To Prevent IP Source Address Spoofing

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

Ankit Katyal
SID: 212-47-226

APPROVALS

Proposal Number: ____________________________________

Approved by: ____

(Dr. Teunis J. Ott)

Date Submitted: _____________________________________

I hereby affirm that I have followed the directions as published in the program Web-page at

http://cs.njit.edu/~mscs/

And I confirm, that this proposal is my own personal work and that all material other than my own is properly referenced.

Student’s Name: ______________________________

Student’s Signature: ______________________________

Date: ______________________________

Acknowledgements

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

I must also acknowledge the invaluable help provided by Mr. Rahul Jain Teaching Assistant to Dr. Ott during the project

Sincerely,

(Ankit Katyal)

Certificate

This is to certify that the project report titled “To Prevent IP Source Address Spoofing” is a bonafide record of the project work carried out under my supervision by Mr. Ankit Katyal, Student ID: 212-47-226 at the Department of Computer Science, New Jersey Institute of Technology, from September-2004 to December-2004, for partial fulfillment of requirements towards his Master’s degree.

Dr. Teunis J. Ott
Department of Computer Science
New Jersey Institute of Technology

Abstract

Source Address Spoofing has become a problem due to the increased number of denial of service attacks being carried out by using means of opening hundreds even thousands of half open TCP connection's or more commonly known as SYN flooding. This project attempts to prevent this by modifying the way the router forwards packets i.e. instead of only checking the destination address for deciding on the forwarding route the source address is also verified to be correct. This can be achieved my modifying the free and open source operating system Linux

This project deals with modifying the behavior of the Linux Kernel by adding a function at such a point in the Linux Kernel where we know packets of a certain type would always pass which in our case are the packets that have to be forwarded and are not meant for the machine itself.

To do this we maintain a link list comprising of a structure which has the source address, entry interface and whether the address is valid or not as its components. To know whether the source address is valid or not whenever a packet of a new flow arrives we first check whether the source address is already known to us and is valid or invalid, otherwise we temporarily declare the source address to be invalid and try to validate it using mechanisms which are explained later.

Table of Contents

1 Introduction and Background

1.1 Statement of Problem Area

Source Spoofing is the activity where a remote system sends out false or misleading source IP addresses most commonly to facilitate a denial of service attack This works on the theory that a machine whenever it receives a new connection request from a new IP address allocates resources for the new connection that is called half open as the SYN-ACK packet is sent back and an ACK packet is expected. If there are thousands of such requests at the same time from different IP addresses then the machine would try to allocate resources for all these half open connections and can consequently crash taking all its offered services down from the internet with it. Thus a malicious machine if it sends out thousands of such requests all targeted at the same machine but with different source addresses would then be able to crash the victim machine. This is essentially done to disable the victim machine or even the entire network. Depending on the nature of enterprise, it can even disable the entire organization and prevent access by genuine users.

In this project the system is so designed and implemented that the routers filter incoming packets, and determine based on the combination of source and destination address, whether the packet has come from a legitimate source address and the proper network interface. If the above two conditions are not met then spoofing is assumed and the packet is dropped.

This paper first explains the fundamental concepts of Modules, the Net Filter Architecture and routing which are used in the project in section 2.In section 3 and 4 the functionality of the project and performance parameters are delimited. Section 5 describes how the data flows through the system and what sort of data structures would be needed. System Interaction with how data would enter the system and how would output of the system would be shown to the user along with the data structures we need to define is discussed in Section 6. Section 7 covers the definition of the modules we require their operation and algorithms for the various modules and Section 8 describes the testing phase which covers the test cases in which we had to write an function that does spoofing of Source Addresses. Section 9 concludes the project with the problems we encountered during the project and how it could be extended.

1.2 Previous Work

To prevent address spoofing thus changes have to be made as to how the packet is handled in the kernel of the Operating System of the router. For this purpose the Operating System that was chosen was Linux as it is open source and the source code is available which can be used to incorporate changes to prevent source address spoofing

This project is based on the functionality provided by the Linux 2.4 kernel code as it is currently the most stable version of the kernel available. It also draws upon the information provided by the documentation of the code most significantly IPV4 routing and various RFC’s that are in context of the project.

1.3 Background

1.3.1 Loadable Linux Kernel Modules

To prevent address spoofing the changes have to be made directly in the Linux kernel which can be very cumbersome as the kernel has to be rebuilt every time there is a change which is very time consuming. To solve this problem we will use Linux Kernel Modules which is basically a chunk of code you add to the Linux Kernel while it is running thus giving it the name loadable kernel module. They basically form an extension of the Linux Kernel and run in the kernel space of the Operating System and should not be confused with user space programs that do not have kernel privileges

Loadable Linux Kernel Modules are thus ideal for writing changes as to how the packet is handled in the network stack as they have the advantage that kernel does not have to rebuilt as often. This saves time and minimizes the possibility of introducing an error in rebuilding and reinstalling the base kernel. Another advantage is that Linux Kernel Modules can save memory, as they have to be loaded when they are to be used as opposed to the base kernel whose parts stay loaded all the time in real storage, not just virtual storage.

Linux Kernel Modules are much faster to maintain and debug. What would require a full reboot to do with the program built into the kernel, can be achieved with a few quick commands with Linux Kernel Modules. Different parameters can be used or even the code can be changed repeatedly in rapid succession, without waiting for a boot. Linux Kernel Modules are also not slower than base kernel modules. Calling either one is simply a branch to the memory location where it resides [6].

1.3.2 The Net Filter Facility

The Linux net filter is a framework in the kernel that allows modules to observe and modify packets as they pass through the protocol stack. Kernel services or modules can register custom hooks by both protocol family and by the point in packet processing at which the filter is to be invoked. The facility is currently available for IPv4, IPv6 and DECnet but could be extended to other protocol families. Each protocol family can provide several processing points in the stack where a packet of that protocol can be passed to a filter. These points are referred to as hook points or hook types. Hence, when registering a custom hook, the protocol family and the protocol specific hook type must be specified. [7]

1.4 Brief Project Description

This project deals with modifying the behavior of the Linux Kernel When modifying the way certain packets are handled in the Linux Networking Code, we could do the following:

- Find a function that handles all packets we are interested in and then write a new function call inside that function, for a new function or set of functions that is then called to make the change effective, we now have to recompile the kernel.(This is a lengthy process).
- ``Netfilter Hooks'' can be thought of as places in the code that are arranged to have most or all packets of some specific type pass by, and are specifically designed to make it easy to add code that ``intercepts'' all packets passing through that point.

In fact, Netfilter Hooks have been designed not only to do the above, but also to make it easy to attach the new code to that location, in the form of a ``Loadable Kernel Module'' (by ``binding'' the module to a specific hook). A ``Loadable Kernel Module'' is an addition to the kernel (it runs in kernel space and has kernel privileges) that can be activated (``installed'') and de-activated (``uninstalled'') without having to recompile the kernel or even rebooting in the Linux Kernel where we know packets of a certain type would always pass which in our case are the packets that have to be forwarded and are not meant for the machine itself.

To prevent source address spoofing we maintain a link list comprising of a structure which has the source address, entry interface and whether the address is valid or not as its components. To know whether the source address is valid or not whenever a packet of a new flow arrives we first check whether the source address is already known to us and is valid or invalid, otherwise we temporarily declare the source address to be invalid and try to validate it using mechanisms which are explained later in section 3 of the report.

To test our new functionality in the router we had to send packets with spoofed source addresses ourselves. This however was harder than expected as the new address invalidated the IP and TCP checksums. The problem was solved by computing the new checksum and we are now able to send out spoofed source address packets which are not allowed to go through by the router.

1.5 Objective of Project

The purpose of the project is to prevent source address spoofing of IP addresses; this is helpful in eliminating malicious attacks on the internet by using spoofed IP addresses. IP Source Address Spoofing is mainly used in denial of service attacks. This works on the theory that a machine whenever it receives a new connection request from a new IP address allocates resources for the new connection that is called half open as the SYN-ACK packet is sent back and an ACK packet is expected. If there are thousands of such requests at the same time from different IP addresses then the machine would try to allocate resources for all these half open connections and can consequently crash taking all its offered services down from the internet with it. Thus a malicious machine if it sends out thousands of such requests all targeted at the same machine but with different source addresses would then be able to crash the victim machine. This is essentially done to disable the victim machine or even the entire network. Depending on the nature of enterprise, it can even disable the entire organization and prevent access by genuine users. There can be other reasons to spoof source address where the perpetrator of a malicious attack could change the source address making it harder for the attack to be traced back to the original source. Thus if the spoofing of IP address is prevented certain kinds of malicious attacks on the internet can be prevented

2 Background of the Linux Routing Functionality

This section deals with the background needed to understand the modifications done to the Linux Kernel and how it is achieved. This is meant for the audiences who have very basic knowledge of the Linux Kernel others can go directly to Section 3

2.1 Loadable Linux Kernel Modules

Linux Kernel Modules which is basically a chunk of code you add to the Linux Kernel while it is running thus giving it the name loadable kernel module. They basically form an extension of the Linux Kernel and run in the kernel space of the Operating System and should not be confused with user space programs that do not have kernel privileges

Loadable Linux Kernel Modules(LKM) are thus ideal for writing changes as to how the packet is handled in the network stack as they have the advantage that kernel does not have to rebuilt as often. This saves time and minimizes the possibility of introducing an error in rebuilding and reinstalling the base kernel. Another advantage is that Linux Kernel Modules can save memory, as they have to be loaded when they are to be used as opposed to the base kernel whose parts stay loaded all the time in real storage, not just virtual storage.

Linux Kernel Modules are much faster to maintain and debug. What would require a full reboot to do with the program built into the kernel, can be achieved with a few quick commands with Linux Kernel Modules. Different parameters can be used or even the code can be changed repeatedly in rapid succession, without waiting for a boot. Linux Kernel Modules are also not slower than base kernel modules. Calling either one is simply a branch to the memory location where it resides [6].

The basic structure of a LKM has been given below

```
int init_module()
{

   <Code>



   return 0;
}






void cleanup_module ()
{

}
```

The LKM is basically a C program but has no main function and has to be declared a module explicitly by using the statement

```
#define MODULE
```

The module has to be compiled by a special command which is

```
gcc -I/usr/src/linux/include -O2 -D__KERNEL__ -Wall   <module_name>.o: <module_name>.c
```

This command compiles the specified module and makes an output file for the module <module_name>.o

As the module has no main () function, the starting interface of a module is the init_module function which is executed whenever the module is first loaded into the kernel memory

This is achieved by giving the following command

```
/sbin/insmod <module_name>.o
```

The cleanup_module function is called whenever the module is unloaded from the kernel memory

This is achieved by the following command

```
/sbin/rmmod <module_name>
```

There can however be problems with the loading of the module if the kernel version defined in the */usr/src/linux/MAKEFILE* is different from the current kernel thus we have to change the kernel version in the *MAKEFILE* and call the make command from */usr/src/linux/* thus ensuring our version matches and then recompile the module again.

2.2 SK_BUFF Structure

The buffers used by the kernel to manage network packets are referred to as sk_buff in Linux. The buffers are always have two parts i.e. a fixed size structure of type sk_buff and a dynamic area which could be fragmented and is large enough to hold the entire data of a single packet.

```
struct sk_buff {
/* These two members must be first. */
struct sk_buff * next; /* Next buffer in list */
struct sk_buff * prev; /* Previous buffer in list */

struct sk_buff_head * list; /* List we are on */
struct sock *sk; /* Socket we are owned by */
struct timeval stamp; /* Time we arrived */
struct net_device *dev; /* Device we arrived on/are leaving by */
```

The section below contains the definition of the pointers that belong to the transport, network, and link headers. They are declared as unions so that only a single word of storage is allocated for each layer's header pointer.

```

/* Transport layer header */
union
{
struct tcphdr *th;
struct udphdr *uh;
.
.
.
} h;

/* Network layer header */
union
{
struct iphdr *iph;
:
.
.
} nh;

/* Link layer header */
union
{
struct ethhdr *ethernet;
unsigned char *raw;
} mac;

```

```
struct dst_entry *dst;

/*
* This is the control buffer. It is free to use for every
* layer. Please put your private variables there. If you
* want to keep them across layers you have to skb_clone()
* first. This is owned by whoever has the skb queued ATM.
*/
char cb[48];

unsigned int len; /* Length of actual data */
unsigned int data_len;
unsigned int csum; /* Checksum */
unsigned char __unused, /* Dead field, */
cloned, /* head may be cloned (check refcnt to be sure). */
pkt_type, /* Packet class */
ip_summed; /* Driver fed us an IP checksum */
__u32 priority; /* Packet queueing prty */
atomic_t users; /* User count – see datagram.c,tcp.c */
unsigned short protocol; /* Packet protocol from driver. (ETH_P_IP etc) */
unsigned short security; /* Sec level of packet*/
unsigned int truesize; /* Buffer size */
```

These pointers all point into the variable size component of the buffer which actually contains the packet data. At allocation time head, data, and tail point to the start of the allocated packet data area and end points to the skb_shared_info structure which begins at next byte beyond the area available for packet data. A large collection of inline functions defined in include/linux/skbuff.h may be used in adjustment of data, tail, and len as headers are added or removed. [7]

```
unsigned char *head; /* Head of buffer */
unsigned char *data; /* Data head pointer */
unsigned char *tail; /* Tail pointer */
unsigned char *end; /* End pointer */
```

The destructor function is called when the last entity that held a pointer to the buffer frees the buffer.

```
void (*destructor)(struct sk_buff *);
/* Destruct function */
#ifdef CONFIG_NETFILTER
/* Can be used for communication between hooks. */
unsigned long nfmark;
/* Cache info */
__u32 nfcache;
/* Associated connection, if any */
struct nf_ct_info *nfct;
#endif /*CONFIG_NETFILTER*/
}
```

MAC Header definition

```
struct ethhdr
{
unsigned char h_dest[ETH_ALEN]; /* dest eth addr */
unsigned char h_source[ETH_ALEN]; /* src eth addr */
unsigned short h_proto; /* packet type*/
};
```

IP Header

```
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
__u8 ihl:4,
version:4;
#elif defined (__BIG_ENDIAN_BITFIELD)
__u8 version:4,
ihl:4;
#endif
__u8 tos;
__u16 tot_len;
__u16 id;
__u16 frag_off;
__u8 ttl;
__u8 protocol;
__u16 check;
__u32 saddr;
__u32 daddr;
};
```

2.3 The Net Filter Facility

As described in [7] The Linux net filter is a framework in the kernel that allows modules to observe and modify packets as they pass through the protocol stack. This means that the there exist certain points in the Linux code IPv4 layer where we are sure that packets of a certain type would always pass. Referring to Figure on page 15 we see that there exist five of these points where our code can be added, they are namely

- NF_IP_PRE_ROUTING - Every packet coming into this box would pass through here
- NF_IP_LOCAL_IN - Every packet destined for this box would pass through here
- NF_IP_FORWARD - If the packet is not for this and destined for another interface.
- NF_IP_LOCAL_OUT - Packets coming from a local process in the box itself.
- NF_IP_POST_ROUTING - Packets about to hit the wire.

Kernel services or modules which we intend to use can register custom hooks by both protocol family and by the point in packet processing i.e. the hook at which the filter is to be invoked. The facility is currently available for IPv4, IPv6 and DECnet but could be extended to other protocol families. When registering a custom hook, the protocol family and the protocol specific hook type must be specified.

If the reader does not want advanced knowledge of how hooks are implemented then the rest of the section can be skipped.

A statically allocated array of lists defined in *net/core/netfilter.c* holds all the hooks registered for each protocol and hook types. *NF_MAX_HOOKS,* the maximum types of hooks a protocol can support has been defined as 8 in *include/linux/netfilter.h*.

```
struct list_head nf_hooks[NPROTO][NF_MAX_HOOKS];
/* Largest hook number + 1 */
#define NF_MAX_HOOKS 8
```

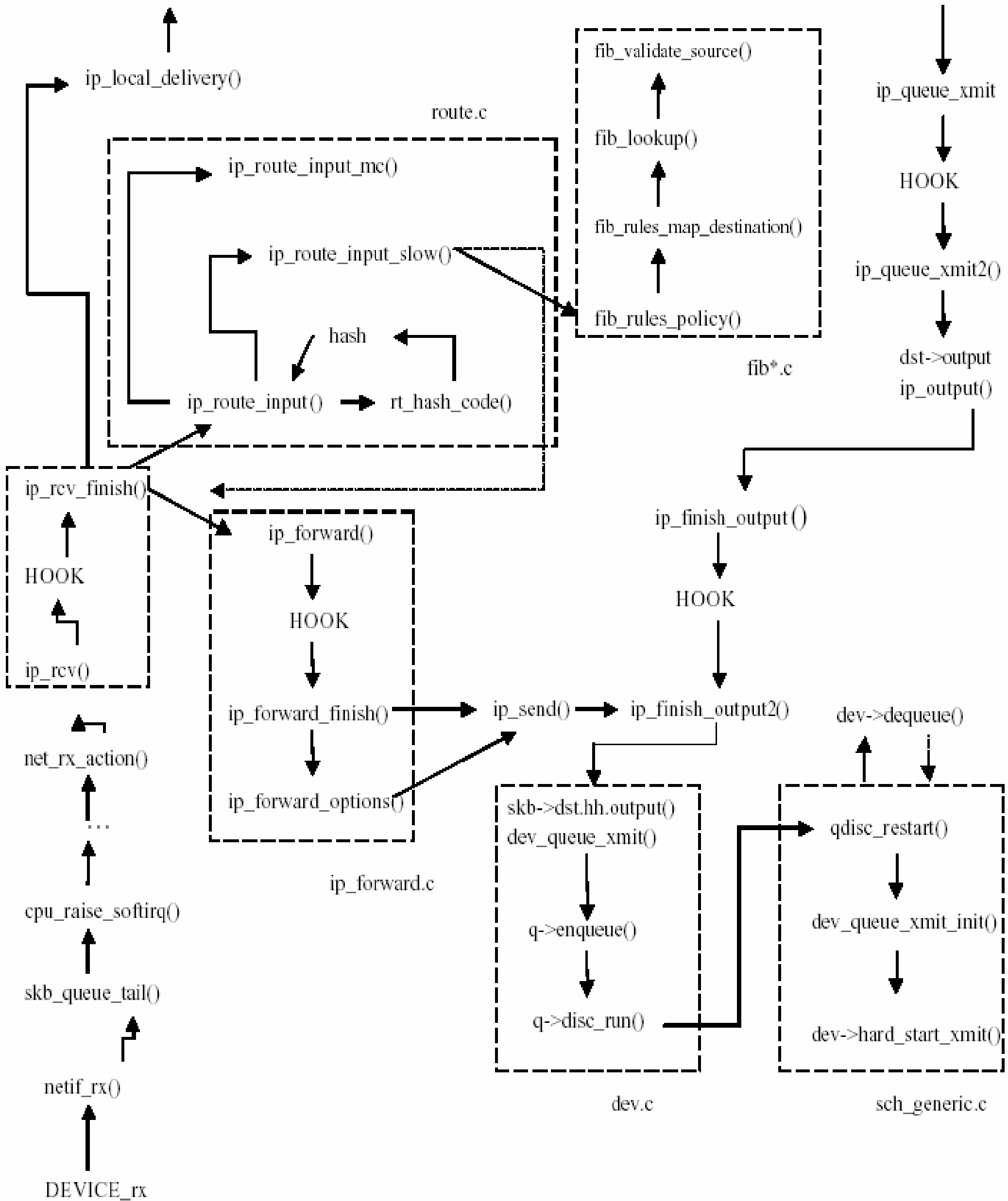


Diagram Depicting the Hooks and their position in the Code [8]

2.3.1 Defining a NetFilter hook

Each custom hook is defined using the following *nf_hook_ops* structure. This structure is passed to
the *nf_register_hook* function.

```
struct nf_hook_ops
{
struct list_head list;

/* User fills in from here down. */
nf_hookfn *hook;
int pf;
int hooknum;
/* Hooks are ordered in ascending priority. */
int priority;
};
```

Structure elements are used as follows:

- list: links all hooks of a common pm and hooknum into the *nf_hooks* array
- pf: protocol family (PF_INET i.e. IPV4 address family) of the filter.
- hooknum: the protocol specific hook type (i.e. NF_IP_FORWARD) identifier.
- priority: order of the hook in the list.
- hook: A pointer to the hook function.

Its prototype is as follows:

```
typedef unsigned int nf_hookfn(unsigned int hooknum,
struct sk_buff **skb,
const struct net_device *in,
const struct net_device *out,
int (*okfn)(struct sk_buff *));

```

Some standard priorities are shown below.

```
enum nf_ip_hook_priorities {
NF_IP_PRI_FIRST = INT_MIN,
NF_IP_PRI_CONNTRACK = -200,
NF_IP_PRI_MANGLE = -150,
NF_IP_PRI_NAT_DST = -100,
NF_IP_PRI_FILTER = 0,
NF_IP_PRI_NAT_SRC = 100,
```

```
NF_IP_PRI_LAST = INT_MAX,
};
```

The *nf_register_hook()* function defined in *net/core/netfilter.c* adds the *nf_hook_ops* structure that defines a custom hook to the appropriate list based on the protocol family and filter type.

Since the list is ordered by ascending priority values, invocation order is lowest numerical value first.

```
int nf_register_hook(struct nf_hops *reg)
{
struct list_head *i;

br_write_lock_bh(BR_NETPROTO_LOCK);
for (i = nf_hooks[reg->pf][reg->hooknum].next;
i != &nf_hooks[reg->pf][reg->hooknum];
i = i->next)
{if (reg->priority <((struct nf_hook_ops *)i)->priority)
break;
}
list_add(&reg->list, i->prev);
br_write_unlock_bh(BR_NETPROTO_LOCK);
return 0;
}
```

2.3.2 IP Packet Transmission through the Netfilter Layer

From *ip_build_xmit()* or *ip_build_xmit_slow()*, the IP packet is pushed to the device/netfilter layer using the *NF_HOOK* macro defined in *include/linux/netfilter.h*. Parameters passed include the output device to be used and the final output function to be invoked on successful verdict from all the hooks in the list. The hook type is *NF_IP_LOCAL_OUT.* The input device is set to NULL, since the packet originated on the local host.

```
err=NF_HOOK(PF_INET,NF_IP_LOCAL_OUT,skb,NULL,rt-> u.dst.dev ,output_maybe_reroute );
```

This macro translates to a call to the *nf_hook_slow()* function if the netfilter debug option is defined or if there are hooks/filters set for the specific protocol family and hook type. Otherwise it simply passes the *sk_buff* directly to the *ok* function.

```
/* This is gross, but inline doesn't cut it for avoiding the
function call in fast path: gcc doesn't inline (needs value tracking?). --RR */
#ifdef CONFIG_NETFILTER_DEBUG
#define NF_HOOK nf_hook_slow
#else
#define NF_HOOK(pf, hook, skb, indev, outdev, okfn) \
(list_empty(&nf_hooks[(pf)][(hook)]) \
? (okfn)(skb) \
: nf_hook_slow((pf), (hook), (skb),
(indev), (outdev), (okfn)))
#endif
```

When the net filter facility is enabled and the look list is non-empty, this macro invokes the *nf_hook_slow()* function. The *nf_hook_slow()* function is defined in net/core/netfilter.c, it's task is to invoke each hook in the specified list, and based on the verdict from the hooks, it either passes the packet to the *okfn* or drops the packet.

```
int nf_hook_slow(int pf, unsigned int hook,struct sk_buff *skb,
struct net_device *indev,
struct net_device *outdev,
int (*okfn)(struct sk_buff *))
{
struct list_head *elem;
unsigned int verdict;
int ret = 0;
```

For a non-linear *sk_buff* each fragment's size, offset and page address are stored in the *skb_frag_struct* array. If the skb is non-linear (i.e. *skb->data_len!=0)*, *skb_linearize()* is called to reorganized all the data into one linear buffer.

```
/* This stopgap cannot be removed until all the hooks are audited. */
if (skb_is_nonlinear(skb) && skb_linearize(skb, GFP_ATOMIC) != 0) {
kfree_skb(skb);
return -ENOMEM;
}
```

After ensuring the *sk_buff* is linear, *nf_hook_slow()* continues. The *ip_summed* field in the *sk_buff* was initialized to 0 (CHECKSUM_NONE) during creation. The objective of this code block is unclear. It should be remembered though that this *nf_hook_slow()* is called for both input and output processing.

```
if (skb->ip_summed == CHECKSUM_HW) {
if (outdev == NULL) {
skb->ip_summed = CHECKSUM_NONE;
} else {
skb_checksum_help(skb);
}
}

/* We may already have this, but read-locks nest anyway */
br_read_lock_bh(BR_NETPROTO_LOCK);

#ifdef CONFIG_NETFILTER_DEBUG
if (skb->nf_debug & (1 << hook)) {
printk("nf_hook: hook %i already set.\n",hook);
nf_dump_skb(pf, skb);
}
skb->nf_debug |= (1 << hook);
#endif

```

Here the function *nf_iterate()* is called to execute all the hooks defined for this protocol family and hook type.

```
elem = &nf_hooks[pf][hook];
verdict = nf_iterate(&nf_hooks[pf][hook],&skb, hook, indev, outdev, &elem, okfn);
```

On return to *nf_hook_slow()*, actions are based on the verdict. A verdict of NF_QUEUE for an IP packet this results in a series of function calls leading to the *ipq_enqueue()* function defined in *net/ipv4/netfilter/ip_queue.c*.

```
if (verdict == NF_QUEUE) {
NFDEBUG("nf_hook: Verdict = QUEUE.\n");
nf_queue(skb, elem, pf, hook, indev, outdev,okfn);
}
```

If NF_ACCEPT is the verdict from all hooks, the *output_maybe_reroute()* function which was passed into *nf_hook_slow()* as the *okfn()* is invoked with the *sk_buff* as the parameter. If the packet is to be dropped *kfree_skb()* is called.

```
switch (verdict) {
case NF_ACCEPT:
ret = okfn(skb);
break;
case NF_DROP:
kfree_skb(skb);
ret = -EPERM;
break;
}
br_read_unlock_bh(BR_NETPROTO_LOCK);
return ret;
}
```

2.3.3 Iterating through the hook chain

The *nf_iterate()* function is defined in net/core/netfilter.c

```
static unsigned int nf_iterate(struct list_head *head,
struct sk_buff **skb,
int hook,
const struct net_device *indev,
const struct net_device*outdev,
struct list_head **i,
int (*okfn)(struct sk_buff *))
{
```

For each hook called this loop is iterated once

```
for (*i = (*i)->next; *i != head; *i = (*i)->next) {
struct nf_hook_ops *elem =
(struct nf_hook_ops *)*i;
```

The value returned by the hook function determines the action taken by the switch statement. An immediate return, possibly aborting the send, is made if the value returned is NF_QUEUE, NF_STOLEN, or NF_DROP. If NF_REPEAT or NF_ACCEPT is returned the ‘for’ *loop* continues.

```
switch (elem->hook(hook, skb, indev, outdev,okfn))
```

```
{
case NF_QUEUE:
return NF_QUEUE;

case NF_STOLEN:
return NF_STOLEN;

case NF_DROP:
return NF_DROP;

case NF_REPEAT:
*i = (*i)->prev;
break;

#ifdef CONFIG_NETFILTER_DEBUG
case NF_ACCEPT:
break;

default:
NFDEBUG("Evil return from %p(%u).\n",
elem->hook, hook);
#endif
}
}
```

If all the hook functions return NF_ACCEPT, then NF_ACCEPT is returned to *nf_hook_slow*.

```
return NF_ACCEPT;
}
```

The output_maybe_reroute() function

If the packet is accepted for transmission by *nf_hook_slow,* the *okfn(), output_maybe_reroute(),* defined in *net/ipv4/ip_output.c* is called. It simply passes control to the *output* function associated with the *dst* structure that is presently bound to the *sk_buff.*

```
static inline int
output_maybe_reroute(struct sk_buff *skb)
{
return skb->dst->output(skb);
}
```

The pointer *skb->dst* refers to the route cache element associated with this packet's source and destination. In *ip_route_output_slow(), rt->u.dst->output* was set to *ip_output()* which is defined in *net/ipv4/ip_output.c.*

```
int ip_output(struct sk_buff *skb)
{
#ifdef CONFIG_IP_ROUTE_NAT
struct rtable *rt = (struct rtable*)skb->dst;
#endif

IP_INC_STATS(IpOutRequests);

#ifdef CONFIG_IP_ROUTE_NAT
if (rt->rt_flags&RTCF_NAT)
ip_do_nat(skb);
#endif

return ip_finish_output(skb);
}
```

The ip_finish_output() function

The *ip_finish_output()* function sets *skb->dev* to the device associated with the route's associated output device structure and the protocol type to *ETH_P_IP*. This indicates that the value 0x8000 must represent an IP packet even if the output device is *not* an ethernet device.

```
__inline__ int ip_finish_output(struct sk_buff *skb)
{
struct net_device *dev = skb->dst->dev;

skb->dev = dev;
skb->protocol = __constant_htons(ETH_P_IP);
```

Next, the NF_HOOK macro is again invoked. This macro expands to *nf_hook_slow()* and invokes all the net filters defined for PF_INET at the NF_IP_POST_ROUTING level. If the verdict from all filters is NF_ACCEPT, the *okfn(), ip_finish_output2()* is called as before.

```

return NF_HOOK(PF_INET, NF_IP_POST_ROUTING,
skb, NULL, dev, ip_finish_output2);
}
```

The ip_finish_output2() function

The *ip_finish_output2()* function is defined in net/ipv4/ip_output.c .

```
static inline int ip_finish_output2(struct sk_buff *skb)
{
struct dst_entry *dst = skb->dst;
struct hh_cache *hh = dst->hh;

#ifdef CONFIG_NETFILTER_DEBUG
nf_debug_ip_finish_output2(skb);
#endif /*CONFIG_NETFILTER_DEBUG*/

```

There are two mechanisms by which calls to the link layer may be made. If the *dst_entry* has an *hh_cache* pointer then the *hh_cache* entry must contain both the hardware header itself and a pointer to an output function at the device / link layer. The *output* function is always set to *dev_queue_xmit().* If there is no *hh* pointer but there is a *neighbor* pointer, then the neighbor structure must have an output function pointer. The output function of the neighbour structure is set to *neigh_resolve_output()* if the network device needs a hardware header. Otherwise (for a loopback, point to point, or virtual device) it set to invoke *dev_queue_xmit()* by the *arp_constructor()* function that is called when each neighbor structure is created.

```
if (hh) {
read_lock_bh(&hh->hh_lock);
memcpy(skb->data - 16, hh->hh_data, 16);
read_unlock_bh(&hh->hh_lock);
skb_push(skb, hh->hh_len);
return hh->hh_output(skb);
} else if (dst->neighbour)
return dst->neighbour->output(skb);

```

If there is no hardware header structure and no neighbor structure available, then there is no way to send the packet and it must be dropped. The net_ratelimit() function is used to limit the number of printk's generated to not more than 1 every 5 seconds to avoid flooding the syslog in case something is badly amiss in the network setup.

```
if (net_ratelimit())
```

```
printk(KERN_DEBUG "ip_finish_output2:
No header cache and no neighbour!\n");
kfree_skb(skb);
return -EINVAL;
}
```

2.4 Connection Tracking

Connection tracking is done to let the net filter framework know the state of a specific connection.

Connection tracking is done either in the NF_IP_PREROUTING hook or the NF_IP_LOCAL_OUT hook for the packets generated on the machine itself. It is basically implemented to manage individual connections and it serves to allocate IP packets as incoming, outgoing or forwarded to already existing connections. The connections are maintained mainly for the packets belonging to the TCP protocol but the UDP packets are also taken care of.

Connection tracking has been implemented as a separate module and has to be loaded for it to work with the project. The commands to load the connection tracking module are given below.

echo -en "ip_conntrack, " /sbin/insmod ip_conntrack

echo -en "ip_conntrack_ftp, " /sbin/insmod ip_conntrack_ftp

echo -en "ip_conntrack_irc, " /sbin/insmod ip_conntrack_irc

Connection tracking has four states

- NEW
- ESTABLISHED
- RELATED
- INVALID

NEW

This state means that the packet is of a new connection and it is most probably the connection establishing packet that is the SYN packet and it is obviously going from the source to the destination

ESTABLISHED

This state means that traffic has passed in both directions and now the packets from that connection would be matched. The requirement to get into the ESTABLISHED state is that client requests from the server and gets a reply in return

RELATED

A connection is in the RELATED state when it is expected that it is spawned from an already ESTABLISHED connection. E.g.: Thread connections that the server spawns after the initial connection to the well known port are considered to be RELATED to the initial connection

INVALID

This state is used in the case where the packet cannot be identified or it does not have any state. This may be caused due to several factors like ICMP packets which are connectionless or sometimes even UDP packets

The conntrack module keeps the states in the memory and only releases a state if certain conditions are met. They are managed in a hash table where a linked list is used to resolve collisions. An entry in this table is of type ip_conntrack_tuple_hash and contains a reverse pointer to the ip_conntrack structure of that connection in addition to the actual address information i.e. tuple which has the source and destination addresses and protocol specific information which includes port numbers

A more detailed implementation has been given below which can be skipped in the context of the project.

The conntrack module has been implemented in the following files

/include/linux/netfilter_ipv4/

ip_conntrack.h
ip_conntrack_core.h
ip_conntrack_ftp.h
ip_conntrack_helper.h
ip_conntrack_icmp.h
ip_conntrack_irc.h
ip_conntrack_protocol.h
ip_conntrack_tcp.h
ip_conntrack_tuple.h
ip_conntrack_core.c
ip_conntrack_ftp.c
ip_conntrack_irc.c
ip_conntrack_proto_generic.c
ip_conntrack_proto_icmp.c
ip_conntrack_proto_tcp.c
ip_conntrack_proto_udp.c
ip_conntrack_standalone.c

TUPLE

A tuple is a structure that contains information that identifies it to a connection. Thus if two packets have the same tuple, they are in the same connection.

```
 union ip_conntrack_manip_proto
  {
                                    u_int16_t all;    //Add other protocols here
                                    struct
                                        {
                                              u_int16_t port;
                                         } tcp;
                                    struct
                                              {
                                              u_int16_t port;
                                              } udp;
                                    struct
                                              {
                                                          u_int16_t id;
                                              } icmp;
   };     // ip_conntrack_tuple.h
```

```
//The manipulable part of the tuple.

struct ip_conntrack_manip
{
   u_int32_t ip;
   union ip_conntrack_manip_proto u;
};

// ip_conntrack_tuple.h

//This contains the information to distinguish a connection.

struct ip_conntrack_tuple
{
        struct ip_conntrack_manip src;
        struct {
               u_int32_t ip;

               union
                {
                 u_int16_t all;    //Add other protocols here.
               struct {       u_int16_t port;               } tcp;
```

```
            struct {     u_int16_t port;           } udp;
            struct {     u_int8_t type, code;      } icmp;
             } u;
       uint16_t protonum;           //The protocol.
      } dst;
};
```

Hash Functions of Connection Track

```
struct ip_conntrack_tuple_hash
{
struct list_head list;
struct ip_conntrack_tuple tuple;
struct ip_conntrack *ctrack;
// this == &ctrack->tuplehash[DIRECTION(this)].
};
struct ip_conntrack
{
……………
struct ip_conntrack_tuple_hash tuplehash[IP_CT_DIR_MAX];
//These are my tuples; original and reply
volatile unsigned long status;
// Have we seen traffic both ways yet
struct timer_list timeout;
        //Timer function; drops refcnt when it goes off.
struct ip_conntrack_expect expected;
/* If we're expecting another related connection, this will be in expected linked list */
struct nf_ct_info master;
/* If we were expected by another connection, this will be it */
…………….
}



static inline u_int32_t



hash_conntrack(const struct ip_conntrack_tuple *tuple)
{
…….
//
return (ntohl(tuple->src.ip + tuple->dst.ip
+ tuple->src.u.all + tuple->dst.u.all
+ tuple->dst.protonum)
+ ntohs(tuple->src.u.all))
% ip_conntrack_htable_size;
}
```

Getting Connection Information

There is an enumerated type in the conntrack module which tells us the state of the connection when we call the function ip_conntrack_get()

```
enum ip_conntrack_info
{
     /* Part of an established connection (either direction). */
    IP_CT_ESTABLISHED,
     IP_CT_RELATED,
     IP_CT_NEW,
     IP_CT_IS_REPLY,    /* >=this indicates reply direction */
     IP_CT_NUMBER = IP_CT_IS_REPLY * 2 - 1
/* Number of distinct IP_CT types (no NEW in reply dirn). */
};
```

IP_CT_NEW

The packet is trying to create a new connection obviously from source to destination

IP_CT_ESTABLISHED

The packet is part of an established connection from source to destination

IP_CT_ESTABLISHED+ IP_CT_IS_REPLY

The packet is part of an established connection from destination to source

IP_CT_RELATED

The packet is related to the connection from source to destination

IP_CT_RELATED + IP_CT_IS_REPLY

The packet is related to the connection and is from destination to source

2.5 Routing Tables

There are three basic routing tables in Linux as described in [9]. These are

- Routing Cache or Multicast
- Forwarding Information Base Table (FIB)
- Neighbor Table

The Neighbor Table

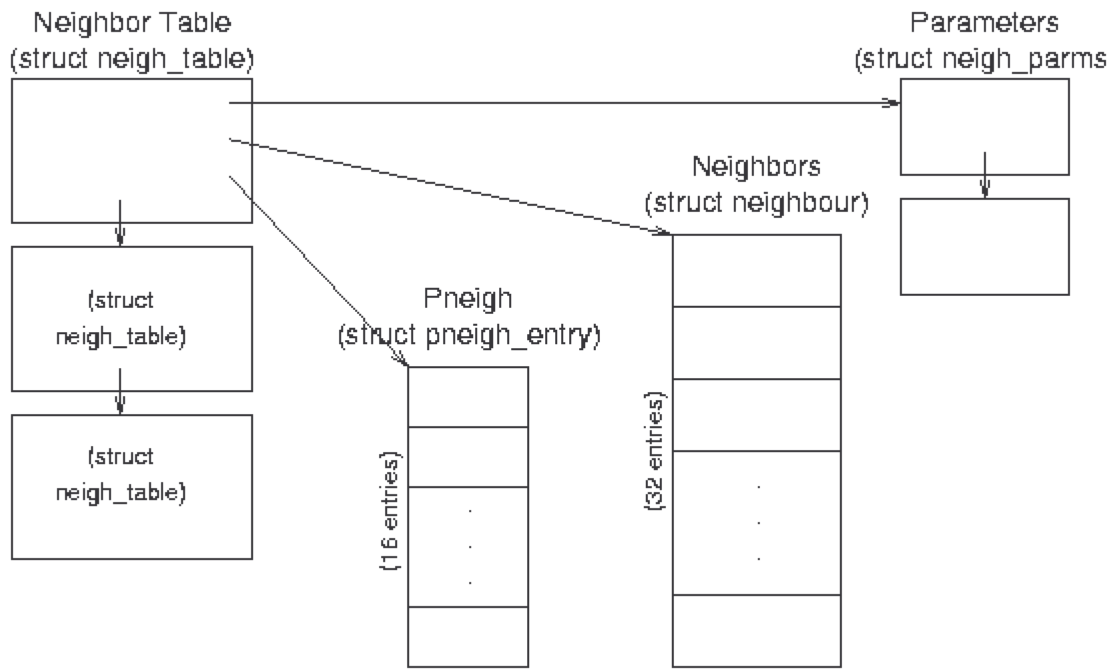
The Neighbor Table whose structure is shown below contains information about computers that are physically linked with the host computer. Entries are not persistent; this table may contain no entries or may contain as many entries as there are computers physically connected to its network. Entries in the table are actually other table structures which contain addressing, device, protocol, and statistical information. In this diagram we see that the structure neighbor table is a pointer to a list of neighbor tables; each table contains a set of general functions and data and a hash table (A table in which keys are mapped to specific positions by a function that gives these positions) of specific information about a set of neighbors. This is a very detailed, low level table containing specific information such as the approximate transit time for messages, queue sizes, device pointers, and pointers to device functions.

Neighbor Table (struct neigh_table) Structure - this structure (a list element) contains common neighbor information and table of neighbor data and pneigh data (which presumably describes a proxy neighbor). All computers connected through a single type of connection will be in the same table.

- struct neigh_table *next - pointer to the next table in the list.
- struct neigh_parms parms - structure containing message travel time, queue length, and statistical information; this is actually the head of a list.
- struct neigh_parms *parms_list - pointer to a list of information structures regarding neighbors.
- struct neighbour *hash_buckets[] - hash table of neighbors associated with this table
- struct pneigh_entry *phash_buckets[] - hash table of structures containing device pointers and keys of proxy neighbors (presumably)
- Other fields include timer information, function pointers, locks, and statistics.

Neighbor Data (struct neighbour) Structure - these structures contain the specific information about each neighbor.

- struct device *dev - pointer to the device or interface that is connected to this neighbor.
- __u8 nud_state - status flags; values can be incomplete, reachable, stale, etc.; also contains state information for permanence and ARP use.
- struct hh_cache *hh - pointer to cached hardware header for transmissions to this neighbor.
- struct sk_buff_head arp_queue - pointer to ARP packets for this neighbor.
- Other fields include list pointers, function (table) pointers, and statistical information.



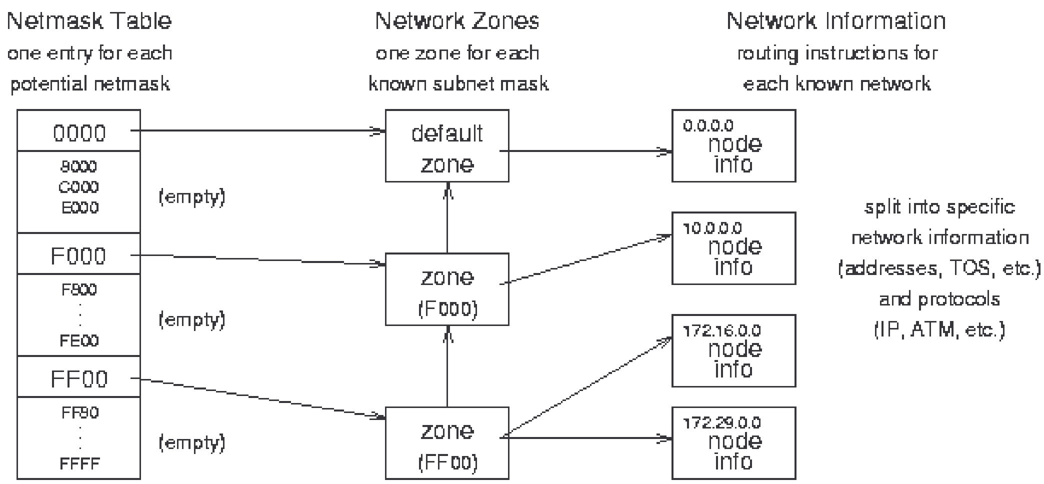
Neighbor Table data structure relationships.

The Forwarding Information Base

The Forwarding Information Base (FIB) is the most important routing structure in the kernel; it is a complex structure that contains the routing information needed to reach any valid IP address by its network mask. Essentially it is a large table with general address information at the top and very specific information at the bottom. The IP layer enters the table with the destination address of a packet and compares it to the most specific netmask to see if they match. If they do not, IP goes on to the next most general netmask and again compares the two. When it finally finds a match, IP copies the route to the distant host into the routing cache and sends the packet on its way.

A fib_table structure forms the basis for a routing table. This structure includes a pointer to an fn_zone structure for each potential prefix length (0 to 32 bits). All routing table entries with the same prefix length are allocated to a specific fn_zone structure (there is one zone for each subnet mask). The fn_zone structure uses an additional hash table to store the individual entries, each represented by a fib_node structure. The hash functions used for this purpose also uses the entry's network prefix. If several routing table entries have the same hash value, then the corresponding fib_node structures are linked in a linear list. Ultimately, the actual data of an entry is not in the fib_node structure itself, but in a fib_info structure referenced in the former structure.

In the diagram given below we can see that the netmask table would have one entry for each potential netmask out of which we use certain netmasks. These entries would then point to the zone entry which is simply the subnet masks which we know exist on the network. This zone structure in turn points to one or more fib_node structures which are simply where the routing instruction for each known network is stored with additional information needed by the network.In routing the longest subnet mask is checked first progressively going lower and each entry is then checked with a hash entry made with the source address, destination address and the specific entry interface.



Forwarding Information Base (FIB)

More specific implementation of the FIB has been given below which can however be skipped in the context of this project.

struct fib_table *local_table, *main_table - these global variables are the access points to the FIB tables; they point to table structures that point to hash tables that point to zones. The contents of the main_table variable are in /proc/net/route.

FIB Table fib_table Structure - include/net/ip_fib.h - these structures contain function jump tables and each point to a hash table containing zone information. There is usually only one or two of these.

- int (*tb_*functions*)() - pointers to table functions (lookup, delete, insert, etc.) that are set during initialization to fn_hash_*function*().
- unsigned char tb_data[0] - pointer to the associated FIB hash table (despite its declaration as a character array).
- unsigned char tb_id - table identifier; 255 for local_table, 254 for main_table.
- unsigned tb_stamp

Netmask Table fn_hash Structure - net/ipv4/fib_hash.c - these structures contain pointers to the individual zones, organized by netmask. (Each zone corresponds to a uniquely specific network mask.) There is one of these for each FIB table (unless two tables point to the same hash table).

- struct fn_zone *fn_zones[33] - pointers to zone entries (one zone for each bit in the mask; fn_zone[0] points to the zone for netmask 0x0000, fn_zone[1] points to the zone for 0x8000, and fn_zone[32] points to the zone for 0xFFFF.
- struct fn_zone *fn_zone_list - pointer to first (most specific) non-empty zone in the list; if there is an entry for netmask 0xFFFF it will point to that zone, otherwise it may point to zone 0xFFF0 or 0xFF00 or 0xF000 etc.

Network Zone fn_zone Structure - net/ipv4/fib_hash.c - these structures contain some hashing information and pointers to hash tables of nodes. There is one of these for each known netmask.

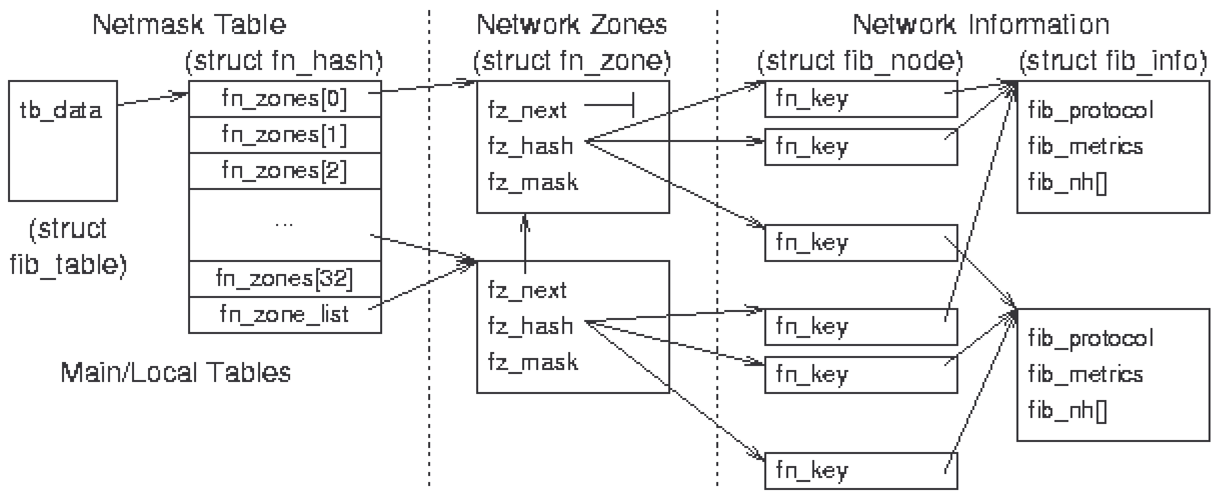
- struct fn_zone *fz_next - pointer to the next non-empty zone in the hash structure (the next most general netmask; e.g., fn_hash- > fn_zone[28]- > fz_next might point to fn_hash- > fn_zone[27]).
- struct fib_node **fz_hash - pointer to a hash table of nodes for this zone.
- int fz_nent - the number of entries (nodes) in this zone.
- int fx_divisor - the number of buckets in the hash table associated with this zone; there are 16 buckets in the table for most zones (except the first zone - 0000 - the loopback device).
- u32 fz_hashmask - a mask for entering the hash table of nodes; 15 (0x0F) for most zones, 0 for zone 0).
- int fz_order - the index of this zone in the parent fn_hash structure (0 to 32).
- u32 fz_mask - the zone netmask defined as ~((1<<(32-fz_order))-1); for example, the first (zero) element is 1 shifted left 32 minus 0 times (0x10000), minus 1 (0xFFFF), and complemented (0x0000). The second element has a netmask of 0x8000, the next 0xC000, the next 0xE000, 0xF000, 0xF800, and so on to the last (32d) element whose mask is 0xFFFF.

Network Node Information fib_node Structure - net/ipv4/fib_hash.c - these structures contain the information unique to each set of addresses and a pointer to information about common features (such as device and protocols); there is one for each known network (unique source/destination/TOS combination).

- struct fib_node *fn_next - pointer to the next node.
- struct fib_info *fn_info - pointer to more information about this node (that is shared by many nodes).
- fn_key_t fn_key - hash table key - the least significant 8 bits of the destination address (or 0 for the loopback device).
- Other fields include specific information about this node (like fn_tos and fn_state).

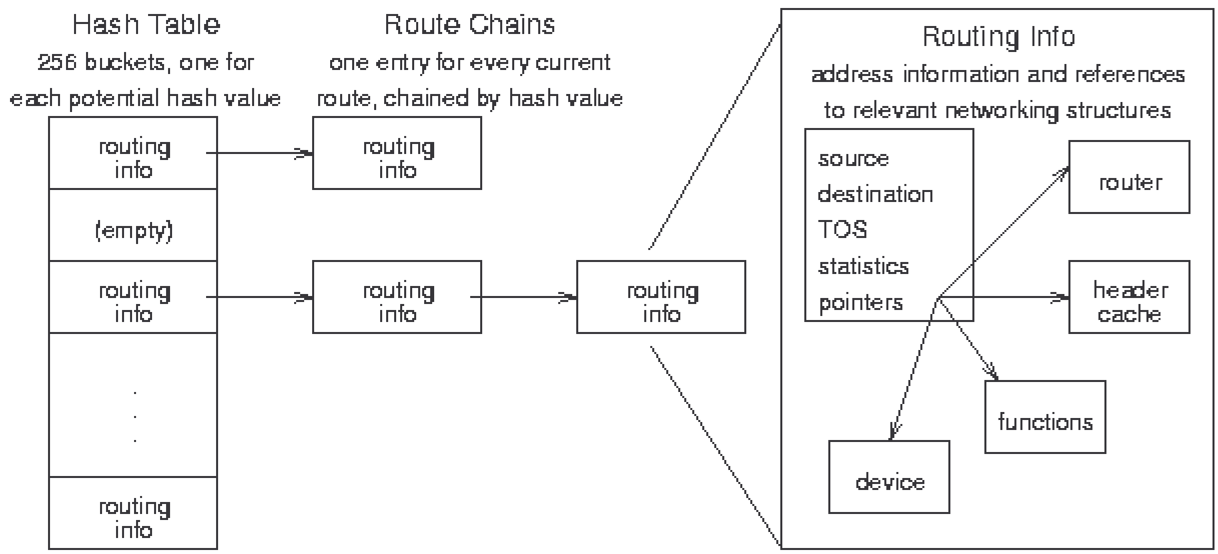
Network Protocol Information (fib_info) Structure - include/net/ip_fib.h - these structures contain protocol and hardware information that are specific to an interface and therefore common to many potential zones; several networks may be addressable through the same interface. There is one of these for each interface.

- fib_protocol - index to a network protocol (e.g., IP) used for this route.
- struct fib_nh fib_nh[0] - contains a pointer to the device used for sending or receiving traffic for this route.
- Other fields include list pointers and statistical and reference data (like fib_refcnt and fib_flags.



Forwarding Information Base (FIB) data relationships

The Routing Cache



Routing Cache conceptual organization

The routing cache is the fastest method Linux has to find a route; As can be seen in the diagram above Linux keeps every route that is currently in use or has been used recently in a hash table (A table in which keys are mapped to specific positions by a function that gives these positions) which has a maximum of 256 buckets (entries) and stores in it a pointer according to the combination of the source address, destination address and incoming interface. When IP needs a route, it goes to the appropriate hash bucket which is found using the hash function and searches the chain (linked list) of cached routes until it finds a match, then sends the packet along that path which the routing information node gives to IP. Routes are chained in order, most frequently used first, and have timers and counters that remove them from the table when they are no longer in use. If the routing cache is unable to provide the route the FIB table is looked up which has been explained before

A more detailed explanation of the routing cache has been given below for the audience which wants to study it further.

struct rtable *rt_hash_table[RT_HASH_DIVISOR] - this global variable contains 256 buckets of (pointers to) chains of routing cache (rtable) entries; the hash function combines the source address, destination address, and TOS to get an entry point to the table (between 0 and 255). The contents of this table are listed in /proc/net/rt_cache.

Routing Table Entry (rtable) Structure - include/net/route.h - these structures contain the destination cache entries and identification information specific to each route.

- union < struct dst_entry dst; struct rtable* rt_next) > u - this is an entry in the table; the union structure allows quick access to the next entry in the table by overusing the rtable's next field to point to the next cache entry if required.
- __u32 rt_dst - the destination address.
- __u32 rt_src - the source address.
- rt_int iif - the input interface.

- __u32 rt_gateway - the address of the neighbor to route through to get to a destination.
- struct rt_key key - a structure containing the cache lookup key (with src, dst, iif, oif, tos, and scope fields)
- Other fields contain flags, type, and other miscellaneous information.

Destination Cache (dst_entry) Structure - include/net/dst.h - these structures contain pointers to specific input and output functions and data for a route.

- struct device *dev - the input/output device for this route.
- unsigned pmtu - the maximum packet size for this route.
- struct neighbor *neighbor - a pointer to the neighbor (next link) for this route.
- struct hh_cache *hh - a pointer to the hardware header cache; since this is the same for every outgoing packet on a physical link, it is kept for quick access and reuse.
- int (*input)(struct sk_buff*) - a pointer to the input function for this route (typically tcp_recv()).
- int (*output)(struct sk_buff*) - a pointer to the output function for this route (typically dev_queue_xmit()).
- struct dst_ops *ops - a pointer to a structure containing the family, protocol, and check, reroute, and destroy functions for this route.
- Other fields hold statistical and state information and links to other routing table entries.

Neighbor Link (neighbor) Structure - include/net/neighbor.h - these structures, one for each host that is exactly one hop away; contain pointers to their access functions and information.

- struct device *dev - a pointer to device that is physically connected to this neighbor.
- struct hh_cache *hh - a pointer to the hardware header that always precedes traffic sent to this neighbor.
- int (*output)(struct sk_buff*) - a pointer to the output function for this neighbor (typically dev_queue_xmit()?).
- struct sk_buff_head arp_queue - the first element in the ARP queue for traffic concerning this neighbor - incoming or outgoing?
- struct neigh_ops *ops - a pointer to a structure that containing family data and and output functions for this link.
- Other fields hold statistical and state information and references to other neighbors.

Updating Routing Information

Linux only updates routing information when necessary, but the tables change in different manners. The routing cache is the most volatile, while the FIB usually does not change at all.

The neighbor table changes as network traffic is exchanged. If a host needs to send something to an address that is on the local subnet but not already in the neighbor table, it simply broadcasts an ARP request and adds a new entry in the neighbor table when it gets a reply. Periodically entries time out and disappear; this cycle continues indefinitely. The kernel handles most changes automatically.

The FIB on most hosts and even routers remains static; it is filled in during initialization with every possible zone to route through all connected routers and never changes unless one of the routers goes down. Changes have to come through external ioctl() calls (Input Output Control) to add or delete zones.

The routing cache changes frequently depending on message traffic. If a host needs to send packets to a remote address, it looks up the address in the routing cache and FIB if necessary and sends the packet off through the appropriate router. On a host connected to a LAN with one router to the Internet, every entry will point to either a neighbor or the router, but there may be many entries that point to the router. The entries are created as connections are made and time out quickly when traffic to that address stops flowing. Everything is done with IP level calls to create routes and kernel timers to delete them.

3 System Functional Specification

3.1 Functions Performed

The functions performed by the system are

- Check if incoming packet is of a new connection or an already established connection
- If the packet is of a new connection then it is checked whether the incoming source address is already known and valid
- If the source address is already known and valid then the packet is deemed to be acceptable
- If the source address is already known and invalid then the packet is deemed unacceptable and the net filter is informed to drop the packet
- If the source address is not known then an echo packet is sent to the original source
- If the echo packet is replied then the source is deemed to be valid and further packets are allowed to go through.
- If the packet is of an already established connection then it is checked whether the source address is valid and if it is then it is allowed to go through

3.2 External and Internal Limitations and Restrictions

The external restrictions are

- IP addresses of the various interfaces of the router cannot be known directly thus to work around it incoming packets have to be caught and their destination addresses have to be extracted along with their devices.
- No User Space functions can be accessed from the kernel thus only internal functions of the kernel are used.

The External Limitations are

- If the spoofing source and the real source exist on the same subnet then the packets are allowed to go through. This is an open research problem and work is still being done to solve it
- If the host computer is in the unlikely case running a firewall that blocks ICMP packets then even a real source could be declared invalid. This limitation is resolved in the fact that this module works intra LAN where there are no firewalls only on the edges

4 System Performance Requirements

4.1 Efficiency

The module has to be extremely efficient as the packet in the network stack cannot be delayed for too long. If the delay is too large in the module it can lead to timeout in the original source leading to packet loss and also slowing down the entire connection according to the implementation of the TCP protocol. Thus connection tracking has been made part of the design as in the implementation only one comparison has to be made that the packet is of a new connection or from an already established connection. There is also the feature of conditionally accepting the packet as further packets cannot be held up while we wait for the ICMP_ECHOREPLY packet of the source which we are probing.

4.2 Reliability

4.2.1 Description of Reliability Measures and Failure Rate

Consistency across the various packets has to be maintained that is if the source address once declared invalid then no packet of that source address should be able to pass through. It should also be precisely be able to declare if the packet is valid or invalid as a genuine source should not have its service denied on the other hand if an invalid source address is allowed to go through then the basic purpose of the module has been defeated.

The software should have a meantime between failures rate as it is part of the kernel and if it fails then the entire kernel crashes bringing down the entire Operating System with it.

4.3 Maintainability

The Module once loaded does not needed further maintenance and can run indefinitely as part of the kernel. It is integrated into the kernel and does not need any further interference from the user.

4.4 Modifiability

The project can be easily modified and recompiled due to the inherent properties of the kernel modules which were chosen as the medium of modification of the kernel which are namely

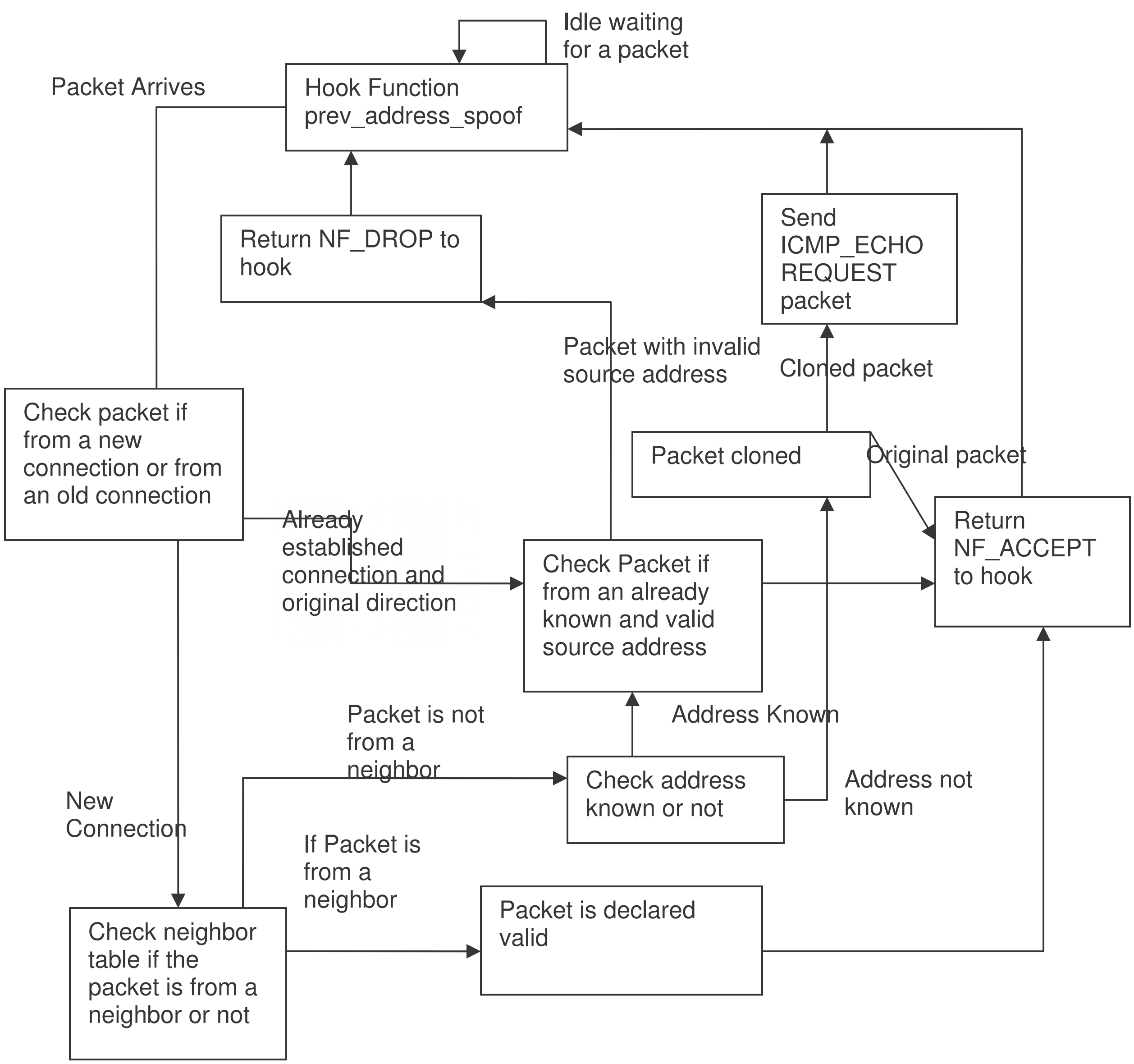
- The Kernel is not recompiled as often. This saves time and minimizes the possibility of introducing an error in rebuilding and reinstalling the base kernel.
- Linux Kernel Modules save memory, as they have to be loaded when they are to be used as opposed to the base kernel whose parts stay loaded all the time in real storage, not just virtual storage.
- Linux Kernel Modules are much faster to maintain and debug.
- Linux Kernel Modules are also not slower than base kernel modules.

4.5 Portability

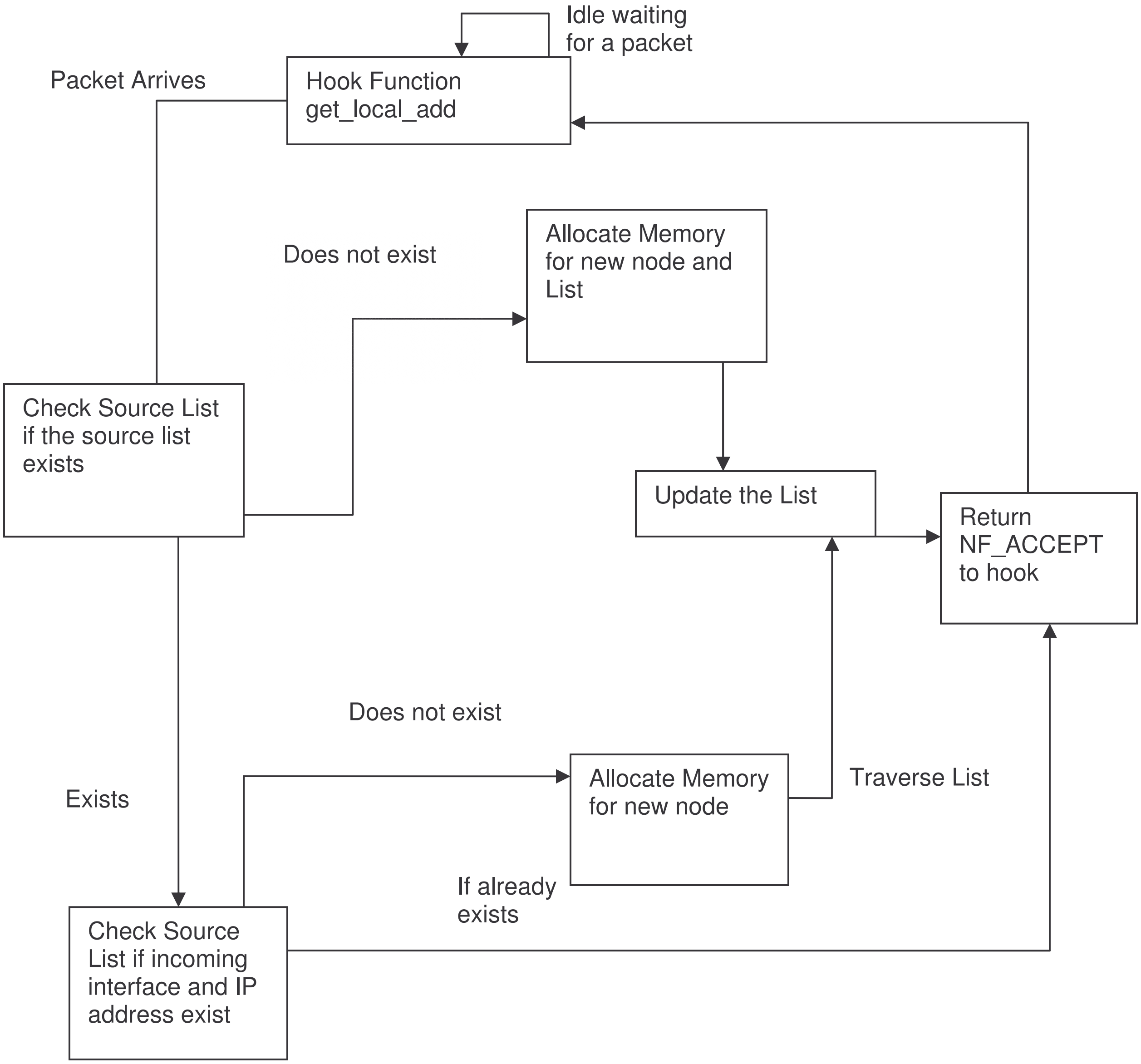
As the module has been written for the Linux Platform, It will only be able to work with the Linux operating system and not any other operating system. The most common cause of this is the case where the operating system is closed source i.e. the source code of the operating system is not available. It can however work across the various 'Flavors of Linux' as the various implementations of the Linux platform are called. The software has been tested across the various implementations of the Linux – 2.4 kernels but portability across the current kernels available which are still in development and currently unstable is not guaranteed as the procedure of implementation of Linux Kernel Modules has been changed in the later kernels

5 System Design Overview

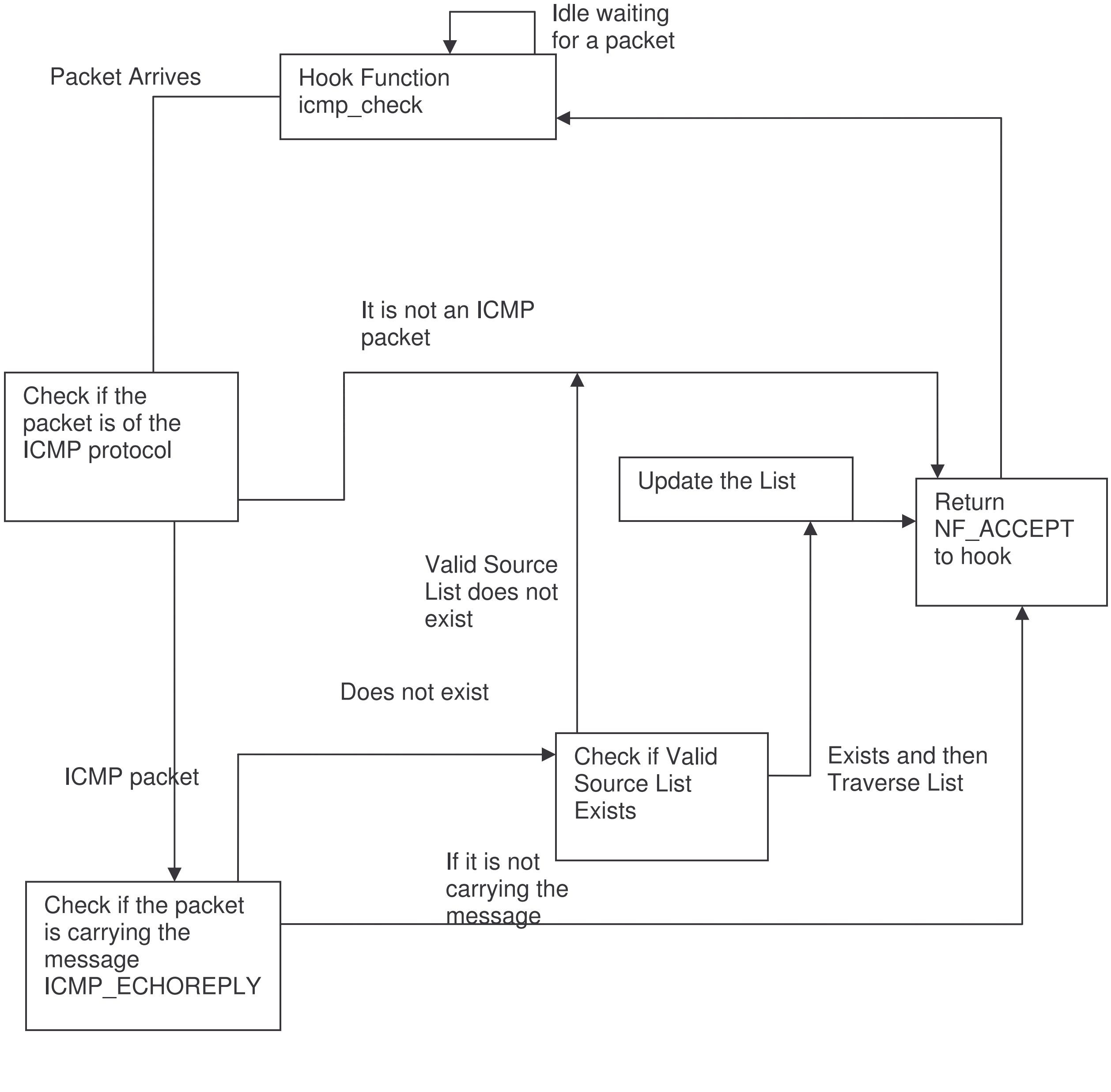
5.1 System Data Flow Diagrams



<u>Data Flow Diagram for the prev_addr_spoof function</u>



Data Flow Diagram for the get_local_add function



Data Flow Diagram for the icmp_check function

5.2 System Internal Data Structure

The systems internal data structures would comprise of two lists, one list that stores the valid and invalid source addresses through the use of a counter and the device they came from along with a counter that specifies how long we are willing to wait for the reply of the ICMP_ECHOREQUEST packet. The other list would comprise of the interface names and the IP addresses that are associated with that IP address.

5.3 Description of System Operation

The project is basically designed to rectify the fault that source addresses are not checked to be valid or not. This is done by using Loadable Linux Kernel Modules, Net Filter hooks and Connection Tracking (explained in detail in section 2). There are two lists maintained by the system The first list stores all the different source addresses of the packets that have been seen by this router, This list also stores information of the source address i.e. the device where the packet came from and whether they are valid or not thus for every new connection that is seen this list is traversed for the proper source address and validated. If it is not valid then the packets are blocked and no further packets are allowed to go through. If the source address is seen for the first time then the address is first checked with the neighbor table i.e. the table which stores the machines which are directly connected to this router to verify whether it is a neighbor or not if it is not an neighbor then the address is stored in the master list with the device and a temporary value that it is not valid but is in the process of being verified. For verification an ICMP_ECHOREQUEST is sent to the original source using the second list which stores the various IP addresses of the router and the devices that are associated with these IP addresses. This list is maintained by a function registered in the NF_IP_LOCAL_IN hook of the net filter facility which extracts the destination address from the incoming packets and the device from which the packet has come from. This is all done in a function that is registered in the NF_IP_FORWARD hook of the net filter facility. If the source address of the packet is valid then the original source of the new connection packet would reply back by an ICMP packet that has the code set as ICMP_ECHOREPLY which would then be caught by another function that has also been registered in the NF_IP_LOCAL_IN hook, This function would then update the source address list and change the source address from being invalid to valid and discard the temporary invalid status This would be done if and only if the packet has arrived from the device that the original packet came from. Thus as proved by testing source address spoofing can be successfully eliminated.

5.4 Implementation Languages

The language used in the implementation would be C as the source code of Implementation of the Linux Kernel – 2.4 is written in the C language thus to use the existing functionality of the kernel i.e. to port our code into the hook of the net filter and extend the functionality of the existing kernel, C language is the language of choice.

5.5 Required Support Software

The required support software that is needed is the Linux 2.4 Kernel as the module is an extension of the above kernel and the Linux Platform to run the module

6 System Data Structure Specifications

6.1 Other User Input Specification

6.1.1 Identification of Input Data

The input of the program would be the SK_BUFF at the IP layer i.e. the packet structure with the pointers to the packet itself as passing of the entire packet will occupy a huge memory space thus the pointer to the structure is passed through which we can access the packet data itself.

6.1.2 Source of Input Data

The source of the input data would be the hook calling function nf_iterate() which goes over the list of hook functions for that particular hook arranged according to ascending values of priority and passes to each of them a pointer of the SK_BUFF structure

6.1.3 Input Device

The input device is the interface receiving the packet from the network at the physical level

6.1.4 Data Format

Data would be received as a pointer to a structure known as SK_BUFF thus its format would be of a structure type and would have to be accessed accordingly

6.2 Other User Output Specification

6.2.1 Identification of Output Data

The output of the module would be the signal that the packet is valid or not i.e. NF_ACCEPT or NF_DROP

6.2.2 Destination of Output Data

The destination of the output data is the variable verdict which collects all the responses from the various hook functions and then according to the responses received decides what to do with the packet

6.2.3 Output Device

As the module mainly deals with the forwarded packets then the output device is the interface from which the packet is routed out of the router

6.2.4 Output Interpretation

The output basically signifies whether after processing the packet has a valid source address or not if it is has an invalid source address then the output NF_DROP would be returned which tells net filter to drop the packet and if it is valid then the value NF_ACCEPT would be returned which tells net filter to continue further processing of the packet

6.3 System Internal Data Structure Specification

6.3.1 Identification of Data Structures

There would be three data structures namely

- A link list structure that would store the valid and invalid source addresses along with some mechanism to identify them and also devices from which they were already seen. It would also include a variable that would identify that the source address is still under probation whether it is valid or invalid
- A link list structure that would store the interface and the IP address associated with that interface
- The third structure would be inherited and would be required in the declaration and registration of the hooks in the net filter architecture

6.3.2 Modules Accessing Structures

The modules and how they would access the structures are given below

Init_module

- This would use the inherited structure and update it with the values required to register the hooks and also register them in the netfiler architecture

cleanup_module

- This would also use the inherited structure to un register the hooks from the netfilter architecture

Prev_Addr_Spoof

- It would use the link list with valid and invalid source addresses to identify whether the incoming packet has a valid source IP address or not
- It would also create and update the same link list as and when it sees unknown or new source IP addresses
- It would also use the link list with the interface and their IP addresses to send out ICMP_ECHOREQUEST packets

get_local_add

- This would use the local link list with the local IP addresses to see if the incoming packets destination address already exists if not then it would update the list with the incoming packets destination address and interface

icmp_check

- This would use the list with the valid and invalid source addresses and check whether the incoming packets source address is validated if invalid then it would update the source list and make the invalid address valid

6.3.3 Logical Structure of Data

The list with valid and invalid IP addresses would have a structure like this

```
struct ip_known // A structure because we want to make a link list
{

   struct net_device *ip_in_dev;

//This would store the device on which the packet has come in.This is useful in checking
//whether the ICMP_ECHOREQUEST packet we sent out has come back from the correct device
//or not

   u32 ip_store;

//This variable would store the actual IP address of the packet

   int valid;

//This variable marks whether the source address is valid or not

   int no_pack;

//This is the temporary marking variable while we wait for the ICMP_ECHOREPLY //packet


   struct ip_known *next;

//This is the variable which would point to the next variable in the singly link list

};
```

The list with the interface and their IP addresses would look like the structure given below

```
struct interf_add
{

 struct net_device *interface_dev;

//The interface address which is stored as we need to compare that with the incoming packets
//interface in order to select the proper source address for the outgoing packet

 u32 interf_ip;

// The IP address of the interface this is stored in order to give the proper IP address to the packet


 struct interf_add *next;

//This is the variable which would point to the next variable in the singly link list


};
```

The structure which would be used to register the hook function is given below

```
struct nf_hook_ops
{


struct list_head list;

// Points to the head of the list


nf_hookfn *hook;

//This pointer is used to point at the function that is to be called whenever a packet hits this hook

int pf;

//This variable would be used to fill in the protocol family for which the packets have to be caught


int hooknum;

//This variable would be used to store in which hook type this hook is registered


int priority;

//This variable gives the priority of the hook as the hooks are ordered in ascending priority in the
//link list
};
```

7 Module Design specifications

7.1 Module Functional specification

7.1.1 Functions Performed

The functions performed by the various modules are

Init_module

- This would use the inherited structure and update it with the values required to register the hooks and also register them in the netfiler architecture

cleanup_module

- This would also use the inherited structure to un register the hooks from the netfilter architecture

Prev_Addr_Spoof

- It would check whether the incoming packet is of a new connection or from an already established connection.
- If the packet is from a new connection then the following steps given below are followed
- It would use the link list with valid and invalid source addresses to identify whether the incoming packet has a valid source IP address or not
- It would also create and update the same link list as and when it sees unknown or new source IP addresses
- It would also use the link list with the interface and their IP addresses to send out ICMP_ECHOREQUEST packets for the new or unknown source IP addresses
- If the packet is from an already established connection then It would use the link list with valid and invalid source addresses to identify whether the incoming packet has a valid source IP address or not

get_local_add

- This would use the local link list with the local IP addresses to see if the incoming packets destination address already exists if not then it would update the list with the incoming packets destination address and interface

icmp_check

- This would use the list with the valid and invalid source addresses and check whether the incoming packets source address is validated if invalid then it would update the source list and make the invalid address valid

7.1.2 Module Interface Specifications

The module interfaces to be built for the various modules are

Init_module

- Uses three global variables namely of the type static and of the structure nf_hook_ops
- Returns output to the calling command when the module has been first loaded

cleanup_module

- Uses three global variables namely of the type static and of the structure nf_hook_ops

Prev_Addr_Spoof

- The module takes in as argument the hooknum which contains from which hook type this buffer is coming from
- It also takes in a pointer to the structure sk_buff which in turns point to the actual packet data itself
- In addition to that a pointer to the incoming and outgoing device is also passed
- The module also needs global variables pointing to the head of both the link lists for access to the link lists

get_local_add

- The module takes in as argument the hooknum which contains from which hook type this buffer is coming from
- It also takes in a pointer to the structure sk_buff which in turns point to the actual packet data itself
- In addition to that a pointer to the incoming and outgoing device is also passed
- The module also needs global variables pointing to the head of the link list containing local IP addresses for access to the link list

icmp_check

- The module takes in as argument the hooknum which contains from which hook type this buffer is coming from
- It also takes in a pointer to the structure sk_buff which in turns point to the actual packet data itself
- In addition to that a pointer to the incoming and outgoing device is also passed
- The module also needs global variables pointing to the head of the link list containing valid and invalid source IP addresses for access to the link list

7.2 Module operational Specification

7.2.1 Locally Declared Data Specifications

Module Prev_Addr_Spoof

struct sk_buff *sb = *skb

This is used mainly to make one pointer less when accessing the skbuffer

struct neighbour *neigh

This is used to declare a pointer of the neighbor type which would be used to access the neighbor table structure

struct net_device *indev = sb->dev

This refers to the incoming interface of the packet which is to be used in comparisons

struct net_device *outdev = sb->dst->dev

This refers to the outgoing interface of the packet which is to be used in comparisons

int pingsend

This is a flag used to identify whether a ping should be sent or not

u32 ip_source = sb->nh.iph->saddr

This stores the incoming packets source address

u32 ip_destination = sb->nh.iph->daddr

This stores the outgoing packets destination address

struct ip_conntrack *connect

This declares a pointer of the type ip_conntrack which is useful to get the type of the connection

enum ip_conntrack_info connect_info

This tells which type of connection the incoming packet belongs to.

struct sk_buff *nskb = skb_copy(sb, GFP_ATOMIC)

This variable of the type sk_buff which would hold the copy of the new sk_buff with pointer to a new packet

Module get_local_add

int flag

This is a flag variable which would signify whether the local address already exists in the list or not

struct sk_buff *sb = *skb

This is used mainly to make one pointer less when accessing the skbuffer

Module icmp_check

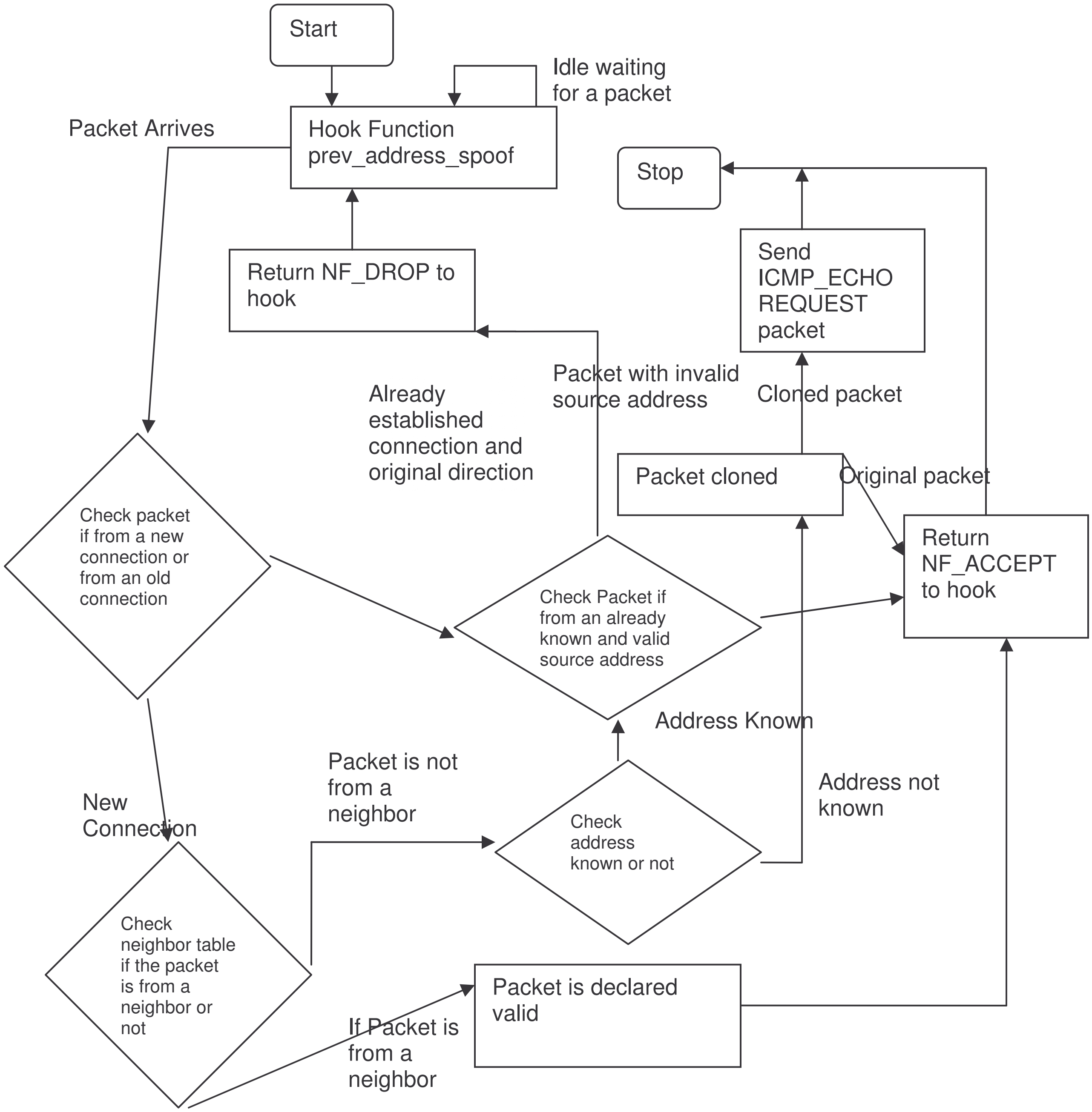
struct sk_buff *sb = *skb

This is used mainly to make one pointer less when accessing the skbuffer

struct icmphdr *icmp

This is used to store the ICMP header of the packet after we extract it from the data portion of the IP packet

7.2.2 Algorithm Specification



Flow Chart for The Prevention of Address Spoofing Function

Algorithm Description

```
function init_module()
start
prevspoof.hook = prev_addr_spoof;
prevspoof.hooknum = NF_IP_FORWARD;
prevspoof.pf = PF_INET;
prevspoof.priority = NF_IP_PRI_FIRST;
nf_register_hook(&prevspoof);
checkic.hook=icmp_check;
checkic.hooknum = NF_IP_LOCAL_IN;
checkic.pf = PF_INET;
checkic.priority = NF_IP_PRI_FIRST;
nf_register_hook(&checkic);
ipadd.hook = get_local_add;
ipadd.hooknum = NF_IP_LOCAL_IN;
ipadd.pf = PF_INET;
ipadd.priority = NF_IP_PRI_FIRST;
nf_register_hook(&ipadd);
return 0;
end function
function cleanup_module()
start
    nf_unregister_hook(&prevspoof);
    nf_unregister_hook(&ipadd);
    nf_unregister_hook(&checkic);
end function
static struct nf_hook_ops prevspoof;
static struct nf_hook_ops ipadd;
static struct nf_hook_ops checkic;
struct interf_add
start
 struct net_device *interface_dev;
 u32 interf_ip;
 struct interf_add *next;
end structure
struct interf_add *curr_interf_add=NULL,*foll_interf_add=NULL;
static struct interf_add *head_local=NULL;
struct ip_known
start
   struct net_device *ip_in_dev;
   u32 ip_store;
```

```
   int valid;
   int no_pack;
   struct ip_known *next;

end structure

struct ip_known *newip_known=NULL,*ip_knownfoll=NULL;
static struct ip_known *ip_head = NULL;

 static void send_ping(u32 dadd,u32 sadd, struct sk_buff *skb_in)
start

   printk("IN new ICMP send\n\n\n");
    skb_in->pkt_type=PACKET_HOST;
   skb_in->nh.iph->saddr=dadd;
   skb_in->nh.iph->daddr=sadd;
   icmp_send(skb_in, ICMP_ECHO, 0, 0);

end function

unsigned int icmp_check(unsigned int hooknum, struct sk_buff **skb,
                                       const struct net_device *in,
                                                         const struct net_device *out,
                                                         int (*okfn)(struct sk_buff *))
start

    struct sk_buff *sb = *skb;
    struct icmphdr *icmp;

     if(sb->nh.iph->protocol != IPPROTO_ICMP)
       return NF_ACCEPT;

       icmp = (struct icmphdr *) (sb->data + sb->nh.iph->ihl * 4);

       if(icmp->type!=ICMP_ECHOREPLY)
       return NF_ACCEPT;

       if(ip_head!=NULL)
      start
           newip_known=ip_head;
            while(newip_known!=NULL)
                start

                if(newip_known->ip_store==sb->nh.iph->saddr)
               start

                if(sb->dev==newip_known->ip_in_dev)
                start

               newip_known->valid=1;
               return NF_ACCEPT;

           end if
            end if


```

```
                  newip_known=newip_known->next;
    end while

    end if

        return NF_ACCEPT;
    end if


    unsigned int get_local_add(unsigned int hooknum, struct sk_buff **skb,
                            const struct net_device *in,
                                        const struct net_device *out,
                                        int (*okfn)(struct sk_buff *))
    start

        int flag=0;
        struct sk_buff *sb = *skb;

        if(head_local==NULL)
        start

          curr_interf_add=(struct interf_add*)kmalloc(sizeof(struct interf_add),GFP_KERNEL);

    if(curr_interf_add==NULL)
    start

              return NF_ACCEPT;

    end if

          curr_interf_add->interface_dev=sb->dev;
          curr_interf_add->interf_ip=sb->nh.iph->daddr;
          curr_interf_add->next=NULL;
          head_local=curr_interf_add;



       else
        start
             flag=0;
             curr_interf_add=head_local;

            while(curr_interf_add!=NULL)

             start

           if(curr_interf_add->interface_dev==sb->dev)

             start
                flag++;

    end if
              foll_interf_add=curr_interf_add;
              curr_interf_add=curr_interf_add->next;

```

```
                        end while

                        if(flag==0)
                        start

                     curr_interf_add=(struct interf_add*)kmalloc(sizeof(struct interf_add),GFP_KERNEL);

                          if(curr_interf_add==NULL)
                          start

                              return NF_ACCEPT;

               end if
                     curr_interf_add->interface_dev=sb->dev;
                     curr_interf_add->interf_ip=sb->nh.iph->daddr;
                     curr_interf_add->next=NULL;
                     foll_interf_add->next=curr_interf_add;

                    end if

                end if

                curr_interf_add=head_local;

                while(curr_interf_add!=NULL)
                start

                  printk("Packet Displayed");
                  printk("device IP Address is  %x \n",curr_interf_add->interf_ip);
                curr_interf_add=curr_interf_add->next;

              end while


               return NF_ACCEPT;

            end function

            unsigned int prev_addr_spoof(unsigned int hooknum, struct sk_buff **skb,
                                             const struct net_device *in,
                                                                    const struct net_device *out,
                                                                    int (*okfn)(struct sk_buff *))

            start

                struct sk_buff *sb = *skb;
                struct neighbour *neigh;
                struct net_device *indev = sb->dev;
                struct net_device *outdev = sb->dst->dev;
                int pingsend=0;


                  u32 ip_source =  sb->nh.iph->saddr;
                 u32 ip_destination =  sb->nh.iph->daddr;
```

```
        u32 ip_saddr=0;

        struct ip_conntrack *connect;
        enum ip_conntrack_info connect_info;

        if(ip_source&&ip_destination)

      start

          connect = ip_conntrack_get(*skb, &connect_info);
                    if(connect_info==IP_CT_NEW)

      start

                       neigh = neigh_lookup(&arp_tbl, &ip_source, indev);
                    printk("IN new connecetion");

              if(neigh!=NULL)
               start

              printk("Packet Accepted");
              printk("Source Address is  %x \n", ip_source);
               printk("Destination Address is  %x \n",ip_destination);
               neigh_release(neigh);
               return NF_ACCEPT;

            end if

        if(head_local!=NULL)
        start

        curr_interf_add=head_local;

        while(curr_interf_add!=NULL)
        start

              pingsend=0;

                if(curr_interf_add->interface_dev==sb->dev)
                 start

                ip_saddr=curr_interf_add->interf_ip;

              if(ip_head==NULL)
    start

    newip_known=(struct ip_known*)kmalloc(sizeof(struct
ip_known),GFP_KERNEL);
    newip_known->ip_store=ip_source;
    newip_known->ip_in_dev=curr_interf_add->interface_dev;
    newip_known->no_pack=10;
    newip_known->valid=0;
    newip_known->next=NULL;
    ip_head=newip_known;
    pingsend=1;
```

```
else
newip_known=ip_head;
while(newip_known!=NULL)
start
if((newip_known->ip_store==sb->nh.iph->saddr))
start
if(newip_known->valid!=0)
start
printk("Packet Accepted");
printk("Source Address is  %x \n", ip_source);
printk("Destination Address is  %x \n",ip_destination);
return NF_ACCEPT;
end if
if(newip_known->no_pack>0)
start
printk("Packet Accepted");
printk("Source Address is  %x \n", ip_source);
printk("Destination Address is  %x \n",ip_destination);
newip_known->no_pack=newip_known->no_pack--;
return NF_ACCEPT;
else
printk("IN packet dropper\n");
printk("Packet Dropped");
printk("Source Address is  %x \n",ip_source);
printk("Destination Address is  %x \n", ip_destination);
return NF_DROP;
end if
end if
ip_knownfoll=newip_known;
newip_known=newip_known->next;
end if
  newip_known=(struct ip_known*)kmalloc(sizeof(struct ip_known),GFP_KERNEL);
newip_known->ip_store=ip_source;
newip_known->ip_in_dev=curr_interf_add->interface_dev;
```

```
newip_known->no_pack=10;
newip_known->valid=0;
newip_known->next=NULL;
pingsend=1;
ip_knownfoll->next=newip_known;
end if
if(pingsend==1)
start
struct sk_buff *nskb = skb_copy(sb, GFP_ATOMIC);
 if (nskb == NULL)
start
            send_ping(ip_source,ip_saddr,sb);
            printk("Packet Accepted");
            printk("Source Address is  %x \n", ip_source);
            printk("Destination Address is  %x \n",ip_destination);
            return NF_STOLEN;
   else
   send_ping(ip_source,ip_saddr,nskb);
   printk("Packet Accepted");
   printk("Source Address is  %x \n", ip_source);
  printk("Destination Address is  %x \n",ip_destination);
 return NF_ACCEPT;
end if
end if
  end if
 curr_interf_add=curr_interf_add->next;
end if
   end while
            printk("Packet Dropped");
            printk("Source Address is  %x \n",ip_source);
            printk("Destination Address is  %x \n", ip_destination);
            return NF_DROP;
      end if
if((connect_info==IP_CT_ESTABLISHED)||(connect_info==IP_CT_RELATED))
start
 printk("IN old connecetion");
      if(ip_head!=NULL)
       start
           newip_known=ip_head;
```

```
                      while(newip_known!=NULL)

                            start
            if((newip_known->ip_store==sb->nh.iph->saddr)||(newip_known->ip_store==sb->nh.iph->daddr))

            start


               if(newip_known->valid!=0)
                start

                             printk("Packet Accepted");
                             printk("Source Address is  %x \n", ip_source);
                             printk("Destination Address is  %x \n",ip_destination);
                            return NF_ACCEPT;


                else

                  if(newip_known->no_pack>0)
                   start

                newip_known->no_pack=newip_known->no_pack--;
                printk("Packet Accepted");
                printk("Source Address is  %x \n", ip_source);
                printk("Destination Address is  %x \n",ip_destination);
                  return NF_ACCEPT;

                             else

                               printk("IN old packet dropper\n");
                                printk("Packet Dropped");
                                 printk("Source Address is  %x \n",ip_source);
                                  printk("Destination Address is  %x \n", ip_destination);
                                 return NF_DROP;

            end if

            end if

               end if
                               newip_known=newip_known->next;
                      end while
                    end if
                                      printk("Packet Accepted ");
                                      printk("Source Address is  %x \n", ip_source);
                                      printk("Destination Address is  %x \n",ip_destination);
                    return NF_ACCEPT;
                   end if
                end if
                 printk("Packet Accepted  due to condition");
                 printk("Source Address is  %x \n", sb->nh.iph->saddr);
                 printk("Destination Address is  %x \n", sb->nh.iph->daddr);
                 return NF_ACCEPT;
              end if
```

7.2.3 Description of Module Operation

Init_module

- This would use the inherited structure and update it with the values required to register the hooks and also register them in the netfiler architecture

cleanup_module

- This would also use the inherited structure to un register the hooks from the netfilter architecture

Prev_Addr_Spoof

- It would check whether the incoming packet is of a new connection or from an already established connection.
- If the packet is from a new connection then the following steps given below are followed
- It would use the link list with valid and invalid source addresses to identify whether the incoming packet has a valid source IP address or not
- It would also create and update the same link list as and when it sees unknown or new source IP addresses
- It would also use the link list with the interface and their IP addresses to send out ICMP_ECHOREQUEST packets for the new or unknown source IP addresses
- If the packet is from an already established connection then It would use the link list with valid and invalid source addresses to identify whether the incoming packet has a valid source IP address or not

get_local_add

- This would use the local link list with the local IP addresses to see if the incoming packets destination address already exists if not then it would update the list with the incoming packets destination address and interface

icmp_check

- This would use the list with the valid and invalid source addresses and check whether the incoming packets source address is validated if invalid then it would update the source list and make the invalid address valid

8 System Verification

8.1 Functions to Be Tested

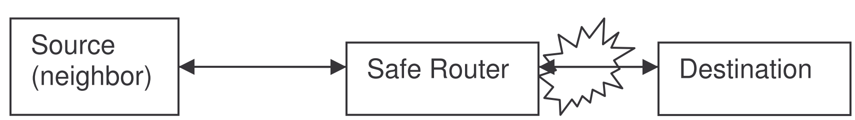
The functions to be tested are

- init_module
- cleanup_module
- prev_addr_spoof
- get_local_add
- icmp_check
- send_ping

8.2 Description of Test Cases

The first test case would comprise of testing whether the module has been successfully integrated into the kernel or not and is able to successfully register the hook functions into the kernel and when unloaded is able to successfully able to un-register the hook functions. This would be achieved by loading the module and getting output on the standard output that the module has been successfully loaded and unloading the module and for verification that it was successfully tested the log files of the kernel i.e. the /var/log/messages was checked.

The second test case would comprise of testing the module to test whether the neighbor can be verified properly and the packet from the neighbor is allowed to pass through successfully.

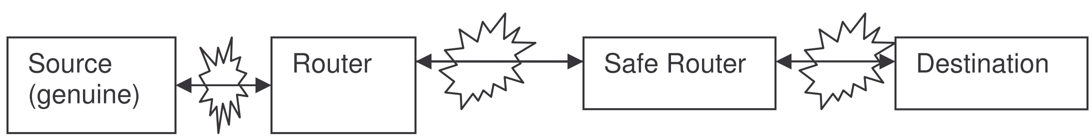


Verification of Successful Packet Transmission of Neighbor

The Third test case would comprise of the neighbor using a spoofed address and then verifying that the packet did not pass through

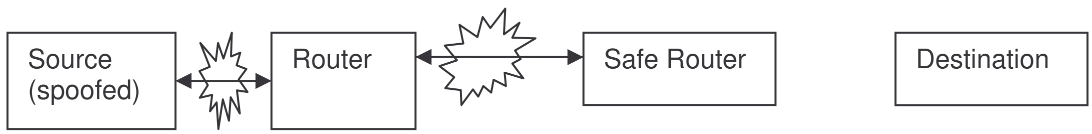
Verification of Unsuccessful Packet Transmission of Spoofing Neighbor

The fourth test case would comprise of verifying whether a packet coming from across a router is able to be correctly verified that the source address is genuine and able to pass through the safe router



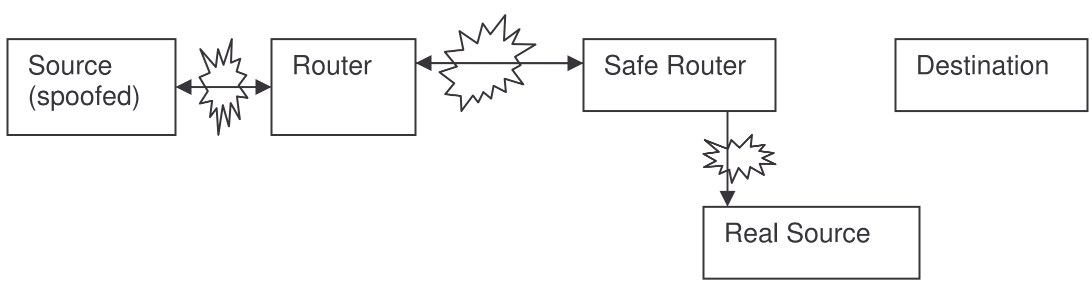
Verification of Successful Packet Transmission of Genuine Source

The fifth test case would comprise of verifying whether a packet coming from across a router is able to be correctly verified that the source address is spoofed by a non existent address on the network and the packet is dropped by the safe router



Verification of Unsuccessful Packet Transmission of Spoofed Source of Non Existent Address

The sixth test case would comprise of verifying whether a packet coming from across a router is able to be correctly verified that the source address is spoofed by an already existing address on the network and the packet is dropped by the safe router



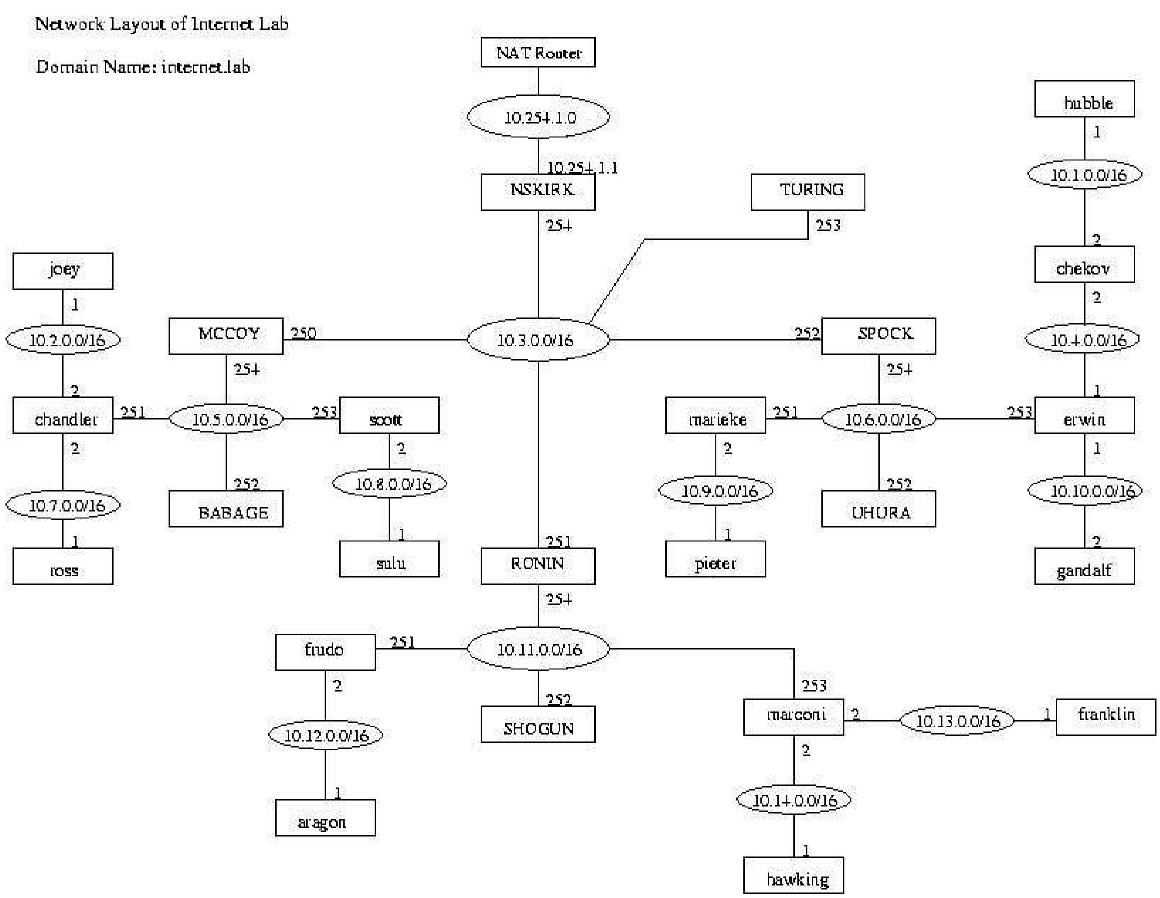
Verification of Unsuccessful Packet Transmission of Spoofed Source of Already Existing Address

8.3 Test Run Procedures and Results

For the purpose of testing we will use three machines namely

- Erwin (IP address: 10.4.0.1 and 10.6.0.254)
- Chekov (IP address 10.1.0.2 and 10.4.0.2)
- Hubble (IP address 10.1.0.1)

The hierarchy of the machines is given as the network diagram below



In this diagram we use Erwin as the safe router where the module has been loaded. Chekov is used as a neighbor to test cases 2 and 3 and Hubble would be used as the machine beyond the router to test cases 4 and 5

We load the module onto Erwin as shown above by the output of test case 1 and log the output to /var/log/messages file which would be reproduced below and use Ethereal which is a network analyzer tool used to decode packets that are caught at the interfaces. Ethereal would be used in Chekov and Hubble

We summarize the test cases with reference to our network as shown below.

Test Case One

The First Test Case verifies that the module has been properly loaded in the router machine Erwin (10.6.0.253, 10.4.0.1 and 10.10.0.1).

Test Case Two

The Second Test Case verifies whether the neighbor Chekov (10.4.0.1) is able to successfully communicate with the machine NSKIRK (10.3.0.254) through the router Erwin (10.6.0.253 and 10.4.0.1).

Test Case Three

The Third Test Case verifies that the neighbor Chekov (10.4.0.2) when using a spoofing address 10.13.0.1 is not able to communicate with the machine NSKIRK (10.3.0.254) through the router Erwin (10.6.0.253 and 10.4.0.1).

Test Case Four

The fourth test case verifies that the machine Hubble (10.1.0.1) is able to communicate with the machine NSKIRK (10.3.0.254) through router Chekov (10.1.0.2 and 10.4.0.2) and then router Erwin (10.4.0.1 and 10.6.0.253)

Test Case Five

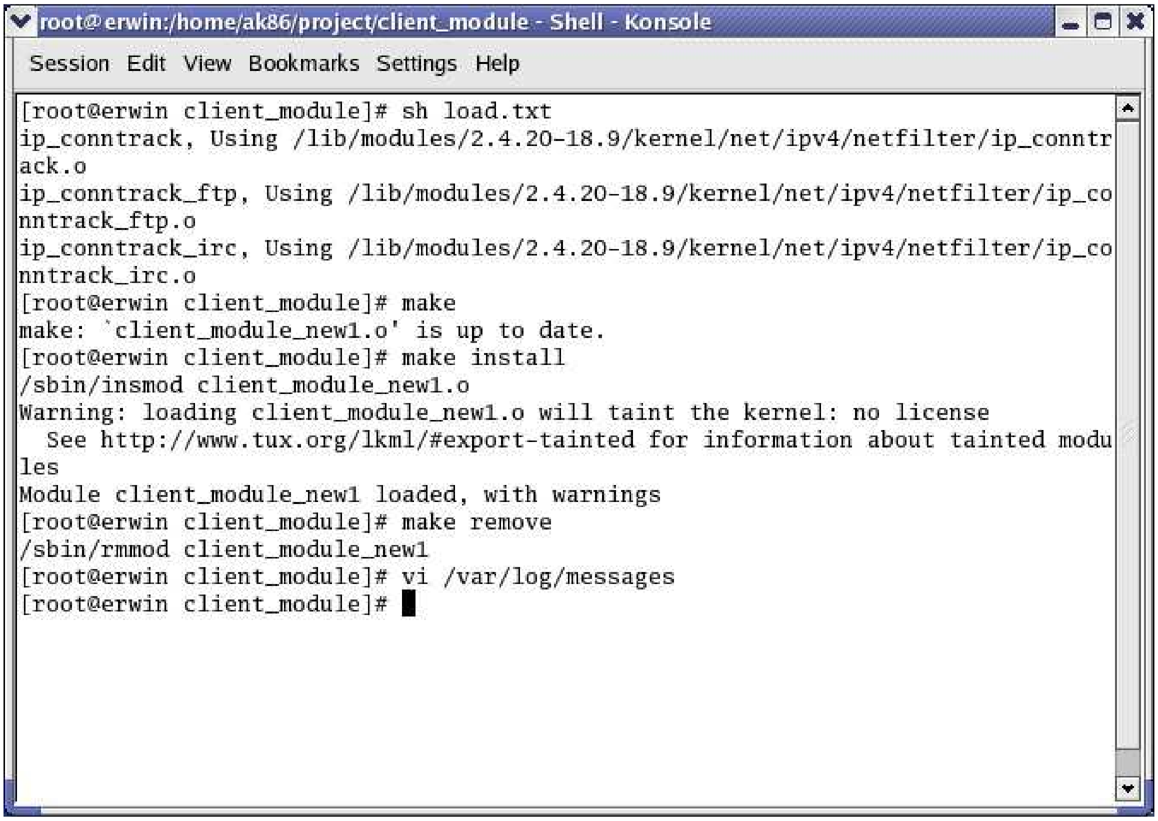
The fifth test case verifies that the machine Hubble (10.1.0.1) is able to send packets through router Chekov (10.1.0.2 and 10.4.0.2) using spoofed IP address 11.13.0.1 which is non existent on the network but the packets were dropped on Erwin (10.4.0.1 and 10.6.0.253) while trying to communicate with the machine NSKIRK (10.3.0.254)

Test Case Six

The sixth test case verifies that the machine Hubble (10.1.0.1) is able to send packets through router Chekov (10.1.0.2 and 10.4.0.2) using spoofed IP address 10.13.0.1 which exists on the network as Franklin but the packets are dropped on Erwin (10.4.0.1 and 10.6.0.253) while trying to communicate with the machine NSKIRK (10.3.0.254)

Test Case One

The first test run would demonstrate the successful loading of the module and also how to go about the rest of the test run as the module has to be loaded every time



In this we can see first the loading of the connection tracking modules by using the command sh load.txt and then running the command make and make install, As is shown in the output the module is successfully loaded and even removed successfully by using the command make remove

Test Case Two and Four

We will first show the output of the test cases 2 and 4 These are when the Sources are genuine and non spoofing and thus we will test that the neighbor sending packets is accepted and able to send packets and also the machine beyond a unsafe router that is directly connected to the safe router is able to send and receive packets

We show Erwin's log file first that is the file /**var**/**log**/**messages**

Dec 6 15:28:56 erwin syslogd 1.4.1: restart.
Dec 6 15:28:56 erwin syslog: syslogd startup succeeded
Dec 6 15:28:56 erwin kernel: klogd 1.4.1, log source = /proc/kmsg started.
Dec 6 15:28:57 erwin syslog: klogd startup succeeded
Dec 6 15:28:56 erwin syslog: syslogd shutdown succeeded

Dec 6 15:29:07 erwin kernel: ip_conntrack version 2.1 (4095 buckets, 32760 max) - 292 bytes per conntrack
Dec 6 15:29:38 erwin kernel: IN new connecetionPacket AcceptedSource Address is 200040a
Dec 6 15:29:38 erwin kernel: Destination Address is fd00030a
Dec 6 15:29:38 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 15:29:38 erwin kernel: Destination Address is 200040a
Dec 6 15:29:38 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 15:29:38 erwin kernel: Destination Address is fd00030a
Dec 6 15:29:38 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 15:29:38 erwin kernel: Destination Address is fd00030a
Dec 6 15:29:38 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 15:29:38 erwin kernel: Destination Address is fd00030a
Dec 6 15:29:38 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 15:29:38 erwin kernel: Destination Address is fd00030a
Dec 6 15:29:38 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 15:29:38 erwin kernel: Destination Address is 200040a
Dec 6 15:29:38 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 15:29:38 erwin kernel: Destination Address is 200040a
Dec 6 15:29:38 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 15:29:38 erwin kernel: Destination Address is 200040a
Dec 6 15:29:38 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 15:29:38 erwin kernel: Destination Address is fd00030a
Dec 6 15:29:38 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 15:29:38 erwin kernel: Destination Address is 200040a
Dec 6 15:29:38 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 15:29:38 erwin kernel: Destination Address is 200040a
Dec 6 15:29:38 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 15:29:38 erwin kernel: Destination Address is 200040a
Dec 6 15:29:38 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 15:29:38 erwin kernel: Destination Address is 200040a
Dec 6 15:29:38 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 15:29:38 erwin kernel: Destination Address is fd00030a
Dec 6 15:29:38 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 15:29:38 erwin kernel: Destination Address is fd00030a
Dec 6 15:29:38 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 15:29:38 erwin kernel: Destination Address is fd00030a
Dec 6 15:29:38 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 15:29:38 erwin kernel: Destination Address is 200040a
Dec 6 15:29:38 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 15:29:38 erwin kernel: Destination Address is fd00030a
Dec 6 15:29:38 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 15:29:38 erwin kernel: Destination Address is fd00030a
Dec 6 15:29:38 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 15:29:38 erwin kernel: Destination Address is 200040a
Dec 6 15:29:38 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 15:29:38 erwin kernel: Destination Address is fd00030a
Dec 6 15:29:38 erwin kernel: Packet Displayeddevice IP Address is 100040a
Dec 6 15:29:38 erwin kernel: Packet Displayeddevice IP Address is fd00060a
Dec 6 15:29:39 erwin kernel: IN new connecetionIN new ICMP send
Dec 6 15:29:39 erwin kernel:
Dec 6 15:29:39 erwin kernel:
Dec 6 15:29:39 erwin kernel: Packet AcceptedSource Address is fa00030a
Dec 6 15:29:39 erwin kernel: Destination Address is 1000040a
Dec 6 15:29:39 erwin kernel: Packet Displayeddevice IP Address is 100040a
Dec 6 15:29:39 erwin kernel: Packet Displayeddevice IP Address is fd00060a
Dec 6 15:29:39 erwin kernel: Packet Displayeddevice IP Address is 100040a

```
Dec  6 15:29:39 erwin kernel: Packet Displayeddevice IP Address is  fd00060a
Dec  6 15:29:40 erwin kernel: Packet Displayeddevice IP Address is  100040a
Dec  6 15:29:40 erwin kernel: Packet Displayeddevice IP Address is  fd00060a
Dec  6 15:30:18 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 15:30:18 erwin kernel: Destination Address is  fd00030a
Dec  6 15:30:18 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 15:30:18 erwin kernel: Destination Address is  200040a
Dec  6 15:30:18 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 15:30:18 erwin kernel: Destination Address is  fd00030a
Dec  6 15:30:19 erwin kernel: IN new connecetionIN new ICMP send
Dec  6 15:30:19 erwin kernel:
Dec  6 15:30:19 erwin kernel:
Dec  6 15:30:19 erwin kernel: Packet AcceptedSource Address is  100010a
Dec  6 15:30:19 erwin kernel: Destination Address is  fe00030a
Dec  6 15:30:19 erwin kernel: Packet Accepted  due to conditionSource Address is  fe00030a
Dec  6 15:30:19 erwin kernel: Destination Address is  100010a
Dec  6 15:30:19 erwin kernel: Packet Displayeddevice IP Address is  100040a
Dec  6 15:30:19 erwin kernel: Packet Displayeddevice IP Address is  fd00060a
Dec  6 15:30:20 erwin kernel: IN new connecetionPacket AcceptedSource Address is  100010a
Dec  6 15:30:20 erwin kernel: Destination Address is  fe00030a
Dec  6 15:30:20 erwin kernel: Packet Accepted  due to conditionSource Address is  fe00030a
Dec  6 15:30:20 erwin kernel: Destination Address is  100010a
Dec  6 15:30:21 erwin kernel: IN new connecetionPacket AcceptedSource Address is  100010a
Dec  6 15:30:21 erwin kernel: Destination Address is  fe00030a
Dec  6 15:30:21 erwin kernel: Packet Accepted  due to conditionSource Address is  fe00030a
Dec  6 15:30:21 erwin kernel: Destination Address is  100010a
Dec  6 15:30:22 erwin kernel: IN new connecetionPacket AcceptedSource Address is  100010a
Dec  6 15:30:22 erwin kernel: Destination Address is  fe00030a
Dec  6 15:30:22 erwin kernel: Packet Accepted  due to conditionSource Address is  fe00030a
Dec  6 15:30:22 erwin kernel: Destination Address is  100010a
Dec  6 15:30:23 erwin kernel: IN new connecetionPacket AcceptedSource Address is  200040a
Dec  6 15:30:23 erwin kernel: Destination Address is  fd00030a
Dec  6 15:30:23 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 15:30:23 erwin kernel: Destination Address is  200040a
Dec  6 15:30:23 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 15:30:23 erwin kernel: Destination Address is  fd00030a
Dec  6 15:30:23 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 15:30:23 erwin kernel: Destination Address is  fd00030a
Dec  6 15:30:23 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 15:30:23 erwin kernel: Destination Address is  fd00030a
Dec  6 15:30:23 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 15:30:23 erwin kernel: Destination Address is  fd00030a
Dec  6 15:30:23 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 15:30:23 erwin kernel: Destination Address is  200040a
Dec  6 15:30:23 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 15:30:23 erwin kernel: Destination Address is  200040a
Dec  6 15:30:23 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 15:30:23 erwin kernel: Destination Address is  200040a
Dec  6 15:30:23 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 15:30:23 erwin kernel: Destination Address is  fd00030a
Dec  6 15:30:23 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 15:30:23 erwin kernel: Destination Address is  200040a
Dec  6 15:30:23 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 15:30:23 erwin kernel: Destination Address is  200040a
Dec  6 15:30:23 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 15:30:43 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
```

Dec 6 15:30:43 erwin kernel: Destination Address is fd00030a
Dec 6 15:32:47 erwin ntpd[957]: time reset -0.280159 s
Dec 6 15:32:47 erwin ntpd[957]: kernel time discipline status change 41
Dec 6 15:32:47 erwin ntpd[957]: synchronisation lost
Dec 6 15:39:19 erwin ntpd[957]: kernel time discipline status change 1

In this we can see that the neighbor Chekov i.e. with the address 10.4.0.2 (in hex 200040a) is able to communicate successfully. This file also shows the fact that when Hubble i.e. With the address 10.1.0.1 (in hex 100010a) tried to communicate then an ICMP_ECHOREQUEST packet was sent and when the ICMP_ECHOREPLY was received the flow of ping packets to 10.3.0.254 was able to continue normally

This can also be proved by the ethereal outputs of the two machines which are given below

Ethereal output Chekov

No.	Time	Source	Destination	Protocol	Info
1	0.000000	10.4.0.2	10.3.0.253	TCP	47585 > ipp [SYN]
2	0.000470	10.3.0.253	10.4.0.2	TCP	ipp > 47585 [SYN, ACK]
3	0.000487	10.4.0.2	10.3.0.253	TCP	47585 > ipp [ACK]
4	0.000521	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
5	0.000527	10.4.0.2	10.3.0.253	HTTP	Continuation
6	0.000532	10.4.0.2	10.3.0.253	HTTP	Continuation
7	0.000941	10.3.0.253	10.4.0.2	TCP	ipp > 47585 [ACK]
8	0.000952	10.4.0.2	10.3.0.253	IPP	IPP request
9	0.000996	10.3.0.253	10.4.0.2	TCP	ipp > 47585 [ACK]
10	0.001009	10.3.0.253	10.4.0.2	TCP	ipp > 47585 [ACK]
11	0.002689	10.3.0.253	10.4.0.2	TCP	ipp > 47585 [ACK]
12	0.002938	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
13	0.002946	10.4.0.2	10.3.0.253	TCP	47585 > ipp [ACK]
14	0.003003	10.3.0.253	10.4.0.2	HTTP	Continuation
15	0.003004	10.3.0.253	10.4.0.2	HTTP	Continuation
16	0.003025	10.4.0.2	10.3.0.253	TCP	47585 > ipp [ACK]
17	0.003028	10.4.0.2	10.3.0.253	TCP	47585 > ipp [ACK]
18	0.004790	10.3.0.253	10.4.0.2	IPP	IPP response
19	0.004798	10.4.0.2	10.3.0.253	TCP	47585 > ipp [ACK]
20	0.004823	10.4.0.2	10.3.0.253	TCP	47585 > ipp [FIN, ACK]
21	0.005221	10.3.0.253	10.4.0.2	TCP	ipp > 47585 [FIN, ACK]
22	0.005230	10.4.0.2	10.3.0.253	TCP	47585 > ipp [ACK]
23	5.010014	10.4.0.2	10.3.0.253	TCP	47586 > ipp [SYN]
24	5.010497	10.3.0.253	10.4.0.2	TCP	ipp > 47586 [SYN, ACK]
25	5.010513	10.4.0.2	10.3.0.253	TCP	47586 > ipp [ACK]
26	5.010568	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
27	5.010574	10.4.0.2	10.3.0.253	HTTP	Continuation
28	5.010579	10.4.0.2	10.3.0.253	HTTP	Continuation
29	5.011046	10.3.0.253	10.4.0.2	TCP	ipp > 47586 [ACK]
30	5.011061	10.3.0.253	10.4.0.2	TCP	ipp > 47586 [ACK]
31	5.011062	10.3.0.253	10.4.0.2	TCP	ipp > 47586 [ACK]
32	5.011106	10.4.0.2	10.3.0.253	IPP	IPP request
33	5.012239	10.3.0.253	10.4.0.2	TCP	ipp > 47586 [ACK]
34	5.012550	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
35	5.012564	10.3.0.253	10.4.0.2	HTTP	Continuation
36	5.012582	10.3.0.253	10.4.0.2	HTTP	Continuation
37	5.012622	10.4.0.2	10.3.0.253	TCP	47586 > ipp [ACK]
38	5.012626	10.4.0.2	10.3.0.253	TCP	47586 > ipp [ACK]

39	5.012630	10.4.0.2	10.3.0.253	TCP	47586 > ipp [ACK]
40	5.014513	10.3.0.253	10.4.0.2	IPP	IPP response
41	5.014533	10.4.0.2	10.3.0.253	TCP	47586 > ipp [ACK]
42	5.014558	10.4.0.2	10.3.0.253	TCP	47586 > ipp [FIN, ACK]
43	5.014948	10.3.0.253	10.4.0.2	TCP	ipp > 47586 [FIN, ACK]
44	5.014957	10.4.0.2	10.3.0.253	TCP	47586 > ipp [ACK]
45	10.019997	10.4.0.2	10.3.0.253	TCP	47587 > ipp [SYN]
46	10.021185	10.3.0.253	10.4.0.2	TCP	ipp > 47587 [SYN, ACK]
47	10.021202	10.4.0.2	10.3.0.253	TCP	47587 > ipp [ACK]
48	10.021257	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
49	10.021263	10.4.0.2	10.3.0.253	HTTP	Continuation
50	10.021269	10.4.0.2	10.3.0.253	HTTP	Continuation
51	10.021711	10.3.0.253	10.4.0.2	TCP	ipp > 47587 [ACK]
52	10.021740	10.3.0.253	10.4.0.2	TCP	ipp > 47587 [ACK]
53	10.021749	10.4.0.2	10.3.0.253	IPP	IPP request
54	10.021762	10.3.0.253	10.4.0.2	TCP	ipp > 47587 [ACK]
55	10.023506	10.3.0.253	10.4.0.2	TCP	ipp > 47587 [ACK]
56	10.023779	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
57	10.023800	10.4.0.2	10.3.0.253	TCP	47587 > ipp [ACK]
58	10.023839	10.3.0.253	10.4.0.2	HTTP	Continuation
59	10.023840	10.3.0.253	10.4.0.2	HTTP	Continuation
60	10.023871	10.4.0.2	10.3.0.253	TCP	47587 > ipp [ACK]
61	10.023874	10.4.0.2	10.3.0.253	TCP	47587 > ipp [ACK]
62	10.025654	10.3.0.253	10.4.0.2	IPP	IPP response
63	10.025674	10.4.0.2	10.3.0.253	TCP	47587 > ipp [ACK]
64	10.025699	10.4.0.2	10.3.0.253	TCP	47587 > ipp [FIN, ACK]
65	10.026095	10.3.0.253	10.4.0.2	TCP	ipp > 47587 [FIN, ACK]
66	10.026104	10.4.0.2	10.3.0.253	TCP	47587 > ipp [ACK]
67	15.030087	10.4.0.2	10.3.0.253	TCP	47588 > ipp [SYN]
68	15.030563	10.3.0.253	10.4.0.2	TCP	ipp > 47588 [SYN, ACK]
69	15.030580	10.4.0.2	10.3.0.253	TCP	47588 > ipp [ACK]
70	15.030645	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
71	15.030651	10.4.0.2	10.3.0.253	HTTP	Continuation
72	15.030656	10.4.0.2	10.3.0.253	HTTP	Continuation
73	15.032067	10.3.0.253	10.4.0.2	TCP	ipp > 47588 [ACK]
74	15.032090	10.4.0.2	10.3.0.253	IPP	IPP request
75	15.032117	10.3.0.253	10.4.0.2	TCP	ipp > 47588 [ACK]
76	15.032135	10.3.0.253	10.4.0.2	TCP	ipp > 47588 [ACK]
77	15.033838	10.3.0.253	10.4.0.2	TCP	ipp > 47588 [ACK]
78	15.034090	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
79	15.034112	10.4.0.2	10.3.0.253	TCP	47588 > ipp [ACK]
80	15.034169	10.3.0.253	10.4.0.2	HTTP	Continuation
81	15.034172	10.3.0.253	10.4.0.2	HTTP	Continuation
82	15.034203	10.4.0.2	10.3.0.253	TCP	47588 > ipp [ACK]
83	15.034206	10.4.0.2	10.3.0.253	TCP	47588 > ipp [ACK]
84	15.035971	10.3.0.253	10.4.0.2	IPP	IPP response
85	15.035991	10.4.0.2	10.3.0.253	TCP	47588 > ipp [ACK]
86	15.036016	10.4.0.2	10.3.0.253	TCP	47588 > ipp [FIN, ACK]
87	15.036419	10.3.0.253	10.4.0.2	TCP	ipp > 47588 [FIN, ACK]
88	15.036428	10.4.0.2	10.3.0.253	TCP	47588 > ipp [ACK]
89	20.040002	10.4.0.2	10.3.0.253	TCP	47589 > ipp [SYN]
90	20.040479	10.3.0.253	10.4.0.2	TCP	ipp > 47589 [SYN, ACK]
91	20.040497	10.4.0.2	10.3.0.253	TCP	47589 > ipp [ACK]
92	20.040552	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
93	20.040558	10.4.0.2	10.3.0.253	HTTP	Continuation
94	20.040564	10.4.0.2	10.3.0.253	HTTP	Continuation

95	20.040995	10.3.0.253	10.4.0.2	TCP	ipp > 47589 [ACK]
96	20.041009	10.3.0.253	10.4.0.2	TCP	ipp > 47589 [ACK]
97	20.041023	10.3.0.253	10.4.0.2	TCP	ipp > 47589 [ACK]
98	20.041066	10.4.0.2	10.3.0.253	IPP	IPP request
99	20.042218	10.3.0.253	10.4.0.2	TCP	ipp > 47589 [ACK]
100	20.042584	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
101	20.042606	10.4.0.2	10.3.0.253	TCP	47589 > ipp [ACK]
102	20.042633	10.3.0.253	10.4.0.2	HTTP	Continuation
103	20.042646	10.3.0.253	10.4.0.2	HTTP	Continuation
104	20.042681	10.4.0.2	10.3.0.253	TCP	47589 > ipp [ACK]
105	20.042685	10.4.0.2	10.3.0.253	TCP	47589 > ipp [ACK]
106	20.044478	10.3.0.253	10.4.0.2	IPP	IPP response
107	20.044498	10.4.0.2	10.3.0.253	TCP	47589 > ipp [ACK]
108	20.044524	10.4.0.2	10.3.0.253	TCP	47589 > ipp [FIN, ACK]
109	20.044995	10.3.0.253	10.4.0.2	TCP	ipp > 47589 [FIN, ACK]
110	20.045004	10.4.0.2	10.3.0.253	TCP	47589 > ipp [ACK]
111	21.074693	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
112	21.074895	10.4.0.1	10.1.0.1	ICMP	Echo (ping) request
113	21.075160	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
114	21.075219	10.1.0.1	10.4.0.1	ICMP	Echo (ping) reply
115	22.073566	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
116	22.073978	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
117	23.072452	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
118	23.072871	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
119	24.071344	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
120	24.071758	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
121	25.050005	10.4.0.2	10.3.0.253	TCP	47590 > ipp [SYN]
122	25.050470	10.3.0.253	10.4.0.2	TCP	ipp > 47590 [SYN, ACK]
123	25.050487	10.4.0.2	10.3.0.253	TCP	47590 > ipp [ACK]
124	25.050543	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
125	25.050550	10.4.0.2	10.3.0.253	HTTP	Continuation
126	25.050555	10.4.0.2	10.3.0.253	HTTP	Continuation
127	25.050978	10.3.0.253	10.4.0.2	TCP	ipp > 47590 [ACK]
128	25.050995	10.3.0.253	10.4.0.2	TCP	ipp > 47590 [ACK]
129	25.051024	10.3.0.253	10.4.0.2	TCP	ipp > 47590 [ACK]
130	25.051050	10.4.0.2	10.3.0.253	IPP	IPP request
131	25.052188	10.3.0.253	10.4.0.2	TCP	ipp > 47590 [ACK]
132	25.052439	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
133	25.052460	10.4.0.2	10.3.0.253	TCP	47590 > ipp [ACK]
134	25.052511	10.3.0.253	10.4.0.2	HTTP	Continuation
135	25.052512	10.3.0.253	10.4.0.2	HTTP	Continuation
136	25.052543	10.4.0.2	10.3.0.253	TCP	47590 > ipp [ACK]
137	25.052546	10.4.0.2	10.3.0.253	TCP	47590 > ipp [ACK]
138	25.054316	10.3.0.253	10.4.0.2	IPP	IPP response
139	25.054336	10.4.0.2	10.3.0.253	TCP	47590 > ipp [ACK]
140	25.054361	10.4.0.2	10.3.0.253	TCP	47590 > ipp [FIN, ACK]
141	25.054752	10.3.0.253	10.4.0.2	TCP	ipp > 47590 [FIN, ACK]
142	25.054761	10.4.0.2	10.3.0.253	TCP	47590 > ipp [ACK]
143	25.070224	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
144	25.070585	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
145	26.069792	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
146	26.070202	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
147	27.069677	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
148	27.070099	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
149	28.069571	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
150	28.069991	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply

151	29.069452	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
152	29.069863	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
153	30.060001	10.4.0.2	10.3.0.253	TCP	47591 > ipp [SYN]
154	30.060473	10.3.0.253	10.4.0.2	TCP	ipp > 47591 [SYN, ACK]
155	30.060490	10.4.0.2	10.3.0.253	TCP	47591 > ipp [ACK]
156	30.060558	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
157	30.060564	10.4.0.2	10.3.0.253	HTTP	Continuation
158	30.060569	10.4.0.2	10.3.0.253	HTTP	Continuation
159	30.060991	10.3.0.253	10.4.0.2	TCP	ipp > 47591 [ACK]
160	30.061004	10.3.0.253	10.4.0.2	TCP	ipp > 47591 [ACK]
161	30.061024	10.3.0.253	10.4.0.2	TCP	ipp > 47591 [ACK]
162	30.061062	10.4.0.2	10.3.0.253	IPP	IPP request
163	30.062197	10.3.0.253	10.4.0.2	TCP	ipp > 47591 [ACK]
164	30.062441	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
165	30.062462	10.4.0.2	10.3.0.253	TCP	47591 > ipp [ACK]
166	30.062524	10.3.0.253	10.4.0.2	HTTP	Continuation
167	30.062525	10.3.0.253	10.4.0.2	HTTP	Continuation
168	30.062556	10.4.0.2	10.3.0.253	TCP	47591 > ipp [ACK]
169	30.062559	10.4.0.2	10.3.0.253	TCP	47591 > ipp [ACK]
170	30.064306	10.3.0.253	10.4.0.2	IPP	IPP response
171	30.064326	10.4.0.2	10.3.0.253	TCP	47591 > ipp [ACK]
172	30.064363	10.4.0.2	10.3.0.253	TCP	47591 > ipp [FIN, ACK]
173	30.064759	10.3.0.253	10.4.0.2	TCP	ipp > 47591 [FIN, ACK]
174	30.064768	10.4.0.2	10.3.0.253	TCP	47591 > ipp [ACK]
175	30.069299	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
176	30.069656	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
177	31.069192	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
178	31.069609	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
179	32.069081	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
180	32.069497	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
181	33.068968	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
182	33.069379	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
183	34.068859	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
184	34.069268	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
185	35.068746	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
186	35.069162	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
187	35.070001	10.4.0.2	10.3.0.253	TCP	47592 > ipp [SYN]
188	35.070430	10.3.0.253	10.4.0.2	TCP	ipp > 47592 [SYN, ACK]
189	35.070445	10.4.0.2	10.3.0.253	TCP	47592 > ipp [ACK]
190	35.070499	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
191	35.070505	10.4.0.2	10.3.0.253	HTTP	Continuation
192	35.070510	10.4.0.2	10.3.0.253	HTTP	Continuation
193	35.070937	10.3.0.253	10.4.0.2	TCP	ipp > 47592 [ACK]
194	35.070952	10.3.0.253	10.4.0.2	TCP	ipp > 47592 [ACK]
195	35.070975	10.3.0.253	10.4.0.2	TCP	ipp > 47592 [ACK]
196	35.071010	10.4.0.2	10.3.0.253	IPP	IPP request
197	35.072142	10.3.0.253	10.4.0.2	TCP	ipp > 47592 [ACK]
198	35.072391	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
199	35.072413	10.4.0.2	10.3.0.253	TCP	47592 > ipp [ACK]
200	35.072468	10.3.0.253	10.4.0.2	HTTP	Continuation
201	35.072469	10.3.0.253	10.4.0.2	HTTP	Continuation
202	35.072512	10.4.0.2	10.3.0.253	TCP	47592 > ipp [ACK]
203	35.072516	10.4.0.2	10.3.0.253	TCP	47592 > ipp [ACK]
204	35.074264	10.3.0.253	10.4.0.2	IPP	IPP response
205	35.074284	10.4.0.2	10.3.0.253	TCP	47592 > ipp [ACK]
206	35.074309	10.4.0.2	10.3.0.253	TCP	47592 > ipp [FIN, ACK]

207	35.074700	10.3.0.253	10.4.0.2	TCP	ipp > 47592 [FIN, ACK]
208	35.074709	10.4.0.2	10.3.0.253	TCP	47592 > ipp [ACK]
209	36.068636	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
210	36.069048	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
211	37.068540	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
212	37.068951	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
213	38.068413	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
214	38.068830	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
215	40.080080	10.4.0.2	10.3.0.253	TCP	47593 > ipp [SYN]
216	40.080555	10.3.0.253	10.4.0.2	TCP	ipp > 47593 [SYN, ACK]
217	40.080578	10.4.0.2	10.3.0.253	TCP	47593 > ipp [ACK]
218	40.080896	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
219	40.080903	10.4.0.2	10.3.0.253	HTTP	Continuation
220	40.080909	10.4.0.2	10.3.0.253	HTTP	Continuation
221	40.082318	10.3.0.253	10.4.0.2	TCP	ipp > 47593 [ACK]
222	40.082351	10.3.0.253	10.4.0.2	TCP	ipp > 47593 [ACK]
223	40.082373	10.3.0.253	10.4.0.2	TCP	ipp > 47593 [ACK]
224	40.082397	10.4.0.2	10.3.0.253	IPP	IPP request
225	40.083535	10.3.0.253	10.4.0.2	TCP	ipp > 47593 [ACK]
226	40.084777	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
227	40.084798	10.4.0.2	10.3.0.253	TCP	47593 > ipp [ACK]
228	40.084830	10.3.0.253	10.4.0.2	HTTP	Continuation
229	40.084832	10.3.0.253	10.4.0.2	HTTP	Continuation
230	40.084877	10.4.0.2	10.3.0.253	TCP	47593 > ipp [ACK]
231	40.084880	10.4.0.2	10.3.0.253	TCP	47593 > ipp [ACK]
232	40.086637	10.3.0.253	10.4.0.2	IPP	IPP response
233	40.086657	10.4.0.2	10.3.0.253	TCP	47593 > ipp [ACK]
234	40.086684	10.4.0.2	10.3.0.253	TCP	47593 > ipp [FIN, ACK]
235	40.087065	10.3.0.253	10.4.0.2	TCP	ipp > 47593 [FIN, ACK]
236	40.087075	10.4.0.2	10.3.0.253	TCP	47593 > ipp [ACK]
237	41.063901	Intel_d3:cb:73	Intel_d3:d2:a0	ARP	Who has 10.4.0.2? Tell
238	41.063916	Intel_d3:d2:a0	Intel_d3:cb:73	ARP	10.4.0.2 is at 00:02:b3
239	44.231539	Intel_d3:cb:73	Broadcast	ARP	Who has 10.4.0.16? Tel
240	45.090003	10.4.0.2	10.3.0.253	TCP	47594 > ipp [SYN]
241	45.090443	10.3.0.253	10.4.0.2	TCP	ipp > 47594 [SYN, ACK]
242	45.090460	10.4.0.2	10.3.0.253	TCP	47594 > ipp [ACK]
243	45.090516	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
244	45.090523	10.4.0.2	10.3.0.253	HTTP	Continuation
245	45.090528	10.4.0.2	10.3.0.253	HTTP	Continuation
246	45.090941	10.3.0.253	10.4.0.2	TCP	ipp > 47594 [ACK]
247	45.090963	10.4.0.2	10.3.0.253	IPP	IPP request
248	45.091001	10.3.0.253	10.4.0.2	TCP	ipp > 47594 [ACK]
249	45.091015	10.3.0.253	10.4.0.2	TCP	ipp > 47594 [ACK]
250	45.092704	10.3.0.253	10.4.0.2	TCP	ipp > 47594 [ACK]
251	45.093947	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
252	45.093969	10.4.0.2	10.3.0.253	TCP	47594 > ipp [ACK]
253	45.094015	10.3.0.253	10.4.0.2	HTTP	Continuation
254	45.094016	10.3.0.253	10.4.0.2	HTTP	Continuation
255	45.094047	10.4.0.2	10.3.0.253	TCP	47594 > ipp [ACK]
256	45.094051	10.4.0.2	10.3.0.253	TCP	47594 > ipp [ACK]
257	45.095812	10.3.0.253	10.4.0.2	IPP	IPP response
258	45.095832	10.4.0.2	10.3.0.253	TCP	47594 > ipp [ACK]
259	45.095857	10.4.0.2	10.3.0.253	TCP	47594 > ipp [FIN, ACK]
260	45.096255	10.3.0.253	10.4.0.2	TCP	ipp > 47594 [FIN, ACK]
261	45.096265	10.4.0.2	10.3.0.253	TCP	47594 > ipp [ACK]

262 45.223422 Intel_d3:cb:73 Broadcast ARP Who has 10.4.0.16? Tell
10.4.0.1
263 45.741099 Hewlett-_43:2e:a6 CDP/VTP CDP Cisco Discovery Protocol
264 46.223307 Intel_d3:cb:73 Broadcast ARP Who has 10.4.0.16? Tell
10.4.0.1
265 50.100004 10.4.0.2 10.3.0.253 TCP 47595 > ipp [SYN]
266 50.100446 10.3.0.253 10.4.0.2 TCP ipp > 47595 [SYN, ACK]
267 50.100467 10.4.0.2 10.3.0.253 TCP 47595 > ipp [ACK]
268 50.100812 10.4.0.2 10.3.0.253 HTTP POST / HTTP/1.1
269 50.100819 10.4.0.2 10.3.0.253 HTTP Continuation
270 50.100825 10.4.0.2 10.3.0.253 HTTP Continuation
271 50.101237 10.3.0.253 10.4.0.2 TCP ipp > 47595 [ACK]
272 50.101238 10.3.0.253 10.4.0.2 TCP ipp > 47595 [ACK]
273 50.101252 10.3.0.253 10.4.0.2 TCP ipp > 47595 [ACK]
274 50.101300 10.4.0.2 10.3.0.253 IPP IPP request
275 50.102420 10.3.0.253 10.4.0.2 TCP ipp > 47595 [ACK]
276 50.102672 10.3.0.253 10.4.0.2 HTTP HTTP/1.1 200 OK
277 50.102695 10.4.0.2 10.3.0.253 TCP 47595 > ipp [ACK]
278 50.102742 10.3.0.253 10.4.0.2 HTTP Continuation
279 50.102743 10.3.0.253 10.4.0.2 HTTP Continuation
280 50.102785 10.4.0.2 10.3.0.253 TCP 47595 > ipp [ACK]
281 50.102788 10.4.0.2 10.3.0.253 TCP 47595 > ipp [ACK]
282 50.104526 10.3.0.253 10.4.0.2 IPP IPP response
283 50.104548 10.4.0.2 10.3.0.253 TCP 47595 > ipp [ACK]
284 50.104576 10.4.0.2 10.3.0.253 TCP 47595 > ipp [FIN, ACK]
285 50.104951 10.3.0.253 10.4.0.2 TCP ipp > 47595 [FIN, ACK]
286 50.104960 10.4.0.2 10.3.0.253 TCP 47595 > ipp [ACK]
287 55.110001 10.4.0.2 10.3.0.253 TCP 47596 > ipp [SYN]
288 55.110454 10.3.0.253 10.4.0.2 TCP ipp > 47596 [SYN, ACK]
289 55.110477 10.4.0.2 10.3.0.253 TCP 47596 > ipp [ACK]
290 55.110791 10.4.0.2 10.3.0.253 HTTP POST / HTTP/1.1
291 55.110798 10.4.0.2 10.3.0.253 HTTP Continuation
292 55.110804 10.4.0.2 10.3.0.253 HTTP Continuation
293 55.111160 10.3.0.253 10.4.0.2 TCP ipp > 47596 [ACK]
294 55.111187 10.4.0.2 10.3.0.253 IPP IPP request
295 55.111221 10.3.0.253 10.4.0.2 TCP ipp > 47596 [ACK]
296 55.111222 10.3.0.253 10.4.0.2 TCP ipp > 47596 [ACK]
297 55.112898 10.3.0.253 10.4.0.2 TCP ipp > 47596 [ACK]
298 55.113168 10.3.0.253 10.4.0.2 HTTP HTTP/1.1 200 OK
299 55.113189 10.4.0.2 10.3.0.253 TCP 47596 > ipp [ACK]
300 55.113238 10.3.0.253 10.4.0.2 HTTP Continuation
301 55.113239 10.3.0.253 10.4.0.2 HTTP Continuation
302 55.113281 10.4.0.2 10.3.0.253 TCP 47596 > ipp [ACK]
303 55.113285 10.4.0.2 10.3.0.253 TCP 47596 > ipp [ACK]
304 55.115022 10.3.0.253 10.4.0.2 IPP IPP response
305 55.115041 10.4.0.2 10.3.0.253 TCP 47596 > ipp [ACK]
306 55.115068 10.4.0.2 10.3.0.253 TCP 47596 > ipp [FIN, ACK]
307 55.115452 10.3.0.253 10.4.0.2 TCP ipp > 47596 [FIN, ACK]
308 55.115462 10.4.0.2 10.3.0.253 TCP 47596 > ipp [ACK]
309 60.110005 10.4.0.2 10.3.0.253 TCP 47597 > ipp [SYN]
310 60.110463 10.3.0.253 10.4.0.2 TCP ipp > 47597 [SYN, ACK]
311 60.110481 10.4.0.2 10.3.0.253 TCP 47597 > ipp [ACK]
312 60.110536 10.4.0.2 10.3.0.253 HTTP POST / HTTP/1.1
313 60.110542 10.4.0.2 10.3.0.253 HTTP Continuation
314 60.110548 10.4.0.2 10.3.0.253 HTTP Continuation
315 60.111891 10.3.0.253 10.4.0.2 TCP ipp > 47597 [ACK]

316	60.111913	10.4.0.2	10.3.0.253	IPP	IPP request
317	60.111951	10.3.0.253	10.4.0.2	TCP	ipp > 47597 [ACK]
318	60.111952	10.3.0.253	10.4.0.2	TCP	ipp > 47597 [ACK]
319	60.113628	10.3.0.253	10.4.0.2	TCP	ipp > 47597 [ACK]
320	60.114889	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
321	60.114910	10.4.0.2	10.3.0.253	TCP	47597 > ipp [ACK]
322	60.114962	10.3.0.253	10.4.0.2	HTTP	Continuation
323	60.114963	10.3.0.253	10.4.0.2	HTTP	Continuation
324	60.114994	10.4.0.2	10.3.0.253	TCP	47597 > ipp [ACK]
325	60.114998	10.4.0.2	10.3.0.253	TCP	47597 > ipp [ACK]
326	60.116737	10.3.0.253	10.4.0.2	IPP	IPP response
327	60.116757	10.4.0.2	10.3.0.253	TCP	47597 > ipp [ACK]
328	60.116783	10.4.0.2	10.3.0.253	TCP	47597 > ipp [FIN, ACK]
329	60.117141	10.3.0.253	10.4.0.2	TCP	ipp > 47597 [FIN, ACK]
330	60.117150	10.4.0.2	10.3.0.253	TCP	47597 > ipp [ACK]
331	65.120086	10.4.0.2	10.3.0.253	TCP	47598 > ipp [SYN]
332	65.120544	10.3.0.253	10.4.0.2	TCP	ipp > 47598 [SYN, ACK]
333	65.120561	10.4.0.2	10.3.0.253	TCP	47598 > ipp [ACK]
334	65.120626	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
335	65.120632	10.4.0.2	10.3.0.253	HTTP	Continuation
336	65.120638	10.4.0.2	10.3.0.253	HTTP	Continuation
337	65.121006	10.3.0.253	10.4.0.2	TCP	ipp > 47598 [ACK]
338	65.121028	10.4.0.2	10.3.0.253	IPP	IPP request
339	65.121053	10.3.0.253	10.4.0.2	TCP	ipp > 47598 [ACK]
340	65.121070	10.3.0.253	10.4.0.2	TCP	ipp > 47598 [ACK]
341	65.122749	10.3.0.253	10.4.0.2	TCP	ipp > 47598 [ACK]
342	65.123986	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
343	65.124007	10.4.0.2	10.3.0.253	TCP	47598 > ipp [ACK]
344	65.124058	10.3.0.253	10.4.0.2	HTTP	Continuation
345	65.124059	10.3.0.253	10.4.0.2	HTTP	Continuation
346	65.124101	10.4.0.2	10.3.0.253	TCP	47598 > ipp [ACK]
347	65.124105	10.4.0.2	10.3.0.253	TCP	47598 > ipp [ACK]
348	65.125829	10.3.0.253	10.4.0.2	IPP	IPP response
349	65.125848	10.4.0.2	10.3.0.253	TCP	47598 > ipp [ACK]
350	65.125873	10.4.0.2	10.3.0.253	TCP	47598 > ipp [FIN, ACK]
351	65.126252	10.3.0.253	10.4.0.2	TCP	ipp > 47598 [FIN, ACK]
352	65.126262	10.4.0.2	10.3.0.253	TCP	47598 > ipp [ACK]
353	70.130002	10.4.0.2	10.3.0.253	TCP	47599 > ipp [SYN]
354	70.130413	10.3.0.253	10.4.0.2	TCP	ipp > 47599 [SYN, ACK]
355	70.130430	10.4.0.2	10.3.0.253	TCP	47599 > ipp [ACK]
356	70.130485	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
357	70.130491	10.4.0.2	10.3.0.253	HTTP	Continuation
358	70.130496	10.4.0.2	10.3.0.253	HTTP	Continuation
359	70.130889	10.3.0.253	10.4.0.2	TCP	ipp > 47599 [ACK]
360	70.130903	10.3.0.253	10.4.0.2	TCP	ipp > 47599 [ACK]
361	70.130927	10.3.0.253	10.4.0.2	TCP	ipp > 47599 [ACK]
362	70.130960	10.4.0.2	10.3.0.253	IPP	IPP request
363	70.132072	10.3.0.253	10.4.0.2	TCP	ipp > 47599 [ACK]
364	70.132325	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
365	70.132346	10.4.0.2	10.3.0.253	TCP	47599 > ipp [ACK]
366	70.132397	10.3.0.253	10.4.0.2	HTTP	Continuation
367	70.132399	10.3.0.253	10.4.0.2	HTTP	Continuation
368	70.132429	10.4.0.2	10.3.0.253	TCP	47599 > ipp [ACK]
369	70.132433	10.4.0.2	10.3.0.253	TCP	47599 > ipp [ACK]
370	70.134178	10.3.0.253	10.4.0.2	IPP	IPP response
371	70.134609	10.4.0.2	10.3.0.253	TCP	47599 > ipp [ACK]

372	70.134646	10.4.0.2	10.3.0.253	TCP	47599 > ipp [FIN, ACK]
373	70.135031	10.3.0.253	10.4.0.2	TCP	ipp > 47599 [FIN, ACK]
374	70.135048	10.4.0.2	10.3.0.253	TCP	47599 > ipp [ACK]
375	75.140005	10.4.0.2	10.3.0.253	TCP	47600 > ipp [SYN]
376	75.140432	10.3.0.253	10.4.0.2	TCP	ipp > 47600 [SYN, ACK]
377	75.140453	10.4.0.2	10.3.0.253	TCP	47600 > ipp [ACK]
378	75.146037	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
379	75.146047	10.4.0.2	10.3.0.253	HTTP	Continuation
380	75.146052	10.4.0.2	10.3.0.253	HTTP	Continuation
381	75.146406	10.3.0.253	10.4.0.2	TCP	ipp > 47600 [ACK]
382	75.146461	10.3.0.253	10.4.0.2	TCP	ipp > 47600 [ACK]
383	75.146475	10.3.0.253	10.4.0.2	TCP	ipp > 47600 [ACK]
384	75.146645	10.4.0.2	10.3.0.253	IPP	IPP request
385	75.147758	10.3.0.253	10.4.0.2	TCP	ipp > 47600 [ACK]
386	75.149000	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
387	75.149022	10.4.0.2	10.3.0.253	TCP	47600 > ipp [ACK]
388	75.149075	10.3.0.253	10.4.0.2	HTTP	Continuation
389	75.149076	10.3.0.253	10.4.0.2	HTTP	Continuation
390	75.149107	10.4.0.2	10.3.0.253	TCP	47600 > ipp [ACK]
391	75.149111	10.4.0.2	10.3.0.253	TCP	47600 > ipp [ACK]
392	75.150842	10.3.0.253	10.4.0.2	IPP	IPP response
393	75.150866	10.4.0.2	10.3.0.253	TCP	47600 > ipp [ACK]
394	75.150896	10.4.0.2	10.3.0.253	TCP	47600 > ipp [FIN, ACK]
395	75.151252	10.3.0.253	10.4.0.2	TCP	ipp > 47600 [FIN, ACK]
396	75.151262	10.4.0.2	10.3.0.253	TCP	47600 > ipp [ACK]
397	80.150010	10.4.0.2	10.3.0.253	TCP	47601 > ipp [SYN]
398	80.150449	10.3.0.253	10.4.0.2	TCP	ipp > 47601 [SYN, ACK]
399	80.150471	10.4.0.2	10.3.0.253	TCP	47601 > ipp [ACK]
400	80.150631	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
401	80.150637	10.4.0.2	10.3.0.253	HTTP	Continuation
402	80.150642	10.4.0.2	10.3.0.253	HTTP	Continuation
403	80.151037	10.3.0.253	10.4.0.2	TCP	ipp > 47601 [ACK]
404	80.151051	10.3.0.253	10.4.0.2	TCP	ipp > 47601 [ACK]
405	80.151076	10.3.0.253	10.4.0.2	TCP	ipp > 47601 [ACK]
406	80.151127	10.4.0.2	10.3.0.253	IPP	IPP request
407	80.152245	10.3.0.253	10.4.0.2	TCP	ipp > 47601 [ACK]
408	80.153482	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
409	80.153549	10.3.0.253	10.4.0.2	HTTP	Continuation
410	80.153550	10.3.0.253	10.4.0.2	HTTP	Continuation
411	80.154770	10.4.0.2	10.3.0.253	TCP	47601 > ipp [ACK]
412	80.154775	10.4.0.2	10.3.0.253	TCP	47601 > ipp [ACK]
413	80.154779	10.4.0.2	10.3.0.253	TCP	47601 > ipp [ACK]
414	80.156605	10.3.0.253	10.4.0.2	IPP	IPP response
415	80.159686	10.4.0.2	10.3.0.253	TCP	47601 > ipp [ACK]
416	80.159722	10.4.0.2	10.3.0.253	TCP	47601 > ipp [FIN, ACK]
417	80.160111	10.3.0.253	10.4.0.2	TCP	ipp > 47601 [FIN, ACK]
418	80.160131	10.4.0.2	10.3.0.253	TCP	47601 > ipp [ACK]

Ethereal Output Hubble

No.	Time	Source	Destination	Protocol	Info
1	0.000000	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
2	0.000352	Intel_d3:d2:9f	Broadcast	ARP	Who has 10.1.0.1? Tell 10.1.0.2
3	0.000369	Intel_d3:d2:79	Intel_d3:d2:9f	ARP	10.1.0.1 is at 00:02:b3:d3:d2:79
4	0.000488	10.4.0.1	10.1.0.1	ICMP	Echo (ping) request
5	0.000546	10.1.0.1	10.4.0.1	ICMP	Echo (ping) reply
6	0.000615	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
7	0.998998	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
8	0.999547	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
9	1.997999	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
10	1.998552	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
11	2.996999	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
12	2.997550	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
13	3.996001	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
14	3.996490	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
15	4.995672	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
16	4.996219	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
17	5.995671	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
18	5.996233	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
19	6.995676	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
20	6.996232	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
21	7.995671	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
22	7.996218	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
23	8.995636	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
24	8.996121	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
25	9.995635	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
26	9.996189	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
27	10.995635	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
28	10.996188	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
29	11.995635	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
30	11.996183	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
31	12.995637	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
32	12.996184	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
33	13.995637	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
34	13.996191	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
35	14.995638	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
36	14.996189	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
37	15.995655	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
38	15.996206	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
39	16.995642	10.1.0.1	10.3.0.254	ICMP	Echo (ping) request
40	16.996196	10.3.0.254	10.1.0.1	ICMP	Echo (ping) reply
41	24.667435	Hewlett-_43:2e:a2	CDP/VTP	CDP	Cisco Discovery Protocol
42	84.670086	Hewlett-_43:2e:a2	CDP/VTP	CDP	Cisco Discovery Protocol

Thus as seen above from the ethereal outputs Chekov was able to transmit successfully and Hubble when it first transmitted received an ICMP_ECHOREQUEST packet from Erwin 10.4.0.1 and after replying to that packet was able to transmit further packets successfully Thus test cases 2 and 4 were successfully tested

Test Case Three

We will now verify test case three by showing that a spoofing neighbors packets are dropped

We will first see Erwin's Kernel messages log file

/var/log/messages

```
Dec  6 17:40:31 erwin syslogd 1.4.1: restart.
Dec  6 17:40:31 erwin syslog: syslogd startup succeeded
Dec  6 17:40:31 erwin kernel: klogd 1.4.1, log source = /proc/kmsg started.
Dec  6 17:40:31 erwin syslog: klogd startup succeeded
Dec  6 17:41:41 erwin kernel: IN new connecetionPacket DroppedSource Address is  1000d0a
Dec  6 17:41:41 erwin kernel: Destination Address is  fd00030a
Dec  6 17:41:53 erwin kernel: IN new connecetionPacket DroppedSource Address is  1000d0a
Dec  6 17:41:53 erwin kernel: Destination Address is  fd00030a
Dec  6 17:42:26 erwin kernel: IN new connecetionPacket DroppedSource Address is  1000d0a
Dec  6 17:42:26 erwin kernel: Destination Address is  fe00030a
Dec  6 17:42:27 erwin kernel: IN new connecetionPacket DroppedSource Address is  1000d0a
Dec  6 17:42:27 erwin kernel: Destination Address is  fe00030a
Dec  6 17:42:28 erwin kernel: IN new connecetionPacket DroppedSource Address is  1000d0a
Dec  6 17:42:28 erwin kernel: Destination Address is  fe00030a
Dec  6 17:42:29 erwin kernel: IN new connecetionPacket DroppedSource Address is  1000d0a
Dec  6 17:42:29 erwin kernel: Destination Address is  fe00030a
```

This shows that in case the neighbor is spoofing the packets are successfully dropped by the module

This is further proved by the ethereal file of Chekov

Ethereal Output Chekov

No.	Time	Source	Destination	Protocol	Info
1	0.000000	10.13.0.1	10.3.0.253	TCP	47927 > ipp [SYN]
2	5.999948	10.13.0.1	10.3.0.253	TCP	47927 > ipp [SYN]
3	17.999999	10.13.0.1	10.3.0.253	TCP	47927 > ipp [SYN]
4	41.999950	10.13.0.1	10.3.0.253	TCP	47927 > ipp [SYN]
5	46.999947	Intel_d3:d2:a0		ARP	Who has 10.4.0.1? Tell 10.4.0.2
6	47.000092	Intel_d3:cb:73		ARP	10.4.0.1 is at 00:02:b3:d3:cb:73

Test Case Five

The Test case would verify that a non existent address on the network is not able to pass through the safe router

We will first see Erwin's Kernel messages log file

/var/log/messages

```
Dec  6 18:18:36 erwin syslogd 1.4.1: restart.
Dec  6 18:18:36 erwin syslog: syslogd startup succeeded
Dec  6 18:18:36 erwin kernel: klogd 1.4.1, log source = /proc/kmsg started.
Dec  6 18:18:36 erwin syslog: klogd startup succeeded
Dec  6 18:18:36 erwin syslog: syslogd shutdown succeeded
Dec  6 18:19:31 erwin kernel: IN new connecetionPacket AcceptedSource Address is  200040a
Dec  6 18:19:31 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:31 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:31 erwin kernel: Destination Address is  200040a
Dec  6 18:19:31 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:31 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:31 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:31 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:31 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:31 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:31 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:31 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:31 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:31 erwin kernel: Destination Address is  200040a
Dec  6 18:19:31 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:31 erwin kernel: Destination Address is  200040a
Dec  6 18:19:31 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:31 erwin kernel: Destination Address is  200040a
Dec  6 18:19:31 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:31 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:31 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:31 erwin kernel: Destination Address is  200040a
Dec  6 18:19:31 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:31 erwin kernel: Destination Address is  200040a
Dec  6 18:19:31 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:31 erwin kernel: Destination Address is  200040a
Dec  6 18:19:31 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:31 erwin kernel: Destination Address is  200040a
Dec  6 18:19:31 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:31 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:31 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:31 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:31 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:31 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:31 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:31 erwin kernel: Destination Address is  200040a
Dec  6 18:19:31 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:31 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:31 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:31 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:31 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:31 erwin kernel: Destination Address is  200040a
Dec  6 18:19:31 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:31 erwin kernel: Destination Address is  fd00030a
```

```
Dec  6 18:19:31 erwin kernel: IN new connecetionIN new ICMP send
Dec  6 18:19:31 erwin kernel:
Dec  6 18:19:31 erwin kernel:
Dec  6 18:19:31 erwin kernel: Packet AcceptedSource Address is  fa00030a
Dec  6 18:19:31 erwin kernel: Destination Address is  1000040a
Dec  6 18:19:31 erwin kernel: Packet Displayeddevice IP Address is  fd00060a
Dec  6 18:19:31 erwin kernel: Packet Displayeddevice IP Address is  100040a
Dec  6 18:19:34 erwin kernel: IN new connecetionIN new ICMP send
Dec  6 18:19:34 erwin kernel:
Dec  6 18:19:34 erwin kernel:
Dec  6 18:19:34 erwin kernel: Packet AcceptedSource Address is  1000d0b
Dec  6 18:19:34 erwin kernel: Destination Address is  fe00030a
Dec  6 18:19:35 erwin kernel: IN new connecetionPacket AcceptedSource Address is  1000d0b
Dec  6 18:19:35 erwin kernel: Destination Address is  fe00030a
Dec  6 18:19:36 erwin kernel: IN new connecetionPacket AcceptedSource Address is  200040a
Dec  6 18:19:36 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:36 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:36 erwin kernel: Destination Address is  200040a
Dec  6 18:19:36 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:36 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:36 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:36 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:36 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:36 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:36 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:36 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:36 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:36 erwin kernel: Destination Address is  200040a
Dec  6 18:19:36 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:36 erwin kernel: Destination Address is  200040a
Dec  6 18:19:36 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:36 erwin kernel: Destination Address is  200040a
Dec  6 18:19:36 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:36 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:36 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:36 erwin kernel: Destination Address is  200040a
Dec  6 18:19:36 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:36 erwin kernel: Destination Address is  200040a
Dec  6 18:19:36 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:36 erwin kernel: Destination Address is  200040a
Dec  6 18:19:36 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:36 erwin kernel: Destination Address is  200040a
Dec  6 18:19:36 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:36 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:36 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:36 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:36 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:36 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:36 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:36 erwin kernel: Destination Address is  200040a
Dec  6 18:19:36 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:36 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:36 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:36 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:36 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:36 erwin kernel: Destination Address is  200040a
```

Dec 6 18:19:36 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:36 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:36 erwin kernel: IN new connecetionPacket AcceptedSource Address is 1000d0b
Dec 6 18:19:36 erwin kernel: Destination Address is fe00030a
Dec 6 18:19:37 erwin kernel: IN new connecetionPacket AcceptedSource Address is 1000d0b
Dec 6 18:19:37 erwin kernel: Destination Address is fe00030a
Dec 6 18:19:38 erwin kernel: IN new connecetionPacket AcceptedSource Address is 1000d0b
Dec 6 18:19:38 erwin kernel: Destination Address is fe00030a
Dec 6 18:19:39 erwin kernel: IN new connecetionPacket AcceptedSource Address is 1000d0b
Dec 6 18:19:39 erwin kernel: Destination Address is fe00030a
Dec 6 18:19:40 erwin kernel: IN new connecetionPacket AcceptedSource Address is 1000d0b
Dec 6 18:19:40 erwin kernel: Destination Address is fe00030a
Dec 6 18:19:41 erwin kernel: IN new connecetionPacket AcceptedSource Address is 200040a
Dec 6 18:19:41 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:41 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:41 erwin kernel: Destination Address is 200040a
Dec 6 18:19:41 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:41 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:41 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:41 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:41 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:41 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:41 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:41 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:41 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:41 erwin kernel: Destination Address is 200040a
Dec 6 18:19:41 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:41 erwin kernel: Destination Address is 200040a
Dec 6 18:19:41 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:41 erwin kernel: Destination Address is 200040a
Dec 6 18:19:41 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:41 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:41 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:41 erwin kernel: Destination Address is 200040a
Dec 6 18:19:41 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:41 erwin kernel: Destination Address is 200040a
Dec 6 18:19:41 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:41 erwin kernel: Destination Address is 200040a
Dec 6 18:19:41 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:41 erwin kernel: Destination Address is 200040a
Dec 6 18:19:41 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:41 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:41 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:41 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:41 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:41 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:41 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:41 erwin kernel: Destination Address is 200040a
Dec 6 18:19:41 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:41 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:41 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:41 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:41 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:41 erwin kernel: Destination Address is 200040a
Dec 6 18:19:41 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:41 erwin kernel: Destination Address is fd00030a

Dec 6 18:19:41 erwin kernel: IN new connecetionPacket AcceptedSource Address is 1000d0b
Dec 6 18:19:41 erwin kernel: Destination Address is fe00030a
Dec 6 18:19:42 erwin kernel: IN new connecetionPacket AcceptedSource Address is 1000d0b
Dec 6 18:19:42 erwin kernel: Destination Address is fe00030a
Dec 6 18:19:46 erwin kernel: IN new connecetionPacket AcceptedSource Address is 200040a
Dec 6 18:19:46 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:46 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:46 erwin kernel: Destination Address is 200040a
Dec 6 18:19:46 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:46 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:46 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:46 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:46 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:46 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:46 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:46 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:46 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:46 erwin kernel: Destination Address is 200040a
Dec 6 18:19:46 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:46 erwin kernel: Destination Address is 200040a
Dec 6 18:19:46 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:46 erwin kernel: Destination Address is 200040a
Dec 6 18:19:46 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:46 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:46 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:46 erwin kernel: Destination Address is 200040a
Dec 6 18:19:46 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:46 erwin kernel: Destination Address is 200040a
Dec 6 18:19:46 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:46 erwin kernel: Destination Address is 200040a
Dec 6 18:19:46 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:46 erwin kernel: Destination Address is 200040a
Dec 6 18:19:46 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:46 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:46 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:46 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:46 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:46 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:46 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:46 erwin kernel: Destination Address is 200040a
Dec 6 18:19:46 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:46 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:46 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:46 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:46 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:46 erwin kernel: Destination Address is 200040a
Dec 6 18:19:46 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 18:19:46 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:49 erwin kernel: IN new connecetionPacket AcceptedSource Address is 1000d0b
Dec 6 18:19:49 erwin kernel: Destination Address is fe00030a
Dec 6 18:19:50 erwin kernel: IN new connecetionPacket AcceptedSource Address is 1000d0b
Dec 6 18:19:50 erwin kernel: Destination Address is fe00030a
Dec 6 18:19:51 erwin kernel: IN new connecetionPacket AcceptedSource Address is 200040a
Dec 6 18:19:51 erwin kernel: Destination Address is fd00030a
Dec 6 18:19:51 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 18:19:51 erwin kernel: Destination Address is 200040a

```
Dec  6 18:19:51 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:51 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:51 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:51 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:51 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:51 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:51 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:51 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:51 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:51 erwin kernel: Destination Address is  200040a
Dec  6 18:19:51 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:51 erwin kernel: Destination Address is  200040a
Dec  6 18:19:51 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:51 erwin kernel: Destination Address is  200040a
Dec  6 18:19:51 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:51 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:51 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:51 erwin kernel: Destination Address is  200040a
Dec  6 18:19:51 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:51 erwin kernel: Destination Address is  200040a
Dec  6 18:19:51 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:51 erwin kernel: Destination Address is  200040a
Dec  6 18:19:51 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:51 erwin kernel: Destination Address is  200040a
Dec  6 18:19:51 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:51 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:51 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:51 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:51 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:51 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:51 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:51 erwin kernel: Destination Address is  200040a
Dec  6 18:19:51 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:51 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:51 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:51 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:51 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 18:19:51 erwin kernel: Destination Address is  200040a
Dec  6 18:19:51 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 18:19:51 erwin kernel: Destination Address is  fd00030a
Dec  6 18:19:51 erwin kernel: IN new connecetionIN packet dropper
Dec  6 18:19:51 erwin kernel: Packet DroppedSource Address is  1000d0b
Dec  6 18:19:51 erwin kernel: Destination Address is  fe00030a
Dec  6 18:19:52 erwin kernel: IN new connecetionIN packet dropper
Dec  6 18:19:52 erwin kernel: Packet DroppedSource Address is  1000d0b
Dec  6 18:19:52 erwin kernel: Destination Address is  fe00030a
Dec  6 18:19:53 erwin kernel: IN new connecetionIN packet dropper
Dec  6 18:19:53 erwin kernel: Packet DroppedSource Address is  1000d0b
Dec  6 18:19:53 erwin kernel: Destination Address is  fe00030a
Dec  6 18:19:54 erwin kernel: IN new connecetionIN packet dropper
Dec  6 18:19:54 erwin kernel: Packet DroppedSource Address is  1000d0b
Dec  6 18:19:54 erwin kernel: Destination Address is  fe00030a
Dec  6 18:19:55 erwin kernel: IN new connecetionIN packet dropper
Dec  6 18:19:55 erwin kernel: Packet DroppedSource Address is  1000d0b
Dec  6 18:19:55 erwin kernel: Destination Address is  fe00030a
```

Thus as can be seen the spoofed address 11.13.0.1 from Hubble gets dropped and is not able to get through. This can be seen further by the ethereal output of Hubble and Chekov as Chekov being the unsafe router forwards the spoofed packet to Erwin

Ethereal Output of Chekov

No.	Time	Source	Destination	Protocol	Info
1	0.000000	10.4.0.2	10.3.0.253	TCP	49397 > ipp [SYN]
2	0.000479	10.3.0.253	10.4.0.2	TCP	ipp > 49397 [SYN, ACK]
3	0.000498	10.4.0.2	10.3.0.253	TCP	49397 > ipp [ACK]
4	0.000535	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
5	0.000541	10.4.0.2	10.3.0.253	HTTP	Continuation
6	0.000547	10.4.0.2	10.3.0.253	HTTP	Continuation
7	0.000974	10.3.0.253	10.4.0.2	TCP	ipp > 49397 [ACK]
8	0.000988	10.3.0.253	10.4.0.2	TCP	ipp > 49397 [ACK]
9	0.001000	10.4.0.2	10.3.0.253	IPP	IPP request
10	0.001022	10.3.0.253	10.4.0.2	TCP	ipp > 49397 [ACK]
11	0.002742	10.3.0.253	10.4.0.2	TCP	ipp > 49397 [ACK]
12	0.003026	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
13	0.003035	10.4.0.2	10.3.0.253	TCP	49397 > ipp [ACK]
14	0.003101	10.3.0.253	10.4.0.2	HTTP	Continuation
15	0.003102	10.3.0.253	10.4.0.2	HTTP	Continuation
16	0.003112	10.4.0.2	10.3.0.253	TCP	49397 > ipp [ACK]
17	0.003115	10.4.0.2	10.3.0.253	TCP	49397 > ipp [ACK]
18	0.004892	10.3.0.253	10.4.0.2	IPP	IPP response
19	0.004901	10.4.0.2	10.3.0.253	TCP	49397 > ipp [ACK]
20	0.004926	10.4.0.2	10.3.0.253	TCP	49397 > ipp [FIN, ACK]
21	0.005334	10.3.0.253	10.4.0.2	TCP	ipp > 49397 [FIN, ACK]
22	0.005343	10.4.0.2	10.3.0.253	TCP	49397 > ipp [ACK]
23	5.009997	10.4.0.2	10.3.0.253	TCP	49398 > ipp [SYN]
24	5.010470	10.3.0.253	10.4.0.2	TCP	ipp > 49398 [SYN, ACK]
25	5.010488	10.4.0.2	10.3.0.253	TCP	49398 > ipp [ACK]
26	5.010544	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
27	5.010550	10.4.0.2	10.3.0.253	HTTP	Continuation
28	5.010555	10.4.0.2	10.3.0.253	HTTP	Continuation
29	5.010956	10.3.0.253	10.4.0.2	TCP	ipp > 49398 [ACK]
30	5.010978	10.4.0.2	10.3.0.253	IPP	IPP request
31	5.011017	10.3.0.253	10.4.0.2	TCP	ipp > 49398 [ACK]
32	5.011018	10.3.0.253	10.4.0.2	TCP	ipp > 49398 [ACK]
33	5.012707	10.3.0.253	10.4.0.2	TCP	ipp > 49398 [ACK]
34	5.012952	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
35	5.012973	10.4.0.2	10.3.0.253	TCP	49398 > ipp [ACK]
36	5.013025	10.3.0.253	10.4.0.2	HTTP	Continuation
37	5.013026	10.3.0.253	10.4.0.2	HTTP	Continuation
38	5.013056	10.4.0.2	10.3.0.253	TCP	49398 > ipp [ACK]
39	5.013060	10.4.0.2	10.3.0.253	TCP	49398 > ipp [ACK]
40	5.014818	10.3.0.253	10.4.0.2	IPP	IPP response
41	5.014838	10.4.0.2	10.3.0.253	TCP	49398 > ipp [ACK]
42	5.014863	10.4.0.2	10.3.0.253	TCP	49398 > ipp [FIN, ACK]
43	5.015259	10.3.0.253	10.4.0.2	TCP	ipp > 49398 [FIN, ACK]
44	5.015269	10.4.0.2	10.3.0.253	TCP	49398 > ipp [ACK]
45	10.019998	10.4.0.2	10.3.0.253	TCP	49399 > ipp [SYN]
46	10.020458	10.3.0.253	10.4.0.2	TCP	ipp > 49399 [SYN, ACK]
47	10.020475	10.4.0.2	10.3.0.253	TCP	49399 > ipp [ACK]
48	10.020533	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1

49	10.020539	10.4.0.2	10.3.0.253	HTTP	Continuation
50	10.020545	10.4.0.2	10.3.0.253	HTTP	Continuation
51	10.020982	10.3.0.253	10.4.0.2	TCP	ipp > 49399 [ACK]
52	10.020996	10.3.0.253	10.4.0.2	TCP	ipp > 49399 [ACK]
53	10.021013	10.3.0.253	10.4.0.2	TCP	ipp > 49399 [ACK]
54	10.021054	10.4.0.2	10.3.0.253	IPP	IPP request
55	10.022196	10.3.0.253	10.4.0.2	TCP	ipp > 49399 [ACK]
56	10.022451	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
57	10.022472	10.4.0.2	10.3.0.253	TCP	49399 > ipp [ACK]
58	10.022512	10.3.0.253	10.4.0.2	HTTP	Continuation
59	10.022526	10.3.0.253	10.4.0.2	HTTP	Continuation
60	10.022556	10.4.0.2	10.3.0.253	TCP	49399 > ipp [ACK]
61	10.022560	10.4.0.2	10.3.0.253	TCP	49399 > ipp [ACK]
62	10.024316	10.3.0.253	10.4.0.2	IPP	IPP response
63	10.024337	10.4.0.2	10.3.0.253	TCP	49399 > ipp [ACK]
64	10.024362	10.4.0.2	10.3.0.253	TCP	49399 > ipp [FIN, ACK]
65	10.024771	10.3.0.253	10.4.0.2	TCP	ipp > 49399 [FIN, ACK]
66	10.024780	10.4.0.2	10.3.0.253	TCP	49399 > ipp [ACK]
67	11.846409	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
68	12.858153	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
69	13.859534	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
70	14.859479	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
71	15.029998	10.4.0.2	10.3.0.253	TCP	49400 > ipp [SYN]
72	15.030469	10.3.0.253	10.4.0.2	TCP	ipp > 49400 [SYN, ACK]
73	15.030487	10.4.0.2	10.3.0.253	TCP	49400 > ipp [ACK]
74	15.030544	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
75	15.030551	10.4.0.2	10.3.0.253	HTTP	Continuation
76	15.030556	10.4.0.2	10.3.0.253	HTTP	Continuation
77	15.030959	10.3.0.253	10.4.0.2	TCP	ipp > 49400 [ACK]
78	15.030981	10.4.0.2	10.3.0.253	IPP	IPP request
79	15.031029	10.3.0.253	10.4.0.2	TCP	ipp > 49400 [ACK]
80	15.031054	10.3.0.253	10.4.0.2	TCP	ipp > 49400 [ACK]
81	15.032711	10.3.0.253	10.4.0.2	TCP	ipp > 49400 [ACK]
82	15.032956	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
83	15.032977	10.4.0.2	10.3.0.253	TCP	49400 > ipp [ACK]
84	15.033039	10.3.0.253	10.4.0.2	HTTP	Continuation
85	15.033040	10.3.0.253	10.4.0.2	HTTP	Continuation
86	15.033071	10.4.0.2	10.3.0.253	TCP	49400 > ipp [ACK]
87	15.033074	10.4.0.2	10.3.0.253	TCP	49400 > ipp [ACK]
88	15.034829	10.3.0.253	10.4.0.2	IPP	IPP response
89	15.034849	10.4.0.2	10.3.0.253	TCP	49400 > ipp [ACK]
90	15.034874	10.4.0.2	10.3.0.253	TCP	49400 > ipp [FIN, ACK]
91	15.035289	10.3.0.253	10.4.0.2	TCP	ipp > 49400 [FIN, ACK]
92	15.035298	10.4.0.2	10.3.0.253	TCP	49400 > ipp [ACK]
93	15.859397	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
94	16.877672	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
95	17.877642	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
96	18.877453	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
97	19.877350	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
98	20.040050	10.4.0.2	10.3.0.253	TCP	49401 > ipp [SYN]
99	20.040496	10.3.0.253	10.4.0.2	TCP	ipp > 49401 [SYN, ACK]
100	20.040513	10.4.0.2	10.3.0.253	TCP	49401 > ipp [ACK]
101	20.040591	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
102	20.040598	10.4.0.2	10.3.0.253	HTTP	Continuation
103	20.040603	10.4.0.2	10.3.0.253	HTTP	Continuation
104	20.041994	10.3.0.253	10.4.0.2	TCP	ipp > 49401 [ACK]

105	20.042017	10.3.0.253	10.4.0.2	TCP	ipp > 49401 [ACK]
106	20.042043	10.3.0.253	10.4.0.2	TCP	ipp > 49401 [ACK]
107	20.042067	10.4.0.2	10.3.0.253	IPP	IPP request
108	20.043193	10.3.0.253	10.4.0.2	TCP	ipp > 49401 [ACK]
109	20.044432	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
110	20.044453	10.4.0.2	10.3.0.253	TCP	49401 > ipp [ACK]
111	20.044514	10.3.0.253	10.4.0.2	HTTP	Continuation
112	20.044515	10.3.0.253	10.4.0.2	HTTP	Continuation
113	20.044546	10.4.0.2	10.3.0.253	TCP	49401 > ipp [ACK]
114	20.044550	10.4.0.2	10.3.0.253	TCP	49401 > ipp [ACK]
115	20.046314	10.3.0.253	10.4.0.2	IPP	IPP response
116	20.046334	10.4.0.2	10.3.0.253	TCP	49401 > ipp [ACK]
117	20.046371	10.4.0.2	10.3.0.253	TCP	49401 > ipp [FIN, ACK]
118	20.046790	10.3.0.253	10.4.0.2	TCP	ipp > 49401 [FIN, ACK]
119	20.046799	10.4.0.2	10.3.0.253	TCP	49401 > ipp [ACK]
120	20.877303	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
121	21.877192	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
122	22.877005	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
123	23.876901	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
124	24.876789	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
125	25.050006	10.4.0.2	10.3.0.253	TCP	49402 > ipp [SYN]
126	25.050460	10.3.0.253	10.4.0.2	TCP	ipp > 49402 [SYN, ACK]
127	25.050477	10.4.0.2	10.3.0.253	TCP	49402 > ipp [ACK]
128	25.050533	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
129	25.050539	10.4.0.2	10.3.0.253	HTTP	Continuation
130	25.050545	10.4.0.2	10.3.0.253	HTTP	Continuation
131	25.050990	10.3.0.253	10.4.0.2	TCP	ipp > 49402 [ACK]
132	25.051010	10.3.0.253	10.4.0.2	TCP	ipp > 49402 [ACK]
133	25.051024	10.3.0.253	10.4.0.2	TCP	ipp > 49402 [ACK]
134	25.051062	10.4.0.2	10.3.0.253	IPP	IPP request
135	25.052197	10.3.0.253	10.4.0.2	TCP	ipp > 49402 [ACK]
136	25.052443	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
137	25.052464	10.4.0.2	10.3.0.253	TCP	49402 > ipp [ACK]
138	25.052510	10.3.0.253	10.4.0.2	HTTP	Continuation
139	25.052511	10.3.0.253	10.4.0.2	HTTP	Continuation
140	25.052542	10.4.0.2	10.3.0.253	TCP	49402 > ipp [ACK]
141	25.052546	10.4.0.2	10.3.0.253	TCP	49402 > ipp [ACK]
142	25.054302	10.3.0.253	10.4.0.2	IPP	IPP response
143	25.054322	10.4.0.2	10.3.0.253	TCP	49402 > ipp [ACK]
144	25.054347	10.4.0.2	10.3.0.253	TCP	49402 > ipp [FIN, ACK]
145	25.054732	10.3.0.253	10.4.0.2	TCP	ipp > 49402 [FIN, ACK]
146	25.054742	10.4.0.2	10.3.0.253	TCP	49402 > ipp [ACK]
147	25.876683	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
148	26.876551	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
149	27.876440	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
150	28.876331	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
151	29.876222	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
152	30.043114	Intel_d3:cb:73	Intel_d3:d2:a0	ARP	Who has 10.4.0.2? Tel
153	30.043136	Intel_d3:d2:a0	Intel_d3:cb:73	ARP	10.4.0.2 is at 00:02:b
154	30.060002	10.4.0.2	10.3.0.253	TCP	49403 > ipp [SYN]
155	30.060446	10.3.0.253	10.4.0.2	TCP	ipp > 49403 [SYN, ACK]
156	30.060464	10.4.0.2	10.3.0.253	TCP	49403 > ipp [ACK]
157	30.060521	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
158	30.060528	10.4.0.2	10.3.0.253	HTTP	Continuation
159	30.060931	10.3.0.253	10.4.0.2	TCP	ipp > 49403 [ACK]
160	30.060944	10.3.0.253	10.4.0.2	TCP	ipp > 49403 [ACK]

161	30.060978	10.4.0.2	10.3.0.253	IPP	IPP request
162	30.062130	10.3.0.253	10.4.0.2	TCP	ipp > 49403 [ACK]
163	30.062377	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
164	30.062398	10.4.0.2	10.3.0.253	TCP	49403 > ipp [ACK]
165	30.062456	10.3.0.253	10.4.0.2	HTTP	Continuation
166	30.062457	10.3.0.253	10.4.0.2	HTTP	Continuation
167	30.062488	10.4.0.2	10.3.0.253	TCP	49403 > ipp [ACK]
168	30.062491	10.4.0.2	10.3.0.253	TCP	49403 > ipp [ACK]
169	30.064244	10.3.0.253	10.4.0.2	IPP	IPP response
170	30.064264	10.4.0.2	10.3.0.253	TCP	49403 > ipp [ACK]
171	30.064289	10.4.0.2	10.3.0.253	TCP	49403 > ipp [FIN, ACK]
172	30.064666	10.3.0.253	10.4.0.2	TCP	ipp > 49403 [FIN, ACK]
173	30.064675	10.4.0.2	10.3.0.253	TCP	49403 > ipp [ACK]
174	30.876104	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
175	31.875991	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
176	32.875886	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
177	33.875837	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
178	34.243652	Hewlett-_43:2e:a6	CDP/VTP	CDP	Cisco Discovery Protoc
179	34.875731	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
180	35.070006	10.4.0.2	10.3.0.253	TCP	49404 > ipp [SYN]
181	35.070452	10.3.0.253	10.4.0.2	TCP	ipp > 49404 [SYN, ACK]
182	35.070468	10.4.0.2	10.3.0.253	TCP	49404 > ipp [ACK]
183	35.070526	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
184	35.070532	10.4.0.2	10.3.0.253	HTTP	Continuation
185	35.070537	10.4.0.2	10.3.0.253	HTTP	Continuation
186	35.070966	10.3.0.253	10.4.0.2	TCP	ipp > 49404 [ACK]
187	35.070969	10.3.0.253	10.4.0.2	TCP	ipp > 49404 [ACK]
188	35.071021	10.3.0.253	10.4.0.2	TCP	ipp > 49404 [ACK]
189	35.071030	10.4.0.2	10.3.0.253	IPP	IPP request
190	35.072755	10.3.0.253	10.4.0.2	TCP	ipp > 49404 [ACK]
191	35.073034	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
192	35.073055	10.4.0.2	10.3.0.253	TCP	49404 > ipp [ACK]
193	35.073095	10.3.0.253	10.4.0.2	HTTP	Continuation
194	35.073096	10.3.0.253	10.4.0.2	HTTP	Continuation
195	35.073133	10.4.0.2	10.3.0.253	TCP	49404 > ipp [ACK]
196	35.073137	10.4.0.2	10.3.0.253	TCP	49404 > ipp [ACK]
197	35.074894	10.3.0.253	10.4.0.2	IPP	IPP response
198	35.074914	10.4.0.2	10.3.0.253	TCP	49404 > ipp [ACK]
199	35.074939	10.4.0.2	10.3.0.253	TCP	49404 > ipp [FIN, ACK]
200	35.075330	10.3.0.253	10.4.0.2	TCP	ipp > 49404 [FIN, ACK]
201	35.075340	10.4.0.2	10.3.0.253	TCP	49404 > ipp [ACK]
202	35.875541	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
203	36.875504	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
204	37.875388	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
205	38.875277	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
206	39.875094	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
207	40.079993	10.4.0.2	10.3.0.253	TCP	49405 > ipp [SYN]
208	40.080446	10.3.0.253	10.4.0.2	TCP	ipp > 49405 [SYN, ACK]
209	40.080462	10.4.0.2	10.3.0.253	TCP	49405 > ipp [ACK]
210	40.080518	10.4.0.2	10.3.0.253	HTTP	POST / HTTP/1.1
211	40.080524	10.4.0.2	10.3.0.253	HTTP	Continuation
212	40.080530	10.4.0.2	10.3.0.253	HTTP	Continuation
213	40.080986	10.3.0.253	10.4.0.2	TCP	ipp > 49405 [ACK]
214	40.081001	10.3.0.253	10.4.0.2	TCP	ipp > 49405 [ACK]
215	40.081003	10.3.0.253	10.4.0.2	TCP	ipp > 49405 [ACK]
216	40.081046	10.4.0.2	10.3.0.253	IPP	IPP request

217	40.082180	10.3.0.253	10.4.0.2	TCP	ipp > 49405 [ACK]
218	40.083413	10.3.0.253	10.4.0.2	HTTP	HTTP/1.1 200 OK
219	40.083434	10.4.0.2	10.3.0.253	TCP	49405 > ipp [ACK]
220	40.083484	10.3.0.253	10.4.0.2	HTTP	Continuation
221	40.083498	10.3.0.253	10.4.0.2	HTTP	Continuation
222	40.083529	10.4.0.2	10.3.0.253	TCP	49405 > ipp [ACK]
223	40.083532	10.4.0.2	10.3.0.253	TCP	49405 > ipp [ACK]
224	40.085276	10.3.0.253	10.4.0.2	IPP	IPP response
225	40.085296	10.4.0.2	10.3.0.253	TCP	49405 > ipp [ACK]
226	40.085321	10.4.0.2	10.3.0.253	TCP	49405 > ipp [FIN, ACK]
227	40.085727	10.3.0.253	10.4.0.2	TCP	ipp > 49405 [FIN, ACK]
228	40.085736	10.4.0.2	10.3.0.253	TCP	49405 > ipp [ACK]

Ethereal Output Hubble

No.	Time	Source	Destination	Protocol	Info
1	0.000000	Hewlett-_43:2e:a2	CDP/VTP	CDP	Cisco Discovery Protocol
2	37.605449	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
3	38.617318	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
4	39.618810	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
5	40.618870	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
6	41.618898	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
7	42.597264	Intel_d3:d2:79	Intel_d3:d2:9f	ARP	Who has 10.1.0.2? Tell 10.1.0.1
8	42.597392	Intel_d3:d2:9f	Intel_d3:d2:79	ARP	10.1.0.2 is at 00:02:b3:d3:d2:9f
9	42.637286	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
10	43.637369	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
11	44.637294	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
12	45.637302	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
13	46.637369	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
14	47.637369	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
15	48.637295	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
16	49.637303	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
17	50.637304	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
18	51.637302	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
19	52.637293	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
20	53.637293	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
21	54.637297	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
22	55.637300	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
23	56.637294	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
24	57.637292	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
25	58.637301	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
26	59.637363	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
27	60.003429	Hewlett-_43:2e:a2	CDP/VTP	CDP	Cisco Discovery Protocol
28	60.637370	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
29	61.637293	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
30	62.637369	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
31	63.637364	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
32	64.637365	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request
33	65.637294	11.13.0.1	10.3.0.254	ICMP	Echo (ping) request

Thus as can be seen no packet was able to pass through Erwin

Test Case Six

We will now verify test case six by showing that a spoofing hosts packets are dropped even if the address does exist on the network which in this case is 10.13.0.1 (Franklin)

We will first see Erwin's Kernel messages log file

/var/log/messages

```
Dec  6 16:05:39 erwin syslogd 1.4.1: restart.
Dec  6 16:05:39 erwin syslog: syslogd startup succeeded
Dec  6 16:05:39 erwin kernel: klogd 1.4.1, log source = /proc/kmsg started.
Dec  6 16:05:39 erwin syslog: klogd startup succeeded
Dec  6 16:05:39 erwin syslog: syslogd shutdown succeeded
Dec  6 16:06:06 erwin kernel: IN new connecetionIN new ICMP send
Dec  6 16:06:06 erwin kernel:
Dec  6 16:06:06 erwin kernel:
Dec  6 16:06:06 erwin kernel: Packet AcceptedSource Address is  fa00030a
Dec  6 16:06:06 erwin kernel: Destination Address is  1000040a
Dec  6 16:06:06 erwin kernel: Packet Displayeddevice IP Address is  fd00060a
Dec  6 16:06:06 erwin kernel: Packet Displayeddevice IP Address is  100040a
Dec  6 16:06:26 erwin kernel: Destination Address is  fd00030a
Dec  6 16:06:26 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 16:06:26 erwin kernel: Destination Address is  200040a
Dec  6 16:06:26 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 16:06:26 erwin kernel: Destination Address is  fd00030a
Dec  6 16:06:26 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 16:06:26 erwin kernel: Destination Address is  fd00030a
Dec  6 16:06:26 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 16:06:26 erwin kernel: Destination Address is  200040a
Dec  6 16:06:26 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 16:06:26 erwin kernel: Destination Address is  fd00030a
Dec  6 16:06:27 erwin kernel: IN new connecetionIN new ICMP send
Dec  6 16:06:27 erwin kernel:
Dec  6 16:06:27 erwin kernel:
Dec  6 16:06:27 erwin kernel: Packet AcceptedSource Address is  1000d0a
Dec  6 16:06:27 erwin kernel: Destination Address is  fe00030a
Dec  6 16:06:27 erwin kernel: Packet Displayeddevice IP Address is  fd00060a
Dec  6 16:06:27 erwin kernel: Packet Displayeddevice IP Address is  100040a
Dec  6 16:06:28 erwin kernel: IN new connecetionPacket AcceptedSource Address is  1000d0a
Dec  6 16:06:28 erwin kernel: Destination Address is  fe00030a
Dec  6 16:06:29 erwin kernel: IN new connecetionPacket AcceptedSource Address is  1000d0a
Dec  6 16:06:29 erwin kernel: Destination Address is  fe00030a
Dec  6 16:06:30 erwin kernel: IN new connecetionPacket AcceptedSource Address is  1000d0a
Dec  6 16:06:30 erwin kernel: Destination Address is  fe00030a
Dec  6 16:06:31 erwin kernel: IN new connecetionPacket AcceptedSource Address is  200040a
Dec  6 16:06:31 erwin kernel: Destination Address is  fd00030a
Dec  6 16:06:31 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 16:06:31 erwin kernel: Destination Address is  200040a
Dec  6 16:06:31 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 16:06:31 erwin kernel: Destination Address is  fd00030a
Dec  6 16:06:31 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 16:06:31 erwin kernel: Destination Address is  fd00030a
Dec  6 16:06:31 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
```

Dec 6 16:06:31 erwin kernel: Destination Address is fd00030a
Dec 6 16:06:31 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 16:06:31 erwin kernel: Destination Address is fd00030a
Dec 6 16:06:31 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 16:06:31 erwin kernel: Destination Address is 200040a
Dec 6 16:06:31 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 16:06:31 erwin kernel: Destination Address is 200040a
Dec 6 16:06:31 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 16:06:31 erwin kernel: Destination Address is 200040a
Dec 6 16:06:31 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 16:06:31 erwin kernel: Destination Address is fd00030a
Dec 6 16:06:31 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 16:06:31 erwin kernel: Destination Address is 200040a
Dec 6 16:06:31 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 16:06:31 erwin kernel: Destination Address is 200040a
Dec 6 16:06:31 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 16:06:31 erwin kernel: Destination Address is 200040a
Dec 6 16:06:31 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 16:06:31 erwin kernel: Destination Address is 200040a
Dec 6 16:06:31 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 16:06:31 erwin kernel: Destination Address is fd00030a
Dec 6 16:06:31 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 16:06:31 erwin kernel: Destination Address is fd00030a
Dec 6 16:06:31 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 16:06:31 erwin kernel: Destination Address is fd00030a
Dec 6 16:06:31 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 16:06:31 erwin kernel: Destination Address is 200040a
Dec 6 16:06:31 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 16:06:31 erwin kernel: Destination Address is fd00030a
Dec 6 16:06:31 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 16:06:31 erwin kernel: Destination Address is fd00030a
Dec 6 16:06:31 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 16:06:31 erwin kernel: Destination Address is 200040a
Dec 6 16:06:38 erwin kernel: IN new connecetionIN packet dropper
Dec 6 16:06:38 erwin kernel: Packet DroppedSource Address is 1000d0a
Dec 6 16:06:38 erwin kernel: Destination Address is fe00030a
Dec 6 16:06:39 erwin kernel: IN new connecetionIN packet dropper
Dec 6 16:06:39 erwin kernel: Packet DroppedSource Address is 1000d0a
Dec 6 16:06:39 erwin kernel: Destination Address is fe00030a
Dec 6 16:06:40 erwin kernel: IN new connecetionIN packet dropper
Dec 6 16:06:40 erwin kernel: Packet DroppedSource Address is 1000d0a
Dec 6 16:06:40 erwin kernel: Destination Address is fe00030a
Dec 6 16:06:41 erwin kernel: IN new connecetionPacket AcceptedSource Address is 200040a
Dec 6 16:06:41 erwin kernel: Destination Address is fd00030a
Dec 6 16:06:41 erwin kernel: Packet Accepted due to conditionSource Address is fd00030a
Dec 6 16:06:41 erwin kernel: Destination Address is 200040a
Dec 6 16:06:41 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 16:06:41 erwin kernel: Destination Address is fd00030a
Dec 6 16:06:41 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 16:06:41 erwin kernel: Destination Address is fd00030a
Dec 6 16:06:41 erwin kernel: IN new connecetionIN packet dropper
Dec 6 16:06:41 erwin kernel: Packet DroppedSource Address is 1000d0a
Dec 6 16:06:41 erwin kernel: Destination Address is fe00030a
Dec 6 16:07:41 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a
Dec 6 16:07:41 erwin kernel: Destination Address is fd00030a
Dec 6 16:07:41 erwin kernel: IN old connecetionPacket Accepted Source Address is 200040a

```
Dec  6 16:07:41 erwin kernel: Destination Address is  fd00030a
Dec  6 16:07:41 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 16:07:41 erwin kernel: Destination Address is  200040a
Dec  6 16:07:41 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 16:07:41 erwin kernel: Destination Address is  fd00030a
Dec  6 16:07:41 erwin kernel: IN new connecetionIN packet dropper
Dec  6 16:07:41 erwin kernel: Packet DroppedSource Address is  1000d0a
Dec  6 16:07:41 erwin kernel: Destination Address is  fe00030a
Dec  6 16:07:42 erwin kernel: IN new connecetionIN packet dropper
Dec  6 16:07:42 erwin kernel: Packet DroppedSource Address is  1000d0a
Dec  6 16:07:42 erwin kernel: Destination Address is  fe00030a
Dec  6 16:07:43 erwin kernel: IN new connecetionIN packet dropper
Dec  6 16:07:43 erwin kernel: Packet DroppedSource Address is  1000d0a
Dec  6 16:07:43 erwin kernel: Destination Address is  fe00030a
Dec  6 16:07:44 erwin kernel: IN new connecetionIN packet dropper
Dec  6 16:07:44 erwin kernel: Packet DroppedSource Address is  1000d0a
Dec  6 16:07:44 erwin kernel: Destination Address is  fe00030a
Dec  6 16:07:45 erwin kernel: IN new connecetionIN packet dropper
Dec  6 16:07:45 erwin kernel: Packet DroppedSource Address is  1000d0a
Dec  6 16:07:45 erwin kernel: Destination Address is  fe00030a
Dec  6 16:07:46 erwin kernel: IN new connecetionPacket AcceptedSource Address is  200040a
Dec  6 16:07:46 erwin kernel: Destination Address is  fd00030a
Dec  6 16:07:46 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 16:07:46 erwin kernel: Destination Address is  200040a
Dec  6 16:07:46 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 16:07:46 erwin kernel: Destination Address is  fd00030a
Dec  6 16:07:46 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 16:07:46 erwin kernel: Destination Address is  fd00030a
Dec  6 16:07:46 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 16:07:46 erwin kernel: Destination Address is  200040a
Dec  6 16:07:46 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 16:07:46 erwin kernel: Destination Address is  fd00030a
Dec  6 16:07:46 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 16:07:46 erwin kernel: Destination Address is  fd00030a
Dec  6 16:07:46 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 16:07:46 erwin kernel: Destination Address is  200040a
Dec  6 16:07:46 erwin kernel: IN old connecetionPacket Accepted Source Address is  200040a
Dec  6 16:07:46 erwin kernel: Destination Address is  fd00030a
Dec  6 16:07:46 erwin kernel: IN new connecetionIN packet dropper
Dec  6 16:07:46 erwin kernel: Packet DroppedSource Address is  1000d0a
Dec  6 16:07:46 erwin kernel: Destination Address is  fe00030a
Dec  6 16:07:51 erwin kernel: IN new connecetionPacket AcceptedSource Address is  200040a
Dec  6 16:07:51 erwin kernel: Destination Address is  fd00030a
Dec  6 16:07:51 erwin kernel: Packet Accepted  due to conditionSource Address is  fd00030a
Dec  6 16:07:51 erwin kernel: Destination Address is  200040a
```

Thus as it is seen Erwin does not allow known IP addresses but from the wrong interface

This can also be seen from the output of the ethereal file of Hubble

Ethereal File of Hubble

No.	Time	Source	Destination	Protocol	Info
1	0.000000	Hewlett-_43:2e:a2	CDP/VTP	CDP	Cisco Discovery Protocol
2	44.045435	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
3	45.045307	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
4	46.045307	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
5	47.045400	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
6	48.045459	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
7	49.042799	Intel_d3:d2:79	Intel_d3:d2:9f	ARP	Who has 10.1.0.2? Tell 10.1.0.1
8	49.042927	Intel_d3:d2:9f	Intel_d3:d2:79	ARP	10.1.0.2 is at 00:02:b3:d3:d2:9f
9	49.062830	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
10	50.062829	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
11	51.062828	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
12	52.062829	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
13	53.062830	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
14	54.062830	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
15	55.062829	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
16	56.062830	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
17	57.062830	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
18	58.062831	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
19	60.003309	Hewlett-_43:2e:a2	CDP/VTP	CDP	Cisco Discovery Protocol
20	117.795225	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
21	118.812820	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
22	119.813294	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
23	120.006084	Hewlett-_43:2e:a2	CDP/VTP	CDP	Cisco Discovery Protocol
24	120.812830	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
25	121.813298	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
26	122.792798	Intel_d3:d2:79	Intel_d3:d2:9f	ARP	Who has 10.1.0.2? Tell 10.1.0.1
27	122.792939	Intel_d3:d2:9f	Intel_d3:d2:79	ARP	10.1.0.2 is at 00:02:b3:d3:d2:9f
28	122.812824	10.13.0.1	10.3.0.254	ICMP	Echo (ping) request
29	131.932207	10.13.0.1	10.3.0.254	TCP	33277 > ftp [SYN]
30	134.922811	10.13.0.1	10.3.0.254	TCP	33277 > ftp [SYN]

9 Conclusions

9.1 Summary

The module was designed to rectify the fault that source addresses are not checked to be valid or not and as shown by the test results was able to successfully rectify the fault. This was done by using Loadable Linux Kernel Modules, Net Filter hooks and Connection Tracking. There were two lists maintained by the system The first list stored all the different source addresses of the packets that were seen by this router, This list also stored information of the source address i.e. the device where the packet came from and whether they are valid or not thus for every new connection that was seen this list was traversed for the proper source address and validated. If it was not valid then the packets were blocked and no further packets were allowed to go through. If the source address was seen for the first time then the address was first checked with the neighbor table i.e. the table which stores the machines which are directly connected to this router to verify whether the packet was from a neighbor or not if it was not from a neighbor then the address was stored in the master list with the device and a temporary value that it was not valid but was in the process of being verified. For verification an ICMP_ECHOREQUEST is sent to the original source using the second list which stores the various IP addresses of the router and the devices that are associated with these IP addresses. This list is maintained by a function registered in the NF_IP_LOCAL_IN hook of the net filter facility which extracts the destination address from the incoming packets and the device from which the packet has come from. This is all done in a function that is registered in the NF_IP_FORWARD hook of the net filter facility. If the source address of the packet is valid then the original source of the new connection packet would reply back by an ICMP packet that has the code set as ICMP_ECHOREPLY which would then be caught by another function that has also been registered in the NF_IP_LOCAL_IN hook, This function would then update the source address list and change the source address from being invalid to valid and discard the temporary invalid status This would be done if and only if the packet has arrived from the device that the original packet came from. Thus as proved by testing source address spoofing can be successfully eliminated

9.2 Problems Encountered and Solved

The problems encountered were

- The module did not compile properly, the solution for this was that the compiler did not have the proper parameters namely gcc -I/usr/src/linux/include -O2 -D__KERNEL__ -Wall while compiling the module and once given the module compiled perfectly
- Once compiled the module while loading gave errors that the kernel version was different and not compiled for the base kernel, The solution for this was to change the parameter ExtraVersions in the /usr/src/linux/MAKEFILE to match the current kernel and then give the make command for this file so that the modules were updated
- The biggest problem was how to send the ICMP packet, there are three ways to achieve it in the Linux kernel
    1. Manufacturing a new packet and sending it back to the original source. This way was not achieved as the sending function (dev_xmit()) would reject the packet.
    2. The second way was to use a program based in user space and send the packet from there. This can be achieved by using a function known as call_usermodehelper(), The problem with this function is however that it can be only called in the process mode while the module runs in the interrupt mode, for this the function had to be scheduled by using the system call schedule_task() This is achieved by the code given below

```
 static void newicmppacket(char * newadd)
{
  char *argv [4] ;
  char **envp;
  char *buff;
  char *buff1;
  int i=0;
   envp = (char **) kmalloc (60 * sizeof (char *), GFP_KERNEL);
  envp[i++]="HOME=/";
   envp[i++]="TERM=linux";
  envp[i++]= "PATH=/sbin:/bin:/usr/sbin:/usr/bin";
   envp[i++]="SHELL=/bin/bash";
  envp[i++]="DEBUG=kernel";
  envp [i] = 0;
  buff1 = kmalloc(256, GFP_KERNEL);
  argv[0]= "/sbin/ping_packet_new";
  sprintf(buff,"%x",newadd->dadd);
  argv[1]=buff;
  sprintf(buff1,"%x",newadd->sadd);
  argv[2]=buff1;
  argv [3]=0;
   i =  call_usermodehelper(argv[0],argv, envp);
 kfree (buff);
 kfree (envp);
  kfree (buff1);
  }
```

```
void schedule_system_call (struct addresspass *newadd )
{

        my_task.routine=newicmppacket;
        my_task.data=newadd;
        schedule_task(&my_task);

}
```

This would schedule a process call when the system comes back from the interrupt mode. This approach however does not work as the character pointer of the structure my_task gets destroyed or corrupted leading to system failure

3. The third approach is to use the icmp_send function in /usr/src/linux/net/ipv4/icmp.c for this however we have to manipulate the packet to make it of this host i.e. make it of the type PACKET_HOST from the type PACKET_OTHERHOST and also change the source and destination addresses. This approach works and then was integrated into the module

- The module also kept crashing and the callback trace revealed that the module crashed due to access to a NULL space. This was corrected by correcting the logic of accessing the link list.
- Connection Tracking did not work, for this the connection tracking module had to be loaded into the kernel first
- During Testing the Spoofing code would not work properly due to the fact that when the IP address was changed the checksum of both the IP header and the TCP header got wrong and thus the packet got dropped. To correct this checksum of both the TCP pseudo header and the IP header had to be recomputed.

9.3 Suggestions for Future Extensions to Project

The project can be extended as and when the problem of source address spoofing of the neighbor is solved to include the functionality of how to prevent spoofing from the same subnet as the victim source and the spoofing source.

Glossary

LKM - Loadable Kernel Module
ACK - Acknowledgment
API - Application Programming Interface
ARP - Address Resolution Protocol
ATM - Asynchronous Transfer Mode
ECN - Explicit Congestion Notification
FIB - Forward Information Base
ICMP - Internet Control Message Protocol
I/O - Input/Output
IP - Internet Protocol
IPv4 - IP version 4
IPv6 - IP version 6
LAN - Local Area Network
MAC - Media Access Control
MSS - Maximum Segment Size
RFC - Request For Comment
RTT - Round Trip Time
SYN - Synchronize of the TCP Protocol
TCP - Transmission Control Protocol
UDP - User Datagram Protocol

Bibliography

[1]Peter Burden
Routing in the network
http://www.scit.wlv.ac.uk/~jphb/comms/iproute.html

[2] L. Todd Heberlein, Matt Bishop
Address Spoofing
http://seclab.cs.ucdavis.edu/papers/spoof-paper.pdf

[3] Morris, R.T.
A Weakness in the 4.2BSD Unix TCP/IP Software
Computing Science Technical Report
No. 117, AT&T Bell Laboratories,
Murray Hill, New Jersey.

[4]Roger S. Pressman
Software Engineering -5th Edition

[5]Behrouz A. Forouzan 2003
TCP/IP Protocol Suite – 2nd Edition

[6] Bryan Henderson
Linux Loadable Kernel Module HOWTO
http://www.tldp.org/HOWTO/Module-HOWTO/x49.html

[7]
The Net Filter Facility
www.cs.clemson.edu/~westall/881/notes/netfilter.pdf

[8] M. Rio et al.
A Map of the Networking Code in Linux Kernel 2.4.20
Technical Report DataTAG-2004-1, 31 March 2004

[9] Glenn Herrin
Linux IP Networking: A Guide to the Implementation and Modification of the Linux Protocol Stack
http://www.kernelnewbies.org/documents/ipnetworking/linuxipnetworking.html

Appendices

Appendix A

Spoofing of Source Address Code

To test the prevention of address the module had to be tested by actually spoofing the source address by using the code given below

```
#define MODULE
//header files
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/skbuff.h>
#include <linux/ip.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <net/ip.h>
#include <asm-i386/checksum.h>
#include <net/tcp.h>

static struct nf_hook_ops nfho;
//netfilter hooks structure to register hooks





unsigned int spoofipfake(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;

    int len = sb->nh.iph->ihl;
    //get length of the IP header
    __u32 fakeip = 0x01000d0a;
    //fake address to  be put into the source address of the packet

  sb->nh.iph->saddr= fakeip;
  //insert fake address
  len = len*4;
  //get length of the packet in bytes
```

```
struct iphdr *ip = sb->nh.iph;
//get IP header
//if the packet is of TCP protocol
if(sb->nh.iph->protocol==IPPROTO_TCP)
 {
      int len1 = sb->len-sb->nh.iph->ihl*4;
      //get actual length of the TCP packet
           struct tcphdr *tcp;
           //TCP header decleration to store TCP header
           struct sock *sk = sb->sk;
           // get the socket from the sk_buff
           sk->saddr=fakeip;
      //change saddr in socket also
      tcp = (struct tcphdr *)(sb->data+sb->nh.iph->ihl*4);
      //get TCP header from IP packet data
           tcp->check=0;
           //set checksum field to 0

          tcp_v4_send_check(sk,tcp,len1,sb);

          //calculate the tcp header checksum

   }


   ip->check=0;
//set IP header checksum to 0
 ip->check=ip_fast_csum((unsigned char *)ip,ip->ihl);
 //Calculate the checksum of the IP header
   return NF_ACCEPT;
}




int init_module()
{
  //fill in hook structure and register hook
   nfho.hook = spoofipfake;
   nfho.hooknum = NF_IP_POST_ROUTING;
    nfho.pf = PF_INET;
   nfho.priority = NF_IP_PRI_FIRST;
   nf_register_hook(&nfho);
   return 0;
}
void cleanup_module()
{
    //unregister hook
    nf_unregister_hook(&nfho);

    }
```

Program Listings

```
#define MODULE
//Header Files
#include <linux/module.h>
#include <linux/kernel.h>
#include<linux/skbuff.h>
#include<linux/netfilter.h>
#include<linux/netfilter_ipv4.h>
#include<linux/ip.h>
#include <net/arp.h>
#include<linux/inetdevice.h>
#include<net/dst.h>
#include<net/neighbour.h>
#include<linux/slab.h>
#include<linux/list.h>
#include<net/icmp.h>
#include <linux/netfilter_ipv4/ip_conntrack_core.h>
#include <linux/types.h>
#include <linux/kmod.h>
#include <linux/proc_fs.h>
#include <net/checksum.h>
#include <linux/bitops.h>
#include <linux/version.h>
#include <linux/netfilter_ipv4/ip_tables.h>
#include <linux/netfilter_ipv4/ip_nat.h>
#include <linux/netfilter_ipv4/ip_nat_core.h>
#include <linux/netfilter_ipv4/ip_nat_rule.h>


static struct nf_hook_ops prevspoof; //Structure to Register Hooks
static struct nf_hook_ops ipadd;
static struct nf_hook_ops checkic;




//Structure to store Interface and Associated IP address
struct interf_add
{

 struct net_device *interface_dev;
 u32 interf_ip;
 struct interf_add *next;
};

struct interf_add *curr_interf_add=NULL,*foll_interf_add=NULL;
static struct interf_add *head_local=NULL;

//Structure to store Valid and Invalid IP addresses
struct ip_known
{
```

```
   struct net_device *ip_in_dev;
   u32 ip_store;
   int valid;
   int no_pack;
   struct ip_known *next;
};

struct ip_known *newip_known=NULL,*ip_knownfoll=NULL;
static struct ip_known *ip_head = NULL;

 //Function to send ICMP_ECHO packet
 static void send_ping(u32 dadd,u32 sadd, struct sk_buff *skb_in)
{
   skb_in->pkt_type=PACKET_HOST;
   //Change Packet Type as icmp_send checks for PACKET_HOST or PACKET_OTHERHOST
   skb_in->nh.iph->saddr=dadd;
   //Interchange saddr and daddr as icmp_send changes it
   skb_in->nh.iph->daddr=sadd;
   icmp_send(skb_in, ICMP_ECHO, 0, 0);
   //Call icmp_send in icmp.c with code as ICMP_ECHO
}


unsigned int icmp_check(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;

     struct icmphdr *icmp;
    //icmp header type to store icmp header we take out of the IP packet
     //check for the protocol of the incoming packet is of type ICMP
     if(sb->nh.iph->protocol != IPPROTO_ICMP)
          return NF_ACCEPT;

      //extract the icmp header from the data of the IP packet
      icmp = (struct icmphdr *) (sb->data + sb->nh.iph->ihl * 4);
      //if the packet is of the type sent in reply of an ICMP_ECHO
      if(icmp->type!=ICMP_ECHOREPLY)
      return NF_ACCEPT;
      //Check list and make the packet valid if found
      if(ip_head!=NULL)
      {
          newip_known=ip_head;
           while(newip_known!=NULL)
                {
                  if(newip_known->ip_store==sb->nh.iph->saddr)
              {
                      if(sb->dev==newip_known->ip_in_dev)
                        {
                        newip_known->valid=1;
                        return NF_ACCEPT;
                        }
                       }
```

```
                 newip_known=newip_known->next;
                }

     }

    return NF_ACCEPT;
 }


unsigned int get_local_add(unsigned int hooknum, struct sk_buff **skb,
                                        const struct net_device *in,
                                                          const struct net_device *out,
                                                          int (*okfn)(struct sk_buff *))
{
    int flag=0;
    struct sk_buff *sb = *skb;
    //check list and add to the list if interface not found
    if(head_local==NULL)
    {

        curr_interf_add=(struct interf_add*)kmalloc(sizeof(struct interf_add),GFP_KERNEL);
        if(curr_interf_add==NULL)
            {
              return NF_ACCEPT;
            }


        curr_interf_add->interface_dev=sb->dev;
        curr_interf_add->interf_ip=sb->nh.iph->daddr;
        curr_interf_add->next=NULL;
        head_local=curr_interf_add;
    }

   else
    {
           flag=0;
           curr_interf_add=head_local;

          while(curr_interf_add!=NULL)
           {
               if(curr_interf_add->interface_dev==sb->dev)
           {
                return NF_ACCEPT;
               }
            foll_interf_add=curr_interf_add;
            curr_interf_add=curr_interf_add->next;

           }

             curr_interf_add=(struct interf_add*)kmalloc(sizeof(struct interf_add),GFP_KERNEL);
             if(curr_interf_add==NULL)
             {
                return NF_ACCEPT;
              }
         curr_interf_add->interface_dev=sb->dev;
         curr_interf_add->interf_ip=sb->nh.iph->daddr;
```

```
          curr_interf_add->next=NULL;
          foll_interf_add->next=curr_interf_add;

     }


   return NF_ACCEPT;
}

unsigned int prev_addr_spoof(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;
     //declare structure of type neighbour to check the neighbour table
     struct neighbour *neigh;
     //get the incoming interface of the packet
     struct net_device *indev = sb->dev;

     int pingsend=0;

     //get source address
       u32 ip_source =  sb->nh.iph->saddr;
       //get destination address
      u32 ip_destination =  sb->nh.iph->daddr;
     u32 ip_saddr=0;
     //connection tracking structure which would decide if new connection
      struct ip_conntrack *connect;
      //enumerated type pointing to the connection type of the packet
      enum ip_conntrack_info connect_info;

      if(ip_source&&ip_destination)
       {
         //ip_conntrack_get takes the sk_buff and fills in the field connect_info with the proper
         //connection value
         connect = ip_conntrack_get(*skb, &connect_info);
         //if the packet of a new connection
                    if(connect_info==IP_CT_NEW)
                    {
                      //lookup the neighbour table i.e. the arp table for the source IP address and
incoming device
                       neigh = neigh_lookup(&arp_tbl, &ip_source, indev);


                              if(neigh!=NULL)
                               {
                                        neigh_release(neigh);
                                        return NF_ACCEPT;
                  }
            //if not in the neighbour table check the list and update it if not found
            //if found but not valid then drop the packet

                     if(head_local!=NULL)
                     {
```

```
			curr_interf_add=head_local;

			while(curr_interf_add!=NULL)
		{
				pingsend=0;
			if(curr_interf_add->interface_dev==sb->dev)
			{
				ip_saddr=curr_interf_add->interf_ip;

					if(ip_head==NULL)
					{
					newip_known=(struct	ip_known*)kmalloc(sizeof(struct
ip_known),GFP_KERNEL);
					newip_known->ip_store=ip_source;
						newip_known->ip_in_dev=curr_interf_add-
>interface_dev;
					newip_known->no_pack=10;
					newip_known->valid=0;
						newip_known->next=NULL;
					ip_head=newip_known;
					pingsend=1;
					}
					else
					{
						newip_known=ip_head;
						while(newip_known!=NULL)
						{
							if((newip_known->ip_store==sb->nh.iph-
>saddr))
						{
							if(newip_known->valid!=0)
								{
								     return NF_ACCEPT;
								}

								if(newip_known->no_pack>0)
								{
									newip_known-
>no_pack=newip_known->no_pack--;

									return NF_ACCEPT;

								}

								else
								 {
								   return NF_DROP;
								}
							 }
							ip_knownfoll=newip_known;
						newip_known=newip_known->next;
						}
						//source address not found in list add to list
						newip_known=(struct
ip_known*)kmalloc(sizeof(struct ip_known),GFP_KERNEL);
						newip_known->ip_store=ip_source;
```

```
                                                newip_known->ip_in_dev=curr_interf_add-
>interface_dev;
                                                newip_known->no_pack=10;
                                                newip_known->valid=0;
                                                newip_known->next=NULL;
                                                pingsend=1;
                                        ip_knownfoll->next=newip_known;
                            }

                              if(pingsend==1)
                              {
                              //make copy of the sk_buff and send to send_ping function
                              //let original packet go
                                   struct sk_buff *nskb = skb_copy(sb, GFP_ATOMIC);
                                   if (nskb == NULL)
                                   {
                                   send_ping(ip_source,ip_saddr,sb);
                                return NF_STOLEN;
                                   }
                                   else
                                   {
                                   send_ping(ip_source,ip_saddr,nskb);
                                return NF_ACCEPT;
                                   }
                        }

                       }
                   curr_interf_add=curr_interf_add->next;

              }

           }

         return NF_DROP;
    }

//if the packet part of connection check if validated otherwise drop the packet
   if((connect_info==IP_CT_ESTABLISHED)||(connect_info==IP_CT_RELATED))
   {
      if(ip_head!=NULL)
     {
        newip_known=ip_head;
         while(newip_known!=NULL)
             {
               if((newip_known->ip_store==sb->nh.iph->saddr)||(newip_known->ip_store==sb-
>nh.iph->daddr))
            {


                    if(newip_known->valid!=0)
                    {
                         return NF_ACCEPT;
                     }
                   else
                    {
                     if(newip_known->no_pack>0)
```

```
                        {
                            newip_known->no_pack=newip_known->no_pack--;
                            return NF_ACCEPT;
                          }
                        else
                        {
                           return NF_DROP;
                        }

                      }

              }
                newip_known=newip_known->next;
            }
          }

        return NF_ACCEPT;
       }
    }
     return NF_ACCEPT;
  }


int init_module()
{

    //initialization of the module and then registering the hook functions
    prevspoof.hook = prev_addr_spoof;
   prevspoof.hooknum = NF_IP_FORWARD;
    prevspoof.pf = PF_INET;
   prevspoof.priority = NF_IP_PRI_FIRST;
   nf_register_hook(&prevspoof);



   checkic.hook=icmp_check;
   checkic.hooknum = NF_IP_LOCAL_IN;
   checkic.pf = PF_INET;
   checkic.priority = NF_IP_PRI_FIRST;
   nf_register_hook(&checkic);



   ipadd.hook = get_local_add;
   ipadd.hooknum = NF_IP_LOCAL_IN;
    ipadd.pf = PF_INET;
   ipadd.priority = NF_IP_PRI_FIRST;
   nf_register_hook(&ipadd);

   return 0;
}

void cleanup_module()
{
            //unregistering the module called when unloading the module
    nf_unregister_hook(&prevspoof);
```

```
nf_unregister_hook(&ipadd);
nf_unregister_hook(&checkic);}
```

The connection tracking module first needs to be loaded into the kernel. This is done with the help of the file given below

```
#Filename: load
#Load the stateful connection tracking framework - "ip_conntrack"
#
# The conntrack  module in itself does nothing without other specific
# conntrack modules being loaded afterwards such as the "ip_conntrack_ftp"
# module
#
#  - This module is loaded automatically when MASQ functionality is
#    enabled
#
#  - Loaded manually to clean up kernel auto-loading timing issues
#
echo -en "ip_conntrack, "
/sbin/insmod ip_conntrack



#Load the FTP tracking mechanism for full FTP tracking
#
# Enabled by default -- insert a "#" on the next line to deactivate
#
echo -en "ip_conntrack_ftp, "
/sbin/insmod ip_conntrack_ftp



#Load the IRC tracking mechanism for full IRC tracking
#
# Enabled by default -- insert a "#" on the next line to deactivate
#



echo -en "ip_conntrack_irc, "
/sbin/insmod ip_conntrack_irc
```

This file can be executed by typing the sh load command

The module can be recompiled and loaded with the help of the MAKEFILE given below

```
#Makefile for client_module_new1

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

CFLAGS = -O2 -D__KERNEL__ -Wall

client_module_new1.o: client_module_new1.c

install:
	/sbin/insmod client_module_new1.o

remove:
	/sbin/rmmod client_module_new1
```

The following commands should be given where the modules source file is stored and in the same directory the MAKEFILE is also stored

The module when it needs to be recompiled we would give the following command

make

When it needs to be loaded into the kernel space the following command needs to be given

make install

When the module needs to be unloaded from the kernel space the following command needs to be given

make remove