verification this project also makes use of some network sniffer softwares like Tcpdump and Ethereal.

Contents

i Title Page
ii Acceptance/Approval Page
iii Abstract
iv Table of Contents

1. Introduction..5

1.1 Problem Definition..5

1.2 Previous Work..7

1.3 Glossary..7

2 Design..9

2.1 Approach..9

2.2 Deciding Factor..12

2.3 Netfilter Hooks..14

2.4 Packet Handling in Linux Kernel 2.4..17

2.5 Decision of the Hook..23

2.6 Algorithm of my project..26

3 Implementation..28

3.1 Code..28

3.1.1 Header File for data structure..28

3.1.2 Initialization and Cleanup..30

3.1.3 Makefile..31

3.1.4 Main Module..32

3.2 Explanation of the Code..39

3.2.1 Flow of the Program..39

3.2.2 Function Description..42

3.2.3 Useful Calculations..45

4 Tests and Results..47

5 Applications..62

6 Future Enhancements..65

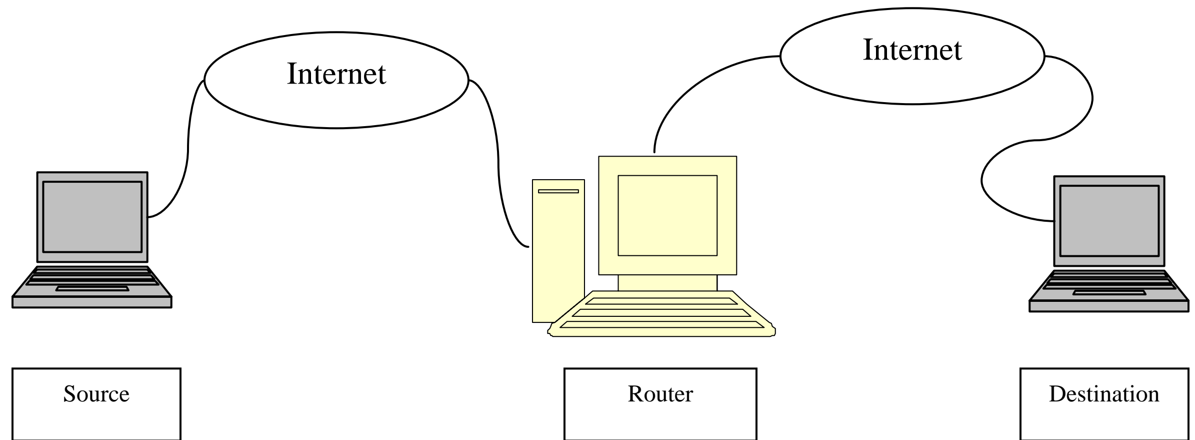
7 Bibliography..66

1. Introduction

1.1 Problem Definition

My project aims at building customizable Linux Routing software. Linux is open source i.e. it allows user to modify the operating system according to their requirement. My project uses this feature to access some 'Network' variables in the Linux Routing software of the kernel in our Router. Also we have added some user defined functions which will process the routing mechanism in a desired way and we have studied the effects of these modifications on the flow of data between different systems that passes through this router.

The project customizes the existing Linux kernel routing software to give the user the ability to drop a particular packet based on the contents of that packet and also gives the user complete control on the number of packet of any flow having that content to be dropped.



Considering the above figure, Source and Destination are two hosts separated by either one or more networks. The Router is a host on one of the networks between Source and Destination also Router forwards the packets from the Source to Destination and vice-versa. My software which is loaded at the Router adds a new capability to it. This capability is to drop the first n packets of the flow from Source to Destination which contains a particular string in the TCP Data.

e.g.:- Considering the flow from Source to Destination. If the packet of this flow does not have the string “yikes” then the packet is forwarded to Destination. The first 10 packets of this flow having the string “yikes” in the data part are dropped and all subsequent packets of the flow containing “yikes” are let through. Thus we can do

packet filtering on the basis of the content and also control the number of filtered packets.

1.2 Previous Work

This project study is based on the information provided by the Linux 2.4.xx kernel code, documentation files and different RFCs. The previous work in the field of customizable Linux Routing software targeted routers that were capable of dropping all packets from a set of source hosts or going to a set of destination hosts or passing through a particular interface. However not much work has been done for routers that drop packets based on their data contents and also routers where users can control the numbers of packets to be dropped. This project gives the user ability to drop packets in a router based on the data content and to decide the number of packets to be dropped.

1.3 Glossary

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 datagrams.

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

Destination: A host that 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 60's.

DNS: Domain Name Server

GOME: GNU Network Object Module Environment

LILO: LInux LOader

MTU: Maximum Transfer Unit

NIST Net: National Institute of Standards and Technology (NIST) Network emulation tool

RTT: Round Trip Time

2. Design

2.1 Approach

There are 2 possible approaches for implementation

- Kernel recompilation.
- Modules using Netfilter hooks.

Detail Description of the methods

- Kernel recompilation: In the kernel recompiling method we have to go through the Linux networking code located in the dir **xxxxxxxx** and find the exact function where we have to insert our code. Then we have to make the appropriate changes to that function to implement the required functionality. A good practice of doing this is to comment the original function and replace it with the new function having the same definition. Finally we have to recompile the modified Linux Kernel. Recompiling creates a new Kernel image incorporating the changes we made. Kernel recompilation keeps all the previous Kernel images unaltered. Then in order to see the effect of the changes we have to reboot and boot up using the newly created Kernel image.
- Modules using Netfilter hooks: The approach of Modules using Netfilter hooks has been explained in

detail in chapter 2.3 which follows. In this approach initially we see the locations of all the available Netfilter hooks and decide as to which hook is the best possible location for inserting our functionality. Then we design and implement the functionality and save it in the form of a handler function which is coded in C. e.g. the handler function can be saved as **router.c**. Then we create a module in which we register the handler function and in this module we also bind the module to one of the available Netfilter hooks. E.g. we can save the module as **viraj.c**. Lastly we write a make file (**makefile**) that compiles the module (**viraj.c**). Now we compile the module by executing **"make"**. The compilation gives us a executable file (**viraj.o**). Now inorder to load the module and see the effects we need to install the module using the command **"insmod viraj.o"**. Finally to stop the module and also its effect we have to uninstall the module which is done using the command **"rmmod viraj"**. The effects of the module are automatically written in the log which is located in the file **/var/logs/messages**. The process of installing and uninstalling the module can be compared logically to

mounting and unmounting of an external drive. e.g.:- Firstly we mount the floppy drive using **"mount /mnt/floppy"**. Then when we have done the required operation of the floppy drive and finally we unmount the floppy drive using **"umount /mnt/floppy"**. Similarly we install the module using **"insmod viraj.o"**. The module binds itself to the proper hook and whenever the execution point reaches the hook, this module is executed and this module inturn calls the handler function (**router.c**). The handler function returns one of the standard return types and ends its execution. Finally when we want to stop the effect of the module we simply uninstall the module using **"rmmod viraj"**. For any subsequent executions of the kernel when the execution point reaches the hook it doesn't find any module installed hence the execution will pass to the next line of the Linux kernel code and the handler module is not called.

2.2 Deciding Factor

- My project involves making changes to Linux routing software and studying the effects of these changes.
- In recompilation approach for every change we have to recompile the entire kernel which takes a long time (approx 45 mins) also frequent recompiles makes the kernel fragile.
- In Netfilter hook option, we can see effects of the changes we made in a short time (2 secs) by just recompiling the module as compared to the lengthy recompilation of the entire kernel.
- By using Netfilter hooks the module written is portable as we can implement the same functionality on any Linux router by simply copying and loading the module. Whereas in kernel recompiling approach the benefit of the code is available only on the recompiled kernel.
- Once the changes have been made to the kernel the effect is seen for the entire uptime of the new kernel. Using modules fitting in Netfilter hooks we see the effect of the module by installing the module and stop the effect by simply uninstalling

the module. There by we have complete control over the module. This is very useful as we can get results only for the required time making it easy to analyze data and infer important results.

2.3 Netfilter Hooks

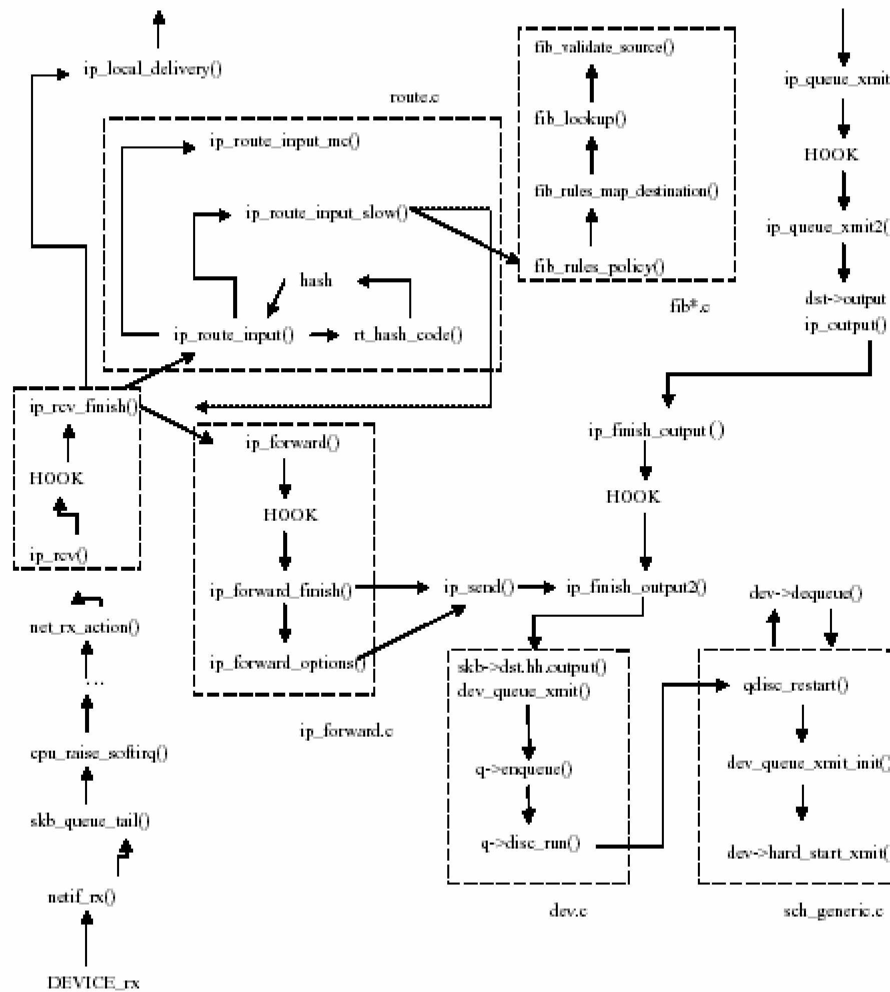
Netfilter is a subsystem in the Linux 2.4 kernel. Netfilter makes network functionalities such as packet filtering, network address translation (NAT) and connection tracking possible through the use of various hooks in the kernel's network code. These hooks are places that kernel code, either statically built or in the form of a loadable module, can register functions to be called for specific network events. An example of such an event is the reception of a packet.

Netfilter defines five hooks for IPv4. The declaration of the symbols for these can be found in linux/netfilter_ipv4.h. These hooks are displayed in the table below:

Table: Available IPv4 hooks

Hook	Called
• NF_IP_PRE_ROUTING	After sanity checks, before routing decisions.
• NF_IP_LOCAL_IN	After routing decisions if 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 "hit the wire".



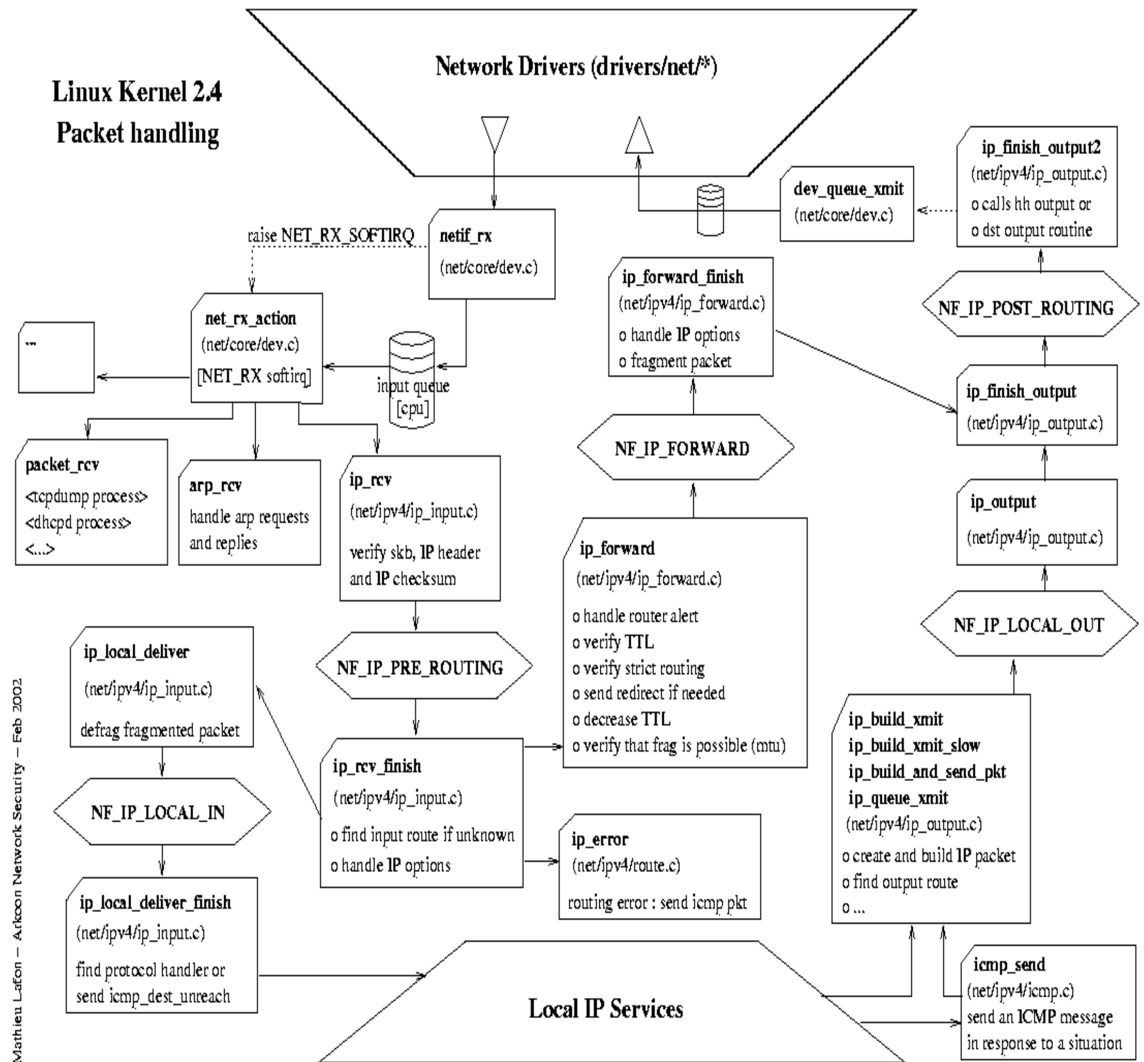
After hook functions have done whatever processing they need to do with a packet they must return one of the predefined Netfilter return codes. These codes are:

Table 2: Netfilter return codes

Return Code	Meaning
• NF_DROP	Discard the packet.
• NF_ACCEPT	Keep the packet.
• NF_STOLEN	Forget about the packet.
• NF_QUEUE	Queue packet for user space.
• NF_REPEAT	Call this hook function again.

2.4 Packet Handling in Linux Kernel 2.4

This chapter describes the packet handling mechanism of the default Linux Kernel 2.4. This figure below shows the Packet Handling in the default Linux Kernel 2.4.



When the packet reaches the host from the network, it goes through the network layer functions and reaches net_rx_action() which is the last function in device layer. From net_rx_action() the packet is passed to ip_rcv() which is the first function in IP layer. ip_rcv() verifies the skb, IP Header and IP checksum. Subsequently the packet is passed to the first Netfilter hook i.e. **NF_IP_PRE_ROUTING** hook. If there is a module installed at this hook the execution control is taken over by the handler function of the module, which does packet processing and returns one of the predefined Netfilter return codes (explained in chapter 2.3) and passes out of the Netfilter hook. If there is no module attached to the hook then the execution passes through the hook without doing any processing. After passing the first Netfilter hook the packet reaches ip_rcv_finish(), which verifies whether the packet is for local delivery.

If it is addressed to this host, the packet is given to ip_local_delivery(), which defragments the fragmented packet. Subsequently the packet is passed to the second Netfilter hook i.e. **NF_IP_LOCAL_IN** hook. If there is a module installed at this hook

the execution control is taken over by the handler function of the module else the execution passes through the hook without doing any processing. After passing the second Netfilter hook the packet reaches ip_local_deliver_finish() which sends icmp_dest_unreach or finds the protocol handler and gives it to the appropriate transport layer function (e.g. TCP or UDP). A packet can also reach the IP layer coming from the upper layers (e.g. delivered by TCP, or UDP, or coming directly to the IP layer from some applications). The first function to process the packet is then ip_queue_xmit(), which creates and builds ip packet and computes the output route. Subsequently the packet is passed to the fourth Netfilter hook i.e. **NF_IP_LOCAL_OUT** hook. If there is a module installed at this hook the execution control is taken over by the handler function of the module else the execution passes through the hook without doing any processing. After passing the fourth Netfilter hook the packet reaches ip_queue_xmit2(), which passes it to ip_output() to be passed to ip_finish_output(). In the output part, the last changes to the packet are made in ip_finish_output(). Subsequently the packet is

passed to the fifth Netfilter hook i.e. **NF_IP_POST_ROUTING** hook. If there is a module installed at this hook the execution control is taken over by the handler function of the module else the execution passes through the hook without doing any processing. After passing the fifth Netfilter hook the packet reaches ip_finish_output2(). Final ip layer changes are made to the packet in the function ip_finish_output2() and the function dev_queue_transmit() is called; the latter enqueues the packet in the output queue. It also tries to run the network scheduler mechanism by calling qdisc_run().

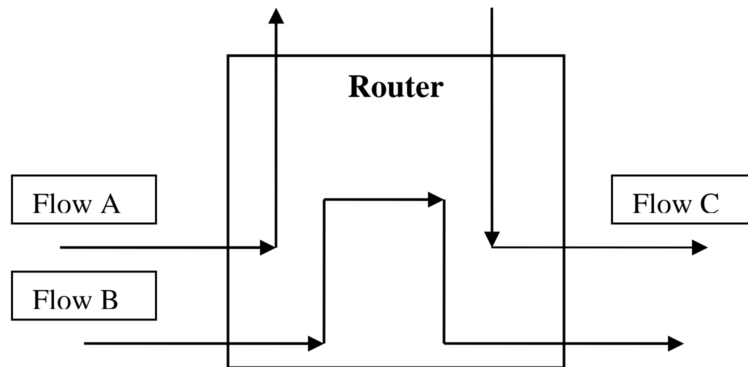
If an incoming packet has a destination IP address other than that of the host, the latter acts as a router (a frequent scenario in small networks). If the host is configured to execute forwarding (this can be seen and set via /proc/sys/net/ipv4/ip_forward), it then has to be processed by a set of complex but very efficient functions. If the ip_forward variable is set to zero, it is not forwarded. The route is calculated by calling ip_route_input(), which (if a fast hash does not exist) calls ip_route_input_slow(). The

ip_route_input_slow() function calls the FIB (Forward Information Base) set of functions in the fib*.c files. The FIB structure is quite complex. If the packet is a multicast packet, the function that calculates the set of devices to transmit the packet to is ip_route_input_mc(). In this case, the IP destination is unchanged. After the route is calculated, ip_rcv_finished() inserts the new IP destination in the IP packet and the output device in the sk_buff structure. The packet is then passed to the forwarding function ip_forward() which handles the route alert, verifies TTL, verifies Strict Routing, sends Redirect if necessary, decreases TTL and verifies that fragmentation is possible based on mtu. Subsequently the packet is passed to the third Netfilter hook i.e. **NF_IP_FORWARD** hook. If there is a module installed at this hook the execution control is taken over by the handler function of the module else the execution passes through the hook without doing any processing. After passing the third Netfilter hook the packet reaches ip_forward_finish() which handles IP options, fragments the packet if necessary and

sends it to the output components i.e. ip_finish_output().

2.5 Decision of the Hook

Objective of the project is to customize the firewall software for a router, so that the router can filter packets based on TCP data and can also control the number of packets of each flow to be filtered. There are three types of TCP/IP flows that can pass through a router. They are as shown in the figure below:



Flow A TCP/IP flows where the Destination is the **Router** itself.

Flow B TCP/IP flows where the **Router** is neither the source nor the destination.

Flow C TCP/IP flows where the Source is the **Router** itself.

The flows that are of importance to us are of type Flow B. Now if we install our software at NF_IP_LOCAL_IN, we are able to catch all the packets

of Flow A but we miss the packets of Flow B, hence NF_IP_LOCAL_IN is not considered for installing our module. Similarly if we install our software at NF_IP_LOCAL_OUT, we are able to catch all the packets of Flow C but we miss the packets of Flow B, hence NF_IP_LOCAL_OUT is not considered for installing our module. If we install our module at either NF_IP_PRE_ROUTING or NF_IP_POST_ROUTING or NF_IP_FORWARD, we are successfully able to catch all the packets of flow B which is required. To shortlist among these three hooks we consider the fact that if we install our software at NF_IP_POST_ROUTING or NF_IP_FORWARD hook we always do complex calculation of the route for the packet before we decide whether the packet needs to be dropped or not. Hence there are some cases where the route is calculated but not used as the packet is dropped. The advantage of using NF_IP_PRE_ROUTING hook is that we take decision regarding dropping of the packet in advance; hence we calculate the route for only the relevant packets. Thus using NF_IP_PRE_ROUTING hook saves us a lot of overhead. Second incentive to choose NF_IP_PRE_ROUTING hook is that, at this hook we receive the entire sk_buff so

the memory allocation is simple (contiguous) and we do not have to worry about fragmentation of sk_buff. In the other hooks the sk_buff we receive has not been fully realized hence we have complexity to account for sk_buff fragmentation due to noncontiguous memory allocation.

2.6 Algorithm of my project

In the router we check all the incoming packets and by reading the contents get information about the source ip, destination ip, source port and destination port if it is a TCP/IP packet. Using this information we determine as to which flow this packet belongs to. For the first packet of each flow that the router encounters, a new node (data structure i.e. sock_details) is created and is appended to an existing link list (pointed to by head_list). This link list is a collection of information of all the flows, i.e. it is a linked list of sock_details, such that head_list points to the details of the first active flow. Then we check if this flow already exists if it does then no new node is created.

By doing pointer arithmetic we can read the TCP data and also simultaneously search the string. If the string is found then the found flag is set to 1 and also the count is incremented by 1. Subsequently we extract other information which is required to maintain the flows, like status of finish flag in the packet. Then we check if (count > limit). If (count > limit) and if found is

set to 1 then the drop flag is set to 1 else drop flag is set to 0. Finally if the drop flag is 1 the packet is dropped using NF_DROP return type else the packet is let through using NF_ACCEPT return type.

3. Implementation

3.1 Code

3.1.1 Header File for data structure

```
#include </usr/src/linux-2.4/include/asm-i386/types.h>
struct sock_details
{
int num_packs_mytype; //count of number of packets of
                      //the flow encountered by the
                      router
int count; //count of times the pattern is found for the
                      flow
int limit; //number of packets having the pattern to be
                      dropped
int finish;//indicates if the FIN flag has been
                      encountered for //the flow
char *reg_exp;//defines the pattern or search string
__u16 sport,dport;//indicates the ports of source and
                      //destination computers of the flow
__u32 saddr,daddr;//indicates the ip address of source
                      and //destination computers of the
                      flow
struct sock_details *next;
struct sock_details *prev;
};
struct head_list
```

```
{
int tot_num_packs;// number of packets encountered by the
router
int num_of_socks;//count of the total number of flows
                active at //any point
struct sock_details *first_socket;//pointer to the
                                  details of the //first
                                  active flow
};
```

3.1.2 Initialization and Cleanup

```
/* Initialization routine */
int init_module()
{
    /* Fill in our hook structure */
    nfho.hook     = hook_func; /* Handler function
*/
    nfho.hooknum  = NF_IP_PRE_ROUTING; /* First for
IPv4 */
    nfho.pf       = PF_INET;
    nfho.priority = NF_IP_PRI_FIRST; // Make our
func first
    nf_register_hook(&nfho); //registering the hook
return 0;
}

/* Cleanup routine */
void cleanup_module()
{
    nf_unregister_hook(&nfho); //unregistering the
hook
}
```

3.1.3 Makefile

```
#Makefile

CC= gcc -I/usr/src/linux-2.4/include

CFLAGS = -O2 -D__KERNEL__ -Wall

hook1.o:hook1.c
```

3.1.4 Main Module

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/skbuff.h>
#include <linux/ip.h>      /* For IP first_socketer */
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/state_table.h> /*/usr/src/linux-2.4.20-
18.9/include/linux*/

int i;//my variable for the for loop
static struct sock_details *current_packet;//used for deletion
of a node
static struct sock_details *current_prev;//used for deletion
of a node
static struct sock_details *current_next;//used for traversing
the linked list of flows
static struct sock_details *printer;//used for traversing the
linked list of flows during printing
static struct head_list head = {0, 0, NULL};
__u16 packet_sport,packet_dport;
__u32 packet_saddr,packet_daddr;
__u8 TCP_HLEN,FIN_FLAG,PROTOCOL;
static char *tcp_data_ptr;
static char *temp_ptr;
char c;
static char *drop_if = "rainfall";
static char *check_if;
static int found,length,drop;


/* This is the structure we shall use to register our function
*/
static struct nf_hook_ops nfho;

/* This is the hook function itself */
unsigned int hook_func(unsigned int hooknum,
                        struct sk_buff **skb,
                        const struct net_device *in,
                        const struct net_device *out,
                        int (*okfn)(struct sk_buff *))
{

struct sk_buff *sb = *skb;
head.tot_num_packs++;
drop =0;

void process_drop()
{
current_packet = head.first_socket;
for( i=0; i<head.num_of_socks;i++)
{
```

```
if(current_packet->saddr == packet_saddr && current_packet-
>sport == packet_sport && current_packet->daddr ==
packet_daddr && current_packet->dport == packet_dport)
{//if the flow is found among existing flows in the list
if(current_packet->count > current_packet->limit)
        drop =0;
else
        drop =1;
break;
}
else
{
if(current_packet->next == NULL)
        {//we reached the end of the list of flows and still no
match found
                drop =0;
        }
else
        {
        current_packet = current_packet->next;
        }
}

}
}
void create_check_if()
{
temp_ptr = (char *) (tcp_data_ptr);
check_if= strncpy (check_if,temp_ptr,length);
//for(i=0;i<length;i++)
//{
//      printk("%c", *(char *)(tcp_data_ptr + i));
//}
//printk("\n");
}

void process_found()
{
found =0;
tcp_data_ptr = ((sb->data + (sb->nh.iph->ihl * 4))+(TCP_HLEN
*4));
printk("The TCP Data is \n");

while(tcp_data_ptr != ((char *)(sb->tail)))
{
printk("%c", *(char *)(tcp_data_ptr));
tcp_data_ptr++;
}
printk("\n");

length = strlen(drop_if);
check_if = (char *)kmalloc(sizeof(char),GFP_KERNEL);
while((((char *)(sb->tail)) - tcp_data_ptr) > length)
{
kfree(check_if);
check_if = (char *)kmalloc(sizeof(char),GFP_KERNEL);
create_check_if();
```

```
if(strncmp(drop_if,check_if,length) == 0)
{
found =1;

break;
}
else
{
found = 0;
}
tcp_data_ptr++;
}

if(found == 1)
printk("The string is found \n");
else
printk("The string is NOT found \n");

}

void print_list()
{
printer = head.first_socket;//initialising the cursor to first
node
if(head.num_of_socks != 0 )//checking to see an empty list
{
for(i=0;i<head.num_of_socks;i++)//traversing the entire list
{
printk("----------------------------------------------------
---------\n");
printk("For flow %d number of packet = %d \n", i+1, printer-
>num_packs_mytype);
printk("S_Addr = %x   D_Addr = %x   S_port = %x   Dport =
%x\n" , printer->saddr,printer->daddr,printer->sport,printer-
>dport);
printk("Count %d  Limit %d Finish %d  REGULAR EXPRESSION : %s
\n", printer->count, printer->limit, printer->finish, printer-
>reg_exp);
printk("----------------------------------------------------
---------\n");
printer = printer->next;
}
}
}

void insert()
{
struct sock_details *temp;//creating a temp node to be
attached to link list
if(head.first_socket == NULL)//checking to see if the list is
empty
{
temp = (struct sock_details *)kmalloc(sizeof(struct
sock_details),GFP_KERNEL);
head.first_socket = temp;
temp->num_packs_mytype = 1;
```

```
temp->saddr = packet_saddr;
temp->daddr = packet_daddr;
temp->sport = packet_sport;
temp->dport = packet_dport;
if(found == 1 )
{
temp->count = 1;
}

else
temp->count = 0;
temp->limit = 10;
temp->finish = 0;
temp->reg_exp = "viraj";
temp->prev = NULL;
temp->next = NULL;
}
else
{
temp = (struct sock_details *)kmalloc(sizeof(struct
sock_details),GFP_KERNEL);
temp->num_packs_mytype = 1;
temp->saddr = packet_saddr;
temp->daddr = packet_daddr;
temp->sport = packet_sport;
temp->dport = packet_dport;
if(found == 1 )
{
temp->count = 1;
}
else
temp->count = 0;
temp->limit = 10;
temp->finish = 0;
temp->reg_exp = "viraj";
current_packet->next = temp;
temp->prev = current_packet;
temp->next = NULL;
}
print_list();//printing the current linked list of TCP/IP
flows
}

void finish()
{
current_packet = head.first_socket;
for( i=0; i<head.num_of_socks;i++)
{
if(current_packet->saddr == packet_saddr && current_packet-
>sport == packet_sport && current_packet->daddr ==
packet_daddr && current_packet->dport == packet_dport)
{//if the flow is found among existing flows in the list
current_packet->finish = 1;
current_packet->num_packs_mytype++;
if(found == 1)
current_packet->count++;
break;
```

```
}
else
{
if(current_packet->next == NULL)
{//we reached the end of the list of flows and still no match
found
head.num_of_socks++;
insert();
current_packet = current_packet->next;
current_packet->finish = 1;
}
else
{
current_packet = current_packet->next;
}
}

}
}

void process()
{
current_packet = head.first_socket;
if(head.num_of_socks == 0)
{     //if the linked list is empty i.e. there are no previous
flows
head.num_of_socks++;
insert();
}
else
{
for(i=0;i<head.num_of_socks;i++)//traversing the non empty
linked list
{
if(current_packet->saddr == packet_saddr && current_packet-
>sport == packet_sport && current_packet->daddr ==
packet_daddr && current_packet->dport == packet_dport)
{     //if the flow is found
if(current_packet->finish == 1)//if this is the packet after
Fin packet
{
//delete entry actually

if ((current_packet->prev == NULL) & (current_packet->next ==
NULL))
{     //if there is only one node in the linked list
head.first_socket = NULL;
head.num_of_socks = 0;
}
else
{
if(current_packet->prev == NULL)
{     //if the first node is to be deleted from the list
current_next = current_packet->next;
head.first_socket = current_next;
current_next->prev = NULL;
current_packet = current_next;
```

```
head.num_of_socks--;
}
else
{
if(current_packet->next == NULL)
{      //if the last packet is to be deleted from the list
current_prev = current_packet->prev;
current_prev->next = NULL;
current_packet = current_prev;
head.num_of_socks--;
}
else
{      //if the node to be deleted from the linked list is
between 2 nodes
current_prev = current_packet->prev;
current_next = current_packet->next;
current_prev->next = current_next;
current_next->prev = current_prev;
current_packet = current_prev;
head.num_of_socks--;
}
}
}


print_list();
}
else
{
current_packet->num_packs_mytype++;
if(found == 1)
current_packet->count++;
printk("Number of packets of this flow = %d \n" ,
current_packet->num_packs_mytype);
print_list();
}
break;
}
else
{//if the flow isn't found in the linked list of existing
flows
if(current_packet->next == NULL)
{      //if we reach the end and still no match is found
//in existing flows we create new entry for this flow
head.num_of_socks++;
insert();
}
else
       current_packet = current_packet->next;


}
}
}
}
```

```
packet_saddr = sb->nh.iph->saddr;//getting Source Adddr of
packet
packet_daddr = sb->nh.iph->daddr;//getting Destn Adddr of
packet
packet_sport = *(unsigned int *) (sb->data + (sb->nh.iph->ihl
* 4));//getting Source Port of packet
packet_dport = *(unsigned int *) ((sb->data + (sb->nh.iph->ihl
* 4)) + 2);//getting Destn Port of packet

if( packet_saddr != packet_daddr )
{
printk("S_ADDR %x   ", packet_saddr);
printk("D_ADDR %x   ", packet_daddr);
printk("S_PORT %x   ", packet_sport);
printk("D_PORT %x  ", packet_dport);
TCP_HLEN = *((sb->data + (sb->nh.iph->ihl * 4)) +
12);//getting TCP Header Length
TCP_HLEN = TCP_HLEN / 16;
printk("Correct TCP_HLEN %d  ", TCP_HLEN);//printing TCP
Header Length
FIN_FLAG = *((sb->data + (sb->nh.iph->ihl * 4)) + 13);
//getting the TCP Flags
FIN_FLAG = FIN_FLAG % 16;
FIN_FLAG = FIN_FLAG % 2;
PROTOCOL = (sb->nh.iph->protocol);//getting the Transport
Layer Protocol
printk("Protocol is  %d   \n ", PROTOCOL);//printing the
Transport Layer Protocol
if(PROTOCOL == 6)//checking to see if it is a TCP/IP Packet
{
process_found();
if(FIN_FLAG == 1)
{
printk("I am dying\n\n");
finish();
}
else
process();
if (found == 1)
process_drop();
}
if(drop == 1)
{
print_list();
printk("This packet is dropped\n");
return NF_DROP;
}
else
return NF_ACCEPT;
}
else
{
return NF_ACCEPT;
}

}
```

3.2 Explanation of the Code

3.2.1 Flow of the Program

Functions and Program Flow

- check if it is TCP/IP packet.
- get packet info and flow info.
- process_found() -> create_if()
- process()
- finish()
- insert()
- process_drop()

The various functions in the module and the flow of execution in the program are as follows.

For each incoming packet we first check if it is a TCP/IP packet, this is done by checking if (sb->nh.iph->protocol) == 6. If the packet is a TCP/IP packet, then we get information about the packet as well as other data that gives us information about the flow to which the packet belongs. This information consists of the source IP Address, Destination IP Address, Source Port, Destination Port, TCP Header Length and the status of Fin Flag in the packet.

Then the function process_found() is called. In this process we traverse the entire TCP data until we reach the end of TCP data marked by (sb->tail). During the traversal if the search string is found, then we set the 'found' flag to 1 and break from the loop to avoid further traversal of TCP Data. If even at the end of TCP Data the pattern is not found then we do not change the value of 'found' flag which by default is set to 0. By looking at the found flag at any future point of packet processing we can tell if it contains the search string or not.

After process_found() has done it's processing we call the function finish() else we call the function process(). In the functions finish() and process() we do certain processing required for maintaining the status of the flows, also if the 'found' flag is set to 1 we increment the value of count for the flow in the node by one else if 'found' flag is set to 0 we do not make any change to count. If during processing of functions finish() and process() we find that the packet is the first packet of the flow, we create a new node for the flow by calling the function insert().

In the function insert() we create a new node of the type sock_detail and enter important information like source IP Address, Destination IP Address, Source Port, Destination Port. Also we set the proper values of limit, count, finish. Then we insert the node at the end of the linked list.

Then if ‘found’ == 1 we call the function process_drop(). In process_drop(), we check if the count for this flow has exceeded the limit of the flow, this is done by checking if (count > limit). If (count > limit), then ‘drop’ flag is set to 0 else ‘drop’ flag is set to 1. Finally if ‘drop’ flag is set to 1 the packet is dropped using NF_DROP return type, else if ‘drop’ flag is set to 0 the packet is let through using NF_ACCEPT return type.

3.2.2 Function Description

- **process_found():** This function traverses the entire TCP Data and simultaneously checks if the search string is found. If the search string is found the traversing of TCP Data is stopped and the 'found' flag is set to 1, before the function process_found() ends it's processing. If the search string is not found at the end of the traversal then the function process_found() ends it's processing without changing the value of 'found' flag which by default is set to 0. In this project a naïve pattern matching algorithm has been used. This can be improved in future.
- **process():** This function checks if the flow which the packet belongs to exists in the linked list, if it doesn't this function calls the function insert() else it finds the relevant node and does further processing. Further it checks if the 'found' flag is set to 1. If 'found' flag is set to 1, the value of count is incremented by 1 else the value of count is not changed. Subsequently if the Fin flag has already been encountered for this flow, it implies that this packet is the "ack" packet for the Fin packet, hence we delete

the node from the link list and make the linked list proper.

- **insert():** This function is called when the router encounters the first packet of any flow. In this function we create a new node of the type sock_details and insert the various values representing the flow into this node. The various values include source IP Address, Destination IP Address, Source Port, Destination Port. Also we set the proper values of limit, count, finish. Then we insert the node at the end of the linked list, which is pointed to by head_list.
- **finish():** This function is called if the Fin flag in the packet is set. In finish() function firstly we traverse the entire linked list of nodes until we get the node representing the flow of the packet or we reach the end of the list. If we find the flow in the linked list then we set the finish flag of the flow to 1 indicating that for this flow the Fin flag has been encountered. If we reach the end of the list and still do not find the node that indicates that it is the first packet of the flow to be encountered by the router, hence we create a new node using the

function insert() and append it to the existing linked list. Finally we set the finish flag to 1 for this newly created node.

- **process_drop():** This function is called only if ‘found’ flag is set to 1, i.e. if the search string is found in the TCP Data. In this function we check if we have check if the limit has been reached for the flow that the current packet belongs to. This is done by checking if (count > limit). limit indicates the number of packets containing the search string that need to be dropped for each flow. Hence if (count > limit), it indicates that for that flow, we have already dropped the required number of packets and any subsequent packets of that flow containing the search string shouldn’t be dropped. So if (count > limit) ‘drop’ flag is set to 0 else drop ‘flag’ is set to 1 indicating that this packet needs to be filtered i.e. dropped.

3.2.3 Useful Calculations

In this project I am working at the IP Layer and I have to access the data of TCP layer. Due to this layer mismatch, in order to prevent a layering violation, the kernel does not let us use the direct pointer to the TCP part using skb->h.th. In order to access TCP part of the sk_buff, we have to use the available pointer in IP layer and access the TCP Part by doing Pointer Arithmetic.

The structure of IP and TCP Header is as follows.

		Offset						
		a=sb->data	-->	Ver	Hlen	Service Type	Total Length	
		a + 4	-->	Identification			Flags	Fragment Offset
IP Header		a + 8	-->	Time to live		Protocol	Header Checksum	
		a + 12	-->	Source IP Address				
		a + 16	-->	Destination IP Address				
		a + 20	-->	Options if any				
		v	-->	Source Port			Destination Port	
		v + 4	-->	Sequence Number				
	TCP Header	v + 8	-->	Acknowlwdgement				
		v + 12	-->	Hlen		Flags	Window Size	
IP Data		v + 16	-->	Checksum			Urgent Pointer	
		v + 20	-->	Options if any				
	TCP Data	v1	-->	Data				

We can reach the start of the IP Header by using the available pointer sb->data. Further we can get the length of the IP header using the pointer (sb->nh.iph->ihl). Now in order to reach the start of TCP Header we have to take a pointer that points to the start of IP header and move it forward by the length of IP Header. To achieve this we use

v = (sb->data) + (sb->nh.iph->ihl);

where (sb->data)denotes the start of IP Header

and (sb->nh.iph->ihl)denotes the length of IP Header.

Further in order to access the TCP Data we have to access a pointer that point to the start of TCP Header (i.e. v) and increment this pointer by the length of TCP Header. To achieve this we use

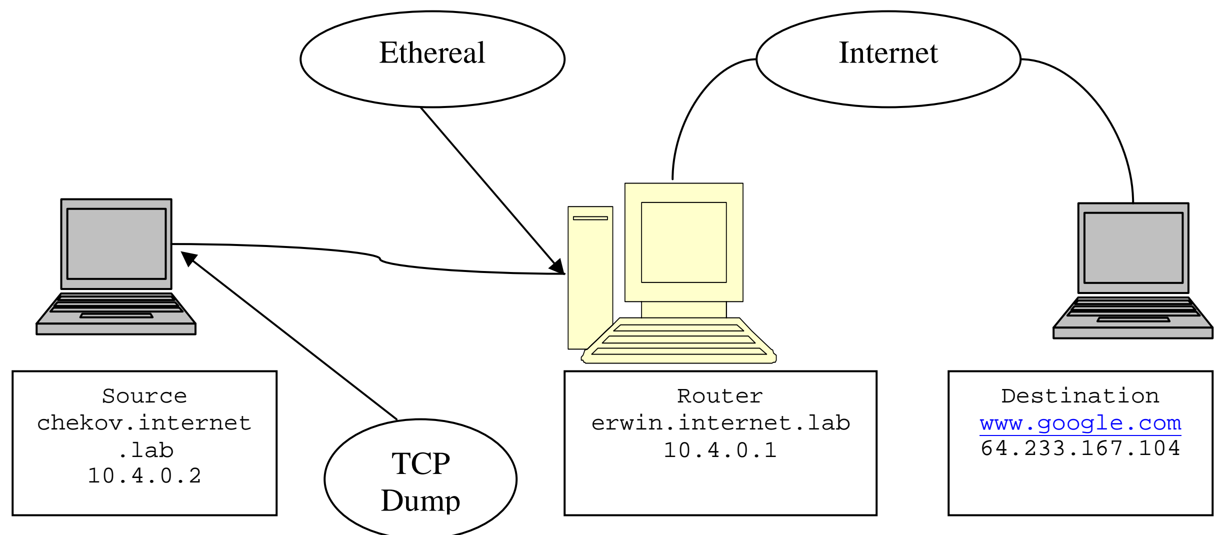
v1 = (v + (TCP_HLEN *4));

where v = (sb->data) + (sb->nh.iph->ihl); denotes the start of TCP Header

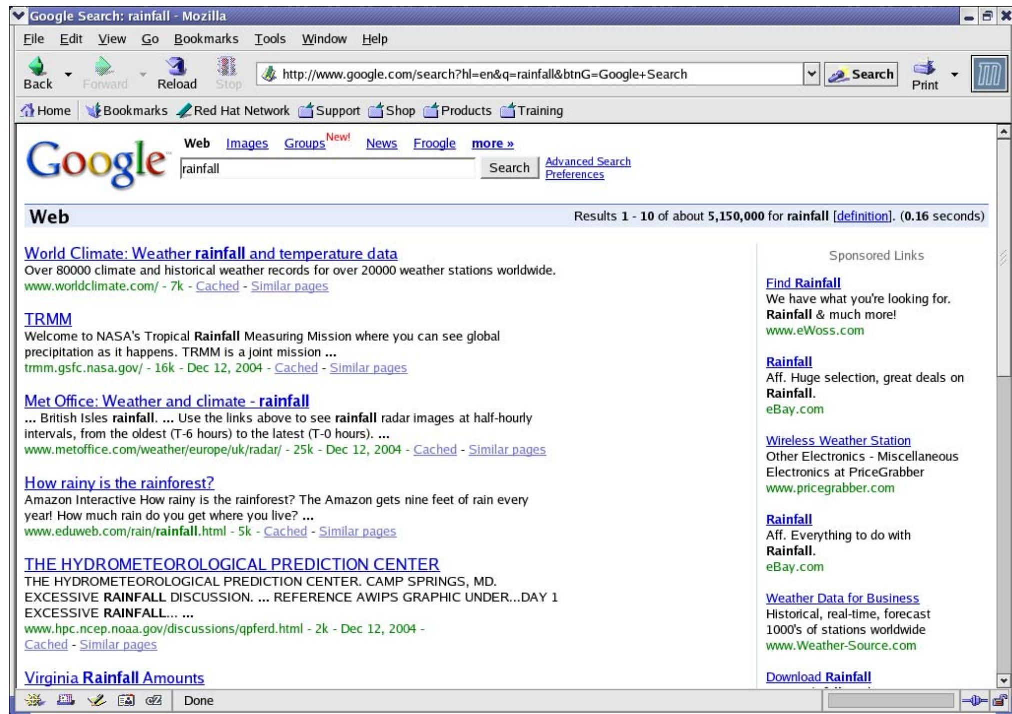
and TCP_HLEN = ((*(v + 12)) / 16); denotes the length of TCP Header because (v+12) points to the TCP Header length and hence *(v + 12) denotes the value of TCP Header length.

4. Tests and Results

The input for this test is in the form of an **http** request which goes from the Source (chekov.internet.lab, IP Address 10.4.0.2) to the Destination (www.google.com, IP Address 64.233.167.104) via the Router (erwin.internet.lab, IP Address 10.4.0.1). For this test we have used **"rainfall"** as the search string and we have set the limit to **3**. The requests are TCP/IP packets. The physical setup of the 3 hosts for this test is as follows.



Screenshot of the Input is shown below



The output generated by the module is stored in the log file. The output file generated by my module is as follows.

Please Note that the lines with “====” are not automatically generated by my module, they are comments added by me later on for a cleaner presentation in the Report/Documentation. Also the entire log file has not been pasted here due to the size constraints of the Report, I have shown only the packets that are important from our point of view and have excused the unimportant packets. e.g.:- For flow 1 and flow 2 the “Syn” packet is not shown and we jump directly to the first data packet which happens to be packet number 6 for flow 1.

```
========== Data Packet from Source ===============
Dec 14 14:40:41 erwin kernel: S_ADDR 200040a   D_ADDR 68a7e940
S_PORT d481   D_PORT 5000  Correct TCP_HLEN 5  The TCP Data is
Dec 14 14:40:41 erwin kernel: GET /sea
Dec 14 14:40:41 erwin kernel: ET /sear
Dec 14 14:40:41 erwin kernel: T /searc
Dec 14 14:40:41 erwin kernel:  /search
Dec 14 14:40:41 erwin kernel: /search?
Dec 14 14:40:41 erwin kernel: search?h
Dec 14 14:40:41 erwin kernel: earch?hl
Dec 14 14:40:41 erwin kernel: arch?hl=
```

Dec 14 14:40:41 erwin kernel: rch?hl=e

Dec 14 14:40:41 erwin kernel: ch?hl=en

Dec 14 14:40:41 erwin kernel: h?hl=en&

Dec 14 14:40:41 erwin kernel: ?hl=en&q

Dec 14 14:40:41 erwin kernel: hl=en&q=

Dec 14 14:40:41 erwin kernel: l=en&q=r

Dec 14 14:40:41 erwin kernel: =en&q=ra

Dec 14 14:40:41 erwin kernel: en&q=rai

Dec 14 14:40:41 erwin kernel: n&q=rain

Dec 14 14:40:41 erwin kernel: &q=rainf

Dec 14 14:40:41 erwin kernel: q=rainfa

Dec 14 14:40:41 erwin kernel: =rainfal

Dec 14 14:40:41 erwin kernel: rainfall

Dec 14 14:40:41 erwin kernel: The string is found

Dec 14 14:40:41 erwin kernel: Number of packets of this flow = 6

Dec 14 14:40:41 erwin kernel: --------------------------------

Dec 14 14:40:41 erwin kernel: For flow 1 number of packet = 6

Dec 14 14:40:41 erwin kernel: S_Addr = 200040a D_Addr = 68a7e940 S_port = d481 Dport = 5000

Dec 14 14:40:41 erwin kernel: Count 1 Limit 3 Finish 0

Dec 14 14:40:41 erwin kernel: --------------------------------

Dec 14 14:40:41 erwin kernel: --------------------------------

Dec 14 14:40:41 erwin kernel: For flow 2 number of packet = 4

Dec 14 14:40:41 erwin kernel: S_Addr = 68a7e940 D_Addr = 200040a S_port = 5000 Dport = d481

Dec 14 14:40:41 erwin kernel: Count 0 Limit 3 Finish 0

Dec 14 14:40:41 erwin kernel: --------------------------------

Dec 14 14:40:41 erwin kernel: This packet is dropped

========== End of Data Packet from Source ========

```
=============== Retransmission No 1 of Data Packet =========
Dec 14 14:40:41 erwin kernel: S_ADDR 200040a   D_ADDR 68a7e940
S_PORT d481   D_PORT 5000  Correct TCP_HLEN 5  The TCP Data is
Dec 14 14:40:41 erwin kernel: GET /sea
Dec 14 14:40:41 erwin kernel: ET /sear
....................................
Dec 14 14:40:41 erwin kernel: =rainfal
Dec 14 14:40:41 erwin kernel: rainfall
Dec 14 14:40:41 erwin kernel: The string is found
Dec 14 14:40:41 erwin kernel: Number of packets of this flow = 7
Dec 14 14:40:41 erwin kernel: --------------------------------
Dec 14 14:40:41 erwin kernel: For flow 1 number of packet = 7
Dec 14 14:40:41 erwin kernel: S_Addr = 200040a   D_Addr =
68a7e940   S_port = d481   Dport = 5000
Dec 14 14:40:41 erwin kernel: Count 2  Limit 3 Finish 0
Dec 14 14:40:41 erwin kernel: --------------------------------
Dec 14 14:40:41 erwin kernel: --------------------------------
Dec 14 14:40:41 erwin kernel: For flow 2 number of packet = 4
Dec 14 14:40:41 erwin kernel: S_Addr = 68a7e940   D_Addr =
200040a   S_port = 5000   Dport = d481
Dec 14 14:40:41 erwin kernel: Count 0  Limit 3 Finish 0
Dec 14 14:40:41 erwin kernel: --------------------------------
Dec 14 14:40:42 erwin kernel: This packet is dropped
======== End of  Retransmission No 1 of Data Packet  ========

======== Retransmission No 2 of Data Packet ==========
Dec 14 14:40:42 erwin kernel: S_ADDR 200040a   D_ADDR 68a7e940
S_PORT d481   D_PORT 5000  Correct TCP_HLEN 5  The TCP Data is
```

Dec 14 14:40:42 erwin kernel: GET /sea

Dec 14 14:40:42 erwin kernel: ET /sear

....

Dec 14 14:40:42 erwin kernel: =rainfal

Dec 14 14:40:42 erwin kernel: rainfall

Dec 14 14:40:42 erwin kernel: The string is found

Dec 14 14:40:42 erwin kernel: Number of packets of this flow = 8

Dec 14 14:40:42 erwin kernel: --------------------------------

Dec 14 14:40:42 erwin kernel: For flow 1 number of packet = 8

Dec 14 14:40:42 erwin kernel: S_Addr = 200040a D_Addr = 68a7e940 S_port = d481 Dport = 5000

Dec 14 14:40:42 erwin kernel: Count 3 Limit 3 Finish 0

Dec 14 14:40:42 erwin kernel: --------------------------------

Dec 14 14:40:42 erwin kernel: --------------------------------

Dec 14 14:40:42 erwin kernel: For flow 2 number of packet = 4

Dec 14 14:40:42 erwin kernel: S_Addr = 68a7e940 D_Addr = 200040a S_port = 5000 Dport = d481

Dec 14 14:40:42 erwin kernel: Count 0 Limit 3 Finish 0

Dec 14 14:40:42 erwin kernel: --------------------------------

Dec 14 14:40:42 erwin kernel: This packet is dropped

========== End of Retransmission No 2 of Data Packet =======

== Random data packet passing the router, not part of our study =====

Dec 14 14:40:42 erwin kernel: S_ADDR fa00030a D_ADDR 1000040a S_PORT fe03 D_PORT 302 Correct TCP_HLEN 10 The TCP Data is

Dec 14 14:40:42 erwin kernel: The string is NOT found

Dec 14 14:40:42 erwin kernel: --------------------------------

```
Dec 14 14:40:42 erwin kernel: For flow 1 number of packet = 8
Dec 14 14:40:42 erwin kernel: S_Addr = 200040a    D_Addr =
68a7e940   S_port = d481   Dport = 5000
Dec 14 14:40:42 erwin kernel: Count 3  Limit 3 Finish 0
Dec 14 14:40:42 erwin kernel: --------------------------------
Dec 14 14:40:42 erwin kernel: --------------------------------
Dec 14 14:40:42 erwin kernel: For flow 2 number of packet = 4
Dec 14 14:40:42 erwin kernel: S_Addr = 68a7e940    D_Addr =
200040a   S_port = 5000   Dport = d481
Dec 14 14:40:42 erwin kernel: Count 0  Limit 3 Finish 0
Dec 14 14:40:42 erwin kernel: --------------------------------
Dec 14 14:40:42 erwin kernel: --------------------------------
Dec 14 14:40:42 erwin kernel: For flow 3 number of packet = 1
Dec 14 14:40:42 erwin kernel: S_Addr = fa00030a    D_Addr =
1000040a   S_port = fe03   Dport = 302
Dec 14 14:40:42 erwin kernel: Count 0  Limit 3 Finish 0
Dec 14 14:40:42 erwin kernel: --------------------------------
==============================================================

======== Retransmission No 3 of Data Packet ===========
Dec 14 14:40:43 erwin kernel: S_ADDR 200040a   D_ADDR 68a7e940
S_PORT d481   D_PORT 5000  Correct TCP_HLEN 5  The TCP Data is
Dec 14 14:40:43 erwin kernel: GET /sea
Dec 14 14:40:43 erwin kernel: ET /sear
.....................................
Dec 14 14:40:43 erwin kernel: =rainfal
Dec 14 14:40:43 erwin kernel: rainfall
Dec 14 14:40:43 erwin kernel: The string is found
Dec 14 14:40:43 erwin kernel: Number of packets of this flow = 9
```

Dec 14 14:40:43 erwin kernel: --------------------------------

Dec 14 14:40:43 erwin kernel: For flow 1 number of packet = 9

Dec 14 14:40:43 erwin kernel: S_Addr = 200040a D_Addr = 68a7e940 S_port = d481 Dport = 5000

Dec 14 14:40:43 erwin kernel: Count 4 Limit 3 Finish 0

Dec 14 14:40:43 erwin kernel: --------------------------------

Dec 14 14:40:43 erwin kernel: --------------------------------

Dec 14 14:40:43 erwin kernel: For flow 2 number of packet = 4

Dec 14 14:40:43 erwin kernel: S_Addr = 68a7e940 D_Addr = 200040a S_port = 5000 Dport = d481

Dec 14 14:40:43 erwin kernel: Count 0 Limit 3 Finish 0

Dec 14 14:40:43 erwin kernel: --------------------------------

Dec 14 14:40:43 erwin kernel: --------------------------------

Dec 14 14:40:43 erwin kernel: For flow 3 number of packet = 1

Dec 14 14:40:43 erwin kernel: S_Addr = fa00030a D_Addr = 1000040a S_port = fe03 Dport = 302

Dec 14 14:40:43 erwin kernel: Count 0 Limit 3 Finish 0

Dec 14 14:40:43 erwin kernel: --------------------------------

======== End of Retransmission No 3 of Data Packet ========

========= Response Packet from Destination for Data Packet from Source ==========

Dec 14 14:40:43 erwin kernel: S_ADDR 68a7e940 D_ADDR 200040a S_PORT 5000 D_PORT d481 Correct TCP_HLEN 5 The TCP Data is

Dec 14 14:40:43 erwin kernel: The string is NOT found

Dec 14 14:40:43 erwin kernel: Number of packets of this flow = 5

Dec 14 14:40:43 erwin kernel: --------------------------------

Dec 14 14:40:43 erwin kernel: For flow 1 number of packet = 9

Dec 14 14:40:43 erwin kernel: S_Addr = 200040a D_Addr = 68a7e940 S_port = d481 Dport = 5000

Dec 14 14:40:43 erwin kernel: Count 4 Limit 3 Finish 0

Dec 14 14:40:43 erwin kernel: --------------------------------

Dec 14 14:40:43 erwin kernel: --------------------------------

Dec 14 14:40:43 erwin kernel: For flow 2 number of packet = 5

Dec 14 14:40:43 erwin kernel: S_Addr = 68a7e940 D_Addr = 200040a S_port = 5000 Dport = d481

Dec 14 14:40:43 erwin kernel: Count 0 Limit 3 Finish 0

Dec 14 14:40:43 erwin kernel: --------------------------------

Dec 14 14:40:43 erwin kernel: --------------------------------

Dec 14 14:40:43 erwin kernel: For flow 3 number of packet = 1

Dec 14 14:40:43 erwin kernel: S_Addr = fa00030a D_Addr = 1000040a S_port = fe03 Dport = 302

Dec 14 14:40:43 erwin kernel: Count 0 Limit 3 Finish 0

Dec 14 14:40:43 erwin kernel: --------------------------------

==

Screen shot of verification by Ethereal

<capture> - Ethereal

File Edit Capture Display Tools Help

No.	Time	Source	Destination	Protocol	Info
1	0.000000	10.4.0.2	64.233.167.104	HTTP	GET /search?hl=en&q=rainfall&btnG=Google+Search HTTP/1.1
2	0.207157	10.4.0.2	64.233.167.104	HTTP	GET /search?hl=en&q=rainfall&btnG=Google+Search HTTP/1.1
3	0.627646	10.4.0.2	64.233.167.104	HTTP	GET /search?hl=en&q=rainfall&btnG=Google+Search HTTP/1.1
4	0.806044	Intel_d3:cb:73	Broadcast	ARP	Who has 10.4.0.16? Tell 10.4.0.1
5	1.466648	10.4.0.2	64.233.167.104	HTTP	GET /search?hl=en&q=rainfall&btnG=Google+Search HTTP/1.1
6	1.474289	64.233.167.104	10.4.0.2	TCP	http > 33236 [ACK] Seq=1117907919 Ack=2916189297 Win=5840 Len=0
7	1.534270	64.233.167.104	10.4.0.2	HTTP	HTTP/1.1 200 OK
8	1.535103	10.4.0.2	64.233.167.104	TCP	33236 > http [ACK] Seq=2916189297 Ack=1117908112 Win=9432 Len=0
9	1.539713	64.233.167.104	10.4.0.2	HTTP	Continuation
10	1.540056	64.233.167.104	10.4.0.2	HTTP	Continuation
11	1.540212	10.4.0.2	64.233.167.104	TCP	33236 > http [ACK] Seq=2916189297 Ack=1117909492 Win=12420 Len=0
12	1.540214	10.4.0.2	64.233.167.104	TCP	33236 > http [ACK] Seq=2916189297 Ack=1117909629 Win=12420 Len=0
13	1.758972	64.233.167.104	10.4.0.2	HTTP	Continuation
14	1.761773	64.233.167.104	10.4.0.2	HTTP	Continuation
15	1.763644	64.233.167.104	10.4.0.2	HTTP	Continuation
16	1.759992	10.4.0.2	64.233.167.104	TCP	33236 > http [ACK] Seq=2916189297 Ack=1117911009 Win=15180 Len=0
17	1.762787	10.4.0.2	64.233.167.104	TCP	33236 > http [ACK] Seq=2916189297 Ack=1117912389 Win=17940 Len=0

⊞ Frame 5 (666 bytes on wire, 666 bytes captured)
⊞ Ethernet II, Src: 00:02:b3:d3:d2:a0, Dst: 00:02:b3:d3:cb:73
⊞ Internet Protocol, Src Addr: 10.4.0.2 (10.4.0.2), Dst Addr: 64.233.167.104 (64.233.167.104)
⊞ Transmission Control Protocol, Src Port: 33236 (33236), Dst Port: http (80), Seq: 2916188685, Ack: 1117907919, Len: 612
⊞ Hypertext Transfer Protocol

```
0000  00 02 b3 d3 cb 73 00 02  b3 d3 d2 a0 08 00 45 00   .....s.. ......E.
0010  02 8c 32 89 40 00 40 06  13 8c 0a 04 00 02 40 e9   ..2.@.@. ......@.
0020  a7 68 81 d4 00 50 ad d1  82 0d 42 a1 eb cf 50 18   .h...P.. ..B...P.
0030  24 d8 b9 45 00 00 47 45  54 20 2f 73 65 61 72 63   $..E..GE T /searc
0040  68 3f 68 6c 3d 65 6e 26  71 3d 72 61 69 6e 66 61   h?hl=en& q=rainfa
0050  6c 6c 26 62 74 6e 47 3d  47 6f 6f 67 6c 65 2b 53   ll&btnG= Google+S
0060  65 61 72 63 68 20 48 54  54 50 2f 31 2e 31 0d 0a   earch HT TP/1.1..
0070  48 6f 73 74 3a 20 77 77  77 2e 67 6f 6f 67 6c 65   Host: ww w.google
0080  2e 63 6f 6d 0d 0a 55 73  65 72 2d 41 67 65 6e 74   .com..Us er-Agent
0090  3a 20 4d 6f 7a 69 6c 6c  61 2f 35 2e 30 20 28 58   : Mozill a/5.0 (X
00a0  31 31 3b 20 55 3b 20 4c  69 6e 75 78 20 69 36 38   11; U; L inux i68
00b0  36 3b 20 65 6e 2d 55 53  3b 20 72 76 3a 31 2e 32   6; en-US ; rv:1.2
00c0  2e 31 29 20 47 65 63 6b  6f 2f 32 30 30 33 30 32   .1) Geck o/200302
00d0  32 35 0d 0a 41 63 63 65  70 74 3a 20 74 65 78 74   25..Acce pt: text
00e0  2f 78 6d 6c 2c 61 70 70  6c 69 63 61 74 69 6f 6e   /xml,app lication
00f0  2f 78 6d 6c 2c 61 70 70  6c 69 63 61 74 69 6f 6e   /xml,app lication
```

Filter: Reset Apply File: <capture> Drops: 0

Verification of Tests using TCP Dump

14:38:56.455455 chekov.33236 > 64.233.167.104.http: P 2916188685:2916189297(612) ack 1117907919 win 9432 (DF)

14:38:56.663365 chekov.33236 > 64.233.167.104.http: P 0:612(612) ack 1 win 9432 (DF)

14:38:57.083355 chekov.33236 > 64.233.167.104.http: P 0:612(612) ack 1 win 9432 (DF)

14:38:57.262706 arp who-has 10.4.0.16 tell 10.4.0.1

14:38:57.923357 chekov.33236 > 64.233.167.104.http: P 0:612(612) ack 1 win 9432 (DF)

14:38:57.931212 64.233.167.104.http > chekov.33236: . ack 612 win 5840

14:38:57.991917 64.233.167.104.http > chekov.33236: P 1:194(193) ack 612 win 5840

14:38:57.991939 chekov.33236 > 64.233.167.104.http: . ack 194 win 9432 (DF)

14:38:57.997021 64.233.167.104.http > chekov.33236: . 194:1574(1380) ack 612 win 5840

14:38:57.997045 chekov.33236 > 64.233.167.104.http: . ack 1574 win 12420 (DF)

14:38:57.997023 64.233.167.104.http > chekov.33236: P 1574:1711(137) ack 612 win 5840

14:38:57.997052 chekov.33236 > 64.233.167.104.http: . ack 1711 win 12420 (DF)

14:38:58.216899 64.233.167.104.http > chekov.33236: . 1711:3091(1380) ack 612 win 5840

14:38:58.216922 chekov.33236 > 64.233.167.104.http: . ack 3091 win 15180 (DF)

14:38:58.219694 64.233.167.104.http > chekov.33236: . 3091:4471(1380) ack 612 win 5840

14:38:58.219717 chekov.33236 > 64.233.167.104.http: . ack 4471 win 17940 (DF)

14:38:58.220815 64.233.167.104.http > chekov.33236: P 4471:5374(903) ack 612 win 5840

14:38:58.220830 chekov.33236 > 64.233.167.104.http: . ack 5374 win 20700 (DF)

14:38:58.232981 64.233.167.104.http > chekov.33236: FP 5374:5535(161) ack 612 win 5840

14:38:58.259141 arp who-has 10.4.0.16 tell 10.4.0.1

14:38:58.261201 chekov.33236 > 64.233.167.104.http: F 612:612(0) ack 5536 win 20700 (DF)

14:38:58.264319 64.233.167.104.http > chekov.33236: . ack 613 win 5840

14:38:59.259032 arp who-has 10.4.0.16 tell 10.4.0.1

14:39:00.052594 CDP v1, ttl=180s DevID 'Lab Switch 1(000883-432e80)' Addr (1): IPv4 127.0.0.1 PortID '6' CAP 0x08 Version: (suppressed) Platform: 'HP 2524'

Results: For this test, from the output of the module written in the logs, we see that flow 1 represents the TCP request flow from the Source chekov (i.e. 10.4.0.2) to the destination google.com (i.e. 64.233.167.104). Flow 2 represents the TCP response flow from the destination google.com (i.e. 64.233.167.104) to the Source chekov (i.e. 10.4.0.2). From the logs we can also see that at the start for flow 1 the count = 0 and the limit = 3.

When the **first data packet** is sent for flow 1, at **14:38:56.455455**, it caught by the search function of the module and the value of count is incremented by 1 for this flow and hence count becomes 1. Further we see that (count > limit) is not true, hence this packet is dropped.

The **first retransmission** of this packet is sent at **14:38:56.663365**. For this packet too, the search string (rainfall) in the data packet is caught by the search function of the module and the value of count is incremented by 1 for this flow and hence count becomes 2. Further we see that (count > limit) is not true, hence this packet also is dropped.

The **second retransmission** of this packet is sent at **14:38:57.083355**. For this packet too, the

search string (rainfall) in the data packet is caught by the search function of the module and the value of count is incremented by 1 for this flow and hence count becomes 3. Further we see that (count > limit) is not true, hence this packet also is dropped.

The **third retransmission** of this packet is sent at **14:38:57.923357**. For this packet too, the search string (rainfall) in the data packet is caught by the search function of the module and the value of count is incremented by 1 for this flow and hence count becomes 4. Further we see that (count > limit) is true, hence this packet is not dropped.

The above data can be tabulated as shown below

For Flow 1	String Present	Count	Limit	Drop Decision Count>limit
Data Packet	Yes	1	3	Drop
Retransmission 1	Yes	2	3	Drop
Retransmission 2	Yes	3	3	Drop
Retransmission 3	Yes	4	3	Do Not Drop

The analysis of this data gives us the following results with respect to TCP behavior.

For Flow 1	Time	Time Interval
Data Packet	14:38:56.455455	
Retransmission 1	14:38:56.663365	0.20791
Retransmission 1	14:38:57.083355	0.41999
Retransmission 1	14:38:57.923357	0.840002

We can clearly see that the First retransmission is done 207 m secs after the transmission of the original Data Packet. The Second retransmission is done 420 m secs after sending out the First retransmission. We can see that the wait time of TCP has almost doubled. Further we see that Third retransmission is done 840 m secs after sending out the Second retransmission. We can see that the wait time of TCP is almost twice of it’s previous wait time. Thus we can confirm TCP retransmission behaviors with the outputs of this software.

5. Applications

- This software can be used as a limiting firewall for institutions like universities and companies. e.g.:- Most of the commercially available firewalls do filtering on the basis of the URL, i.e. if an organization has a firewall to block employees from accessing shopping sites, it usually has a static list having all the blocked sites. One disadvantage of this method is that this static list needs to updated frequently also at any point of time there is a possibility that we do not cover all the sites needed to be blocked. Some of the firewalls dynamically check the URL for blocked content. i.e. if anyone types http://www.onlineshopping.com . The firewall catches the pattern "shopping" in the URL and blocks access, but there is big loop hole in this method. i.e. we can still access this site through the same firewall all we need to do is get the ip address of the server using either nslookup or dig or host, i.e. nslookup http://www.onlineshopping.com , which gives us an address like 216.109.118.78. Now we can access

the site through the firewall using http://216.109.118.78 . This happens because other firewalls check only the URL and do not check the TCP Data part. Using my firewall all we need to do is set the search string to **shopping** and set a very high limit. If the users behind my firewall try to access the site using the URL http://www.onlineshopping.com , the packets are dropped at the router itself and the request doesn't reach the server itself. On the contrary if the user tries to access the site using the URL http://216.109.118.78 , the request would go through but the response packets will contain the pattern **shopping** and hence will be dropped at the router, never reaching user. Thus using this firewall we can successfully isolate a user from unauthorized access to sites or servers. Also this software does not have any overhead of maintaining lists or URLs.

- It can be used as a lightweight simulator. e.g.:- This software can be used to generate different types of TCP/IP patterns, which can be given as input to other network programs.

- Using this simulator we can study the behavior of TCP for different types of drop patterns. e.g.:- This program can generate various types of drop patterns, hence it can be used to study the behavior of TCP for different drop patterns. Using this we can verify many TCP concepts like slow start, flow management etc. This can be achieved ny changing the value of limit in the Firewall software.
- This project uses link list where memory is dynamically allocated and deallocated, hence its use is time independent. So this project can be used to develop time independent software for the kernel.
- This project can be used for studying as to how exactly a flow is created, maintained and finished; we can also see as to which format is the data actually sent. This can be done by checking the log files created by this software. The log files are located at **"/var/logs"** and the default log file is **"messages"**.

6. **Future Enhancements**

- Ability to search a pattern or regular expression rather than a strict string. Eg.xx.xx.xx.xx
- To do flow maintenance based on time i.e. using **"jiffies"** to insert timestamps to the nodes whenever they are accessed. In this method the dead flows will not be accessed for a longer time as compared to the flows alive, hence we can delete all the dead flows, which have older timestamps.
- To allow different keywords for different flows.
- Implementing functionality to allow user input of the search keyword.
- To use KVM Algorithm or Suffix trees to preprocess the string in order to improve the search time.

7. Bibliography

- Behrouz A. Forouzan, 2003, TCP/IP Protocol Suite, Mc Graw Hill.
- A Map of the Networking Code in Linux Kernel 2.4.20 by M. Rio et al. 31 March 2004.
- Hacking the Linux Kernel Network Stack http://www.phrack.org/show.php?p=61&a=13
- The "Networking" code in Linux, Teunis J. Ott and Rahul Jain July 29, 2004
- D0 Code – comprehensively cross – referenced and searchable code
  http://www-d0.fnal.gov/D0Code/source/
- The journey of a packet through the linux 2.4 network stack - Harald Welte
  http://gnumonks.org/ftp/pub/doc/packet-journey-2.4.html
- The netfilter framework in Linux 2.4 - Harald Welt
  http://gnumonks.org/papers/netfilter-lk2000/presentation.html
- Overview of Routing and Packet Filter Interactions
  http://linux-ip.net/html/adv-overview.html

- The Packet handling in default Linux kernel 2.4 has been taken from http://open-source.arkoon.net/kernel/kernel_net.png