CIS644 Lab2 MiniFirewall -Qinyun Zhu

1. Policy Configuration

In my system, policies have the following conditions: direction which specifies the type of direction of a packet, incoming or outgoing, protocol, source IP, destination IP, source IP mask, destination IP mask, source port and destination port. A policy has two kinds of actions: block or unblock. The arguments of the policy configuration tool are defined as follows:

--in the policy is applied to incoming packets

--out the policy is applied to outgoing packets

--proto this argument specifies the protocol of the policy applying to. The parameters should be **TCP, UDP, ICMP or the protocol number**.

--srcip source IP of the packet

--dstip destination IP of the packet

--srcmask source IP mask of the packet

--dstmask destination IP mask of the packet

--srcport source port of the packet. It is only valid when the protocol is **TCP** or **UDP**.

--dstport destination of the packet. It is only valid when the protocol is **TCP** or **UDP**.

--action specify action the filter should do when the rule for the policy is fired. If a parameter was given, the value **block** or **BLOCK** means block the packet matches the policy; **unblock** or **UNBLOCK** tells the filter accept the packet. If the action is not specified, the default action is block.

For all policy conditions mentioned above, you can omit one or give an **all** parameter to it, which means all possible situations are applied in the policy rule.

There are also some control options:

--print print all the rules in the filter

--clear delete all the rules in the filter

--delete delete a rule in the filter. A parameter is required to specify the index of the rule to delete.

1.1. Implementation of the policy

The structure of storing the policy is:

```
struct policy{
    /*direction*/
    u_char direct;
    /*conditions*/
    u_char proto;
    u_long srcip;
    u_long dstip;
    u_long srcmask;
    u_long dstmask;
    union{
    u_short srcport;
    u_short index;   //index of a rule in filter
    };
    u_short dstport;
    /*action*/
    u_short act;
    /*validation*/
    u_char val;
    struct policy* next;
};
```

The **direct** field is used to specify the direction of the packets the policy should apply to and tos tag a control message which is sent to the filter. The value of it can be *DIRECT_IN, DIRECT_OUT* or *DIRECT_CTL*.

The **val** field is used to indicate which field in the policy is valid. Each bit of the field shows the validity of one condition field in the policy. The possible values of field val is defined as:

```
#define VAL_PROTO 1
#define VAL_SRCIP 2
#define VAL_DSTIP 4
#define VAL_SRCM 8
#define VAL_DSTM 16
#define VAL_SRCPORT 32
#define VAL_DSTPORT 64
```

The **index** field is used to specify the index of the rule to delete when the message is a delete command for the filter.

The **action** field has four actions, two for policy and two for command. They are *ACT_BLOCK, ACT_UNBLOCK* for policy actions and *ACT_CTL_DEL and ACT_CTL_CLEAR* for controlling commands.

1.2. Implementation of the Configuration Tool

I use the getopt_long() to pass and configure the commands from user. The tool interprets the commands and their arguments and then it fills the fields of the policy object and set validity bits. The "all" arguments mean that the fields are not valid. The rule matching engine of kernel filter has to ignore them. Finally, the tool writes the object to the /proc/minifirewall "file" to pass the message to the filter in the kernel space. In another case, the tool only reads from the "file" to get the policy rules and print them. Following code, which is used to parse a command argument and execute corresponding function, is part of the main() function.

```
switch(c){

case 0:

        if(vf == CTL_FLAG_PRINT){

                printrules();

                return;

        }

        if(vf == CTL_FLAG_CLEAR){

                rule.direct = DIRECT_CTL;

                rule.act = ACT_CTL_CLEAR;

                writearule(&rule);

                return;

        }
```

```
        if(vf == CTL_FLAG_DELETE){
            if(!optarg)
                printf("ERROR: delete number is required\n");
            rule.direct = DIRECT_CTL;
            rule.act = ACT_CTL_DEL;
            rule.index = (u_short)atoi(optarg);
            writearule(&rule);
            printrules();
            return;
        }
        rule.direct = vf;
        break;
    case 'p':
        if(0 == strcmp(optarg,"all") || 0 == strcmp(optarg,"ALL")) break;
        rule.proto = getproto(optarg);
        rule.val |= VAL_PROTO;
        break;
    case 's':
        if(0 == strcmp(optarg,"all") || 0 == strcmp(optarg,"ALL")) break;
        rule.srcip = (u_long)inet_addr(optarg);
        rule.val |= VAL_SRCIP;
        break;
    case 'd':
        if(0 == strcmp(optarg,"all") || 0 == strcmp(optarg,"ALL")) break;
        rule.dstip = (u_long)inet_addr(optarg);
```

```
            rule.val |= VAL_DSTIP;
            break;
    case 'l':
            if(0 == strcmp(optarg,"all") || 0 == strcmp(optarg,"ALL")) break;
            rule.srcmask = (u_long)inet_addr(optarg);
            rule.val |= VAL_SRCM;
            break;
    case 'k':
            if(0 == strcmp(optarg,"all") || 0 == strcmp(optarg,"ALL")) break;
            rule.dstmask = (u_long)inet_addr(optarg);
            rule.val |= VAL_DSTM;
            break;
    case 'm':
            if(0 == strcmp(optarg,"all") || 0 == strcmp(optarg,"ALL")) break;
            rule.srcport = htons(atoi(optarg));
            rule.val |= VAL_SRCPORT;
            break;
    case 'n':
            if(0 == strcmp(optarg,"all") || 0 == strcmp(optarg,"ALL")) break;
            rule.dstport = htons(atoi(optarg));
            rule.val |= VAL_DSTPORT;
            break;
    case 'a':
            if(-1 == (rule.act = getaction(optarg))){
                    puts("ERROR: illegal action!\n");
```

```
                    return;
                }
                break;
            case '?':
                puts("ERROR: check command format\n");
                break;
            default:
                puts("ERROR\n");
                abort();
            }
    }
    writearule(&rule);   //send the message to the filter
```

The function of writing the message to the virtual file is

```
void writearule(struct policy* prule)
{
    int fd = open("/proc/minifirewall",O_WRONLY);
    if(fd==-1){
  printf("ERROR: file not found.\n");
  return;
 }
    write(fd,(void*)prule,sizeof(struct policy));
    close(fd);
}
```

The function of reading and printing policy rules is

```
void printrules()
```

```
{
            struct policy* buffer;
            int rt,c;
            int fdr=open("/proc/minifirewall",O_RDONLY);
            if(fdr==-1){
       printf("ERROR: file not found.\n");
       return;
 }
            c = 1;
            while(-1 != (rt = read(fdr,buffer,sizeof(struct policy))) && rt != 0){
                                    printf("[%d] ",c++);
                                    printpolicy(buffer);
                                    lseek(fdr,0,SEEK_SET);
            }
            close(fdr);
}
```

2. Filter in the Kernel Space

The program creates a virtual file named "minifirewal" under the /porc to communicate with the user space program in the initializing function of the loadable kernel module. Also, it will register two Netfilter hook NF_INET_PRE_ROUTING and NF_INET_POST_ROUTING to filter the incoming packets and outgoing packets. The filter goes through all the stored policy rules and figure out a proper action. If there were any conflicts amoung the actions of the rules fired, the action of the last fired rule would be the final action. Following code is the initializing code for the virtual file and hooks.

```
proc_entry = create_proc_entry( "minifirewall", 0644, NULL );

if (proc_entry == NULL) {
    ret = -ENOMEM;
    printk(KERN_INFO "minifirewall: Couldn't create proc entry\n");
 } else {
    proc_entry->read_proc = minifirewall_read;
```

```
    proc_entry->write_proc = minifirewall_write;

    printk(KERN_INFO "minifirewall: Module loaded.\n");

           nfho_in.hook = hook_in_func;
           nfho_in.hooknum = NF_INET_PRE_ROUTING;
           nfho_in.pf = PF_INET;
           nfho_in.priority = NF_IP_PRI_FIRST;
           nf_register_hook(&nfho_in);
           nfho_out.hook = hook_out_func;
           nfho_out.hooknum = NF_INET_POST_ROUTING;
           nfho_out.pf = PF_INET;
           nfho_out.priority = NF_IP_PRI_FIRST;
           nf_register_hook(&nfho_out);
 }
```

Following is the callback function minifirewall_write() for the write operating of the virtual file. It reads the message from the buffer and does some operations or appends it to the end of our policy rule list according to the relating fields.

```
newrule = (struct policy *)vmalloc(sizeof(struct policy));

 if(!newrule){
                  printk(KERN_INFO "rule insertion failed!\n");
                  return -ENOSPC;
 }

 if (copy_from_user( newrule, buff, len )) {
   vfree(newrule);
   return -EFAULT;

 }

          //interpret and execute
          //the control commands from the user space
          if(DIRECT_CTL == newrule->direct){
                  struct policy* ptemp;
                  switch(newrule->act){
                          //delete a rule
                          case ACT_CTL_DEL:
                                  ptemp = findrule(inrule,(int)(newrule->index));
                                  deleterule(&inrule,ptemp);
                                  break;
                          //clear all the rules
                          case ACT_CTL_CLEAR:
                                  freerules(inrule);
                                  inrule = 0;
```

```
                break;
        }
        in_next = &inrule;
        vfree(newrule);
        return 0;
    }

    //add a rule to the end of the rule list
    addruletail(&inrule,newrule);
```

The callback function minifirewall_read() fill the output buffer with one policy rule.

```
    memcpy(page,*in_next,len);

    in_next = &((*in_next)->next);
```

where len is defined by

```
const int len = sizeof(struct policy);
```

Following is the most important part of the filter—the function of rule match engine which checks an incoming packet and a policy rule and decide if the patterns of the packet match the policy rule. Each condition checking in this function first checks the validity of relating fields and then checks the value in the packet and rule.

```
//match engine
//if the packet can fire the rule then return 1 else return 0
int matchrule(struct policy* rule, struct sk_buff *sock_buff, int direct)
{
    //if any condition fails in the rule then return 0 else 1
    //check validation of rule, incoming buffer and rule direction type
    if(!rule || !sock_buff || direct != rule->direct)
        return 0;

    //check conditions about IP layer
    ip_header = (struct iphdr *)skb_network_header(sock_buff);

    //check source ip address without netmask
    if((rule->val & VAL_SRCIP) && !(rule->val & VAL_SRCM)
        && rule->srcip != (u_long)(ip_header->saddr))
        return 0;

    //check source subnet with netmask
    if((rule->val & VAL_SRCIP) && (rule->val & VAL_SRCM)
        && rule->srcip != ((u_long)(ip_header->saddr) & rule->srcmask))
        return 0;

    //check destination ip address without netmask
```

```
	if((rule->val & VAL_DSTIP) && !(rule->val & VAL_DSTM)
		&& rule->dstip != (u_long)(ip_header->daddr))
		return 0;

	//check source subnet with netmask
	if((rule->val & VAL_DSTIP) && (rule->val & VAL_DSTM)
		&& rule->dstip != ((u_long)(ip_header->daddr) & rule->dstmask))
		return 0;

	//check upper layer protocol
	if((rule->val & VAL_PROTO) && rule->proto != (u_char)(ip_header->protocol))
		return 0;

	//check ports if it is TCP
	if(rule->proto == IPPROTO_TCP){
		tcp_header = (struct tcphdr *)skb_transport_header(sock_buff);
		if((rule->val & VAL_SRCPORT) && rule->srcport != (u_short)(tcp_header->source))
			return 0;
		if((rule->val & VAL_DSTPORT) && rule->dstport != (u_short)(tcp_header->dest))
			return 0;
	}
		//check ports if it is UDP
	if(rule->proto == IPPROTO_UDP){
		udp_header = (struct udphdr *)skb_transport_header(sock_buff);
		if((rule->val & VAL_SRCPORT) && rule->srcport != (u_short)(udp_header-
>source))
			return 0;
		if((rule->val & VAL_DSTPORT) && rule->dstport != (u_short)(udp_header->dest))
			return 0;
	}

	printk("rule fired: direct:%d, proto:%d, src:%d, dst:%d, act %d\n",direct,rule-
>proto,ip_header->saddr,ip_header->daddr,rule->act);
	return 1;
}
```

In the hook functions hook_in_func() and hook_out_func(), the following code goes through the policy rule list with a captured packet to find out the proper action. The action of the last fired rule will be the action taken by the filter.

```
	while(rhd != 0){
		if(matchrule(rhd,skb,DIRECT_IN)){
			act = getaction(rhd->act);
		}
		rhd = rhd->next;
	}
```

Where act is initialized as

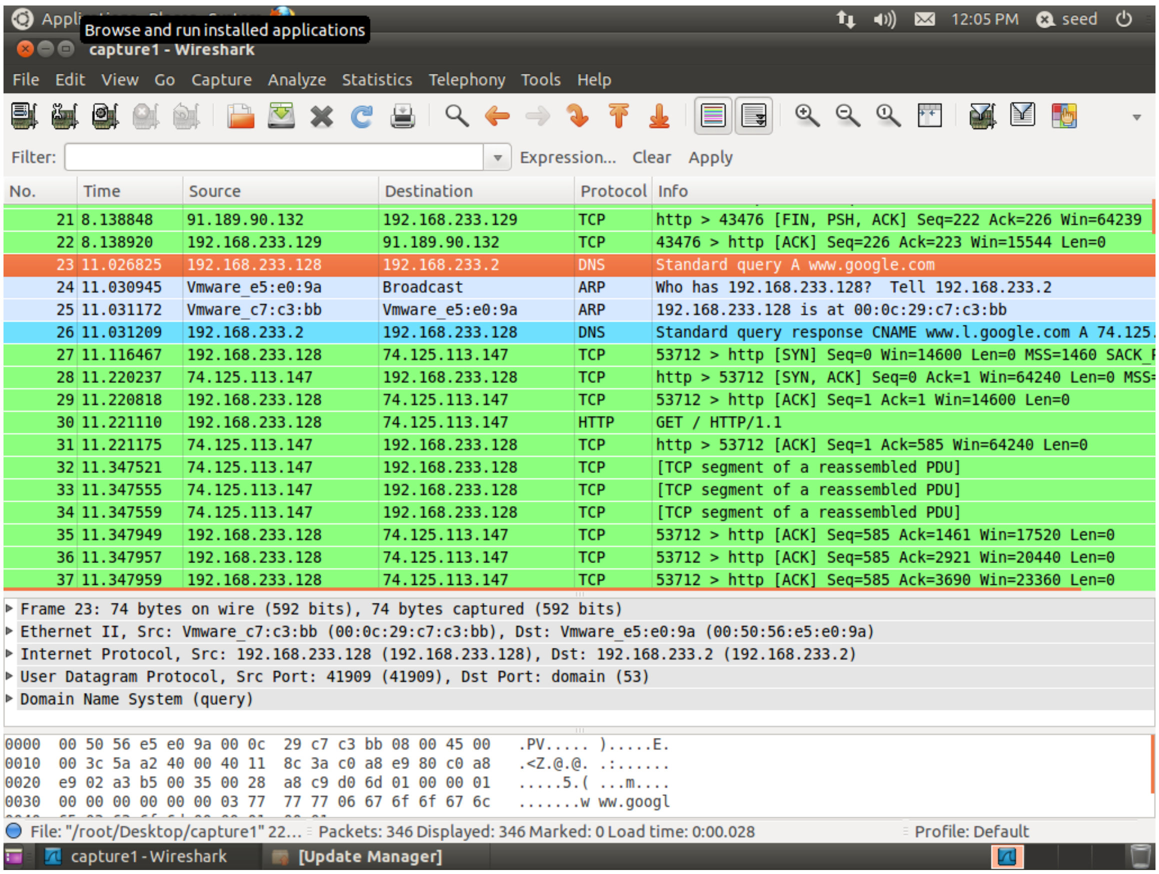
```
	unsigned int act = NF_ACCEPT;
```

3. Tests of the Functionalities

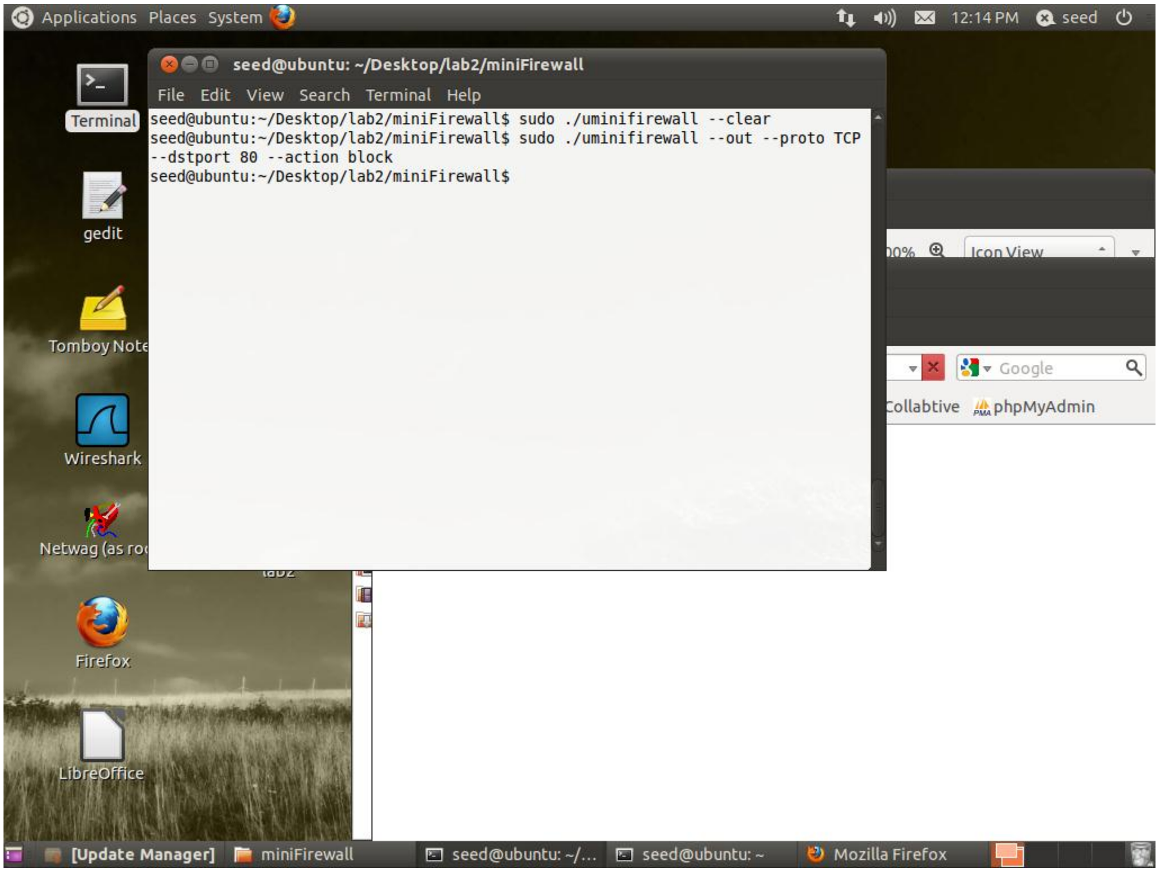
Following are some tests of the typical functionalities of minifirewall.

3.1. Block outgoing packets with specific protocol and specific destination port

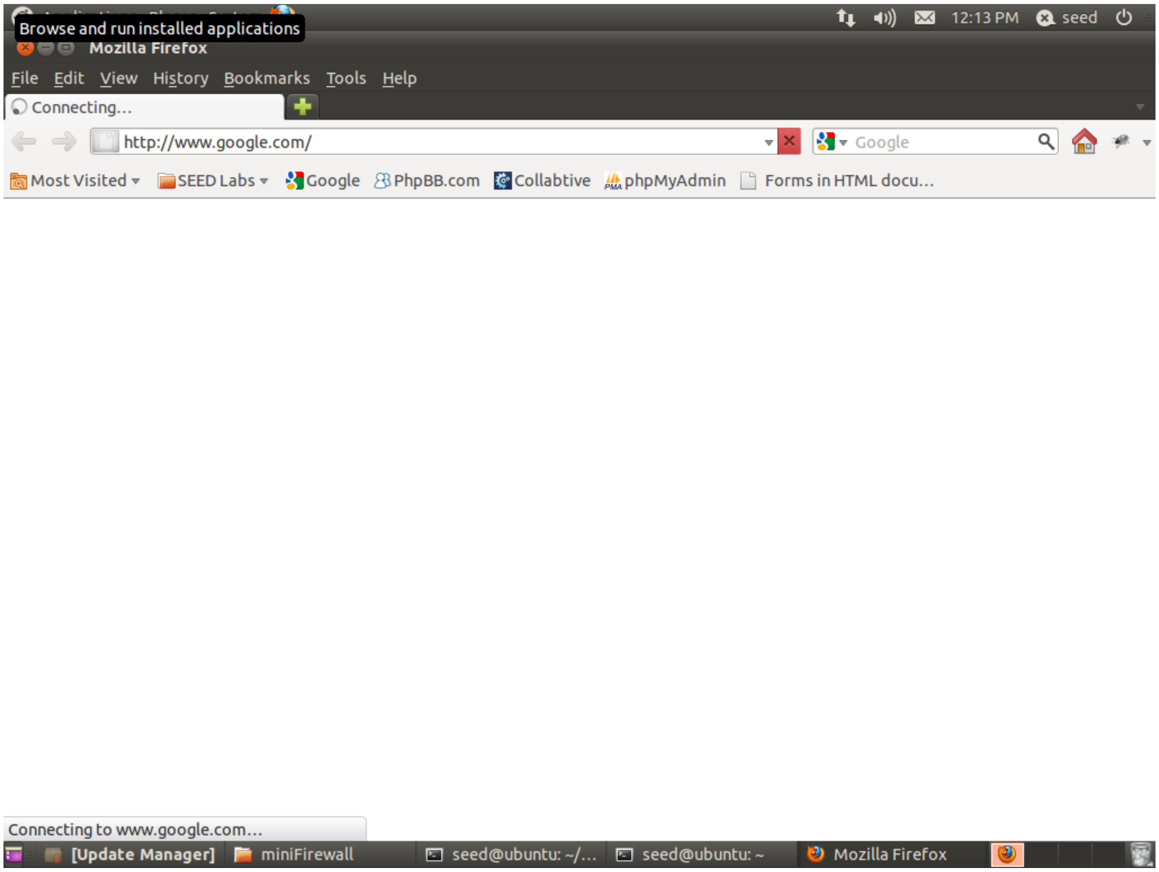
Before setting the policy rule, I captured the packets via a witness virtual machine when our testing computer tried to connect to www.google.com. They can communicate normally (the IP address of our machine is 192.168.233.128).



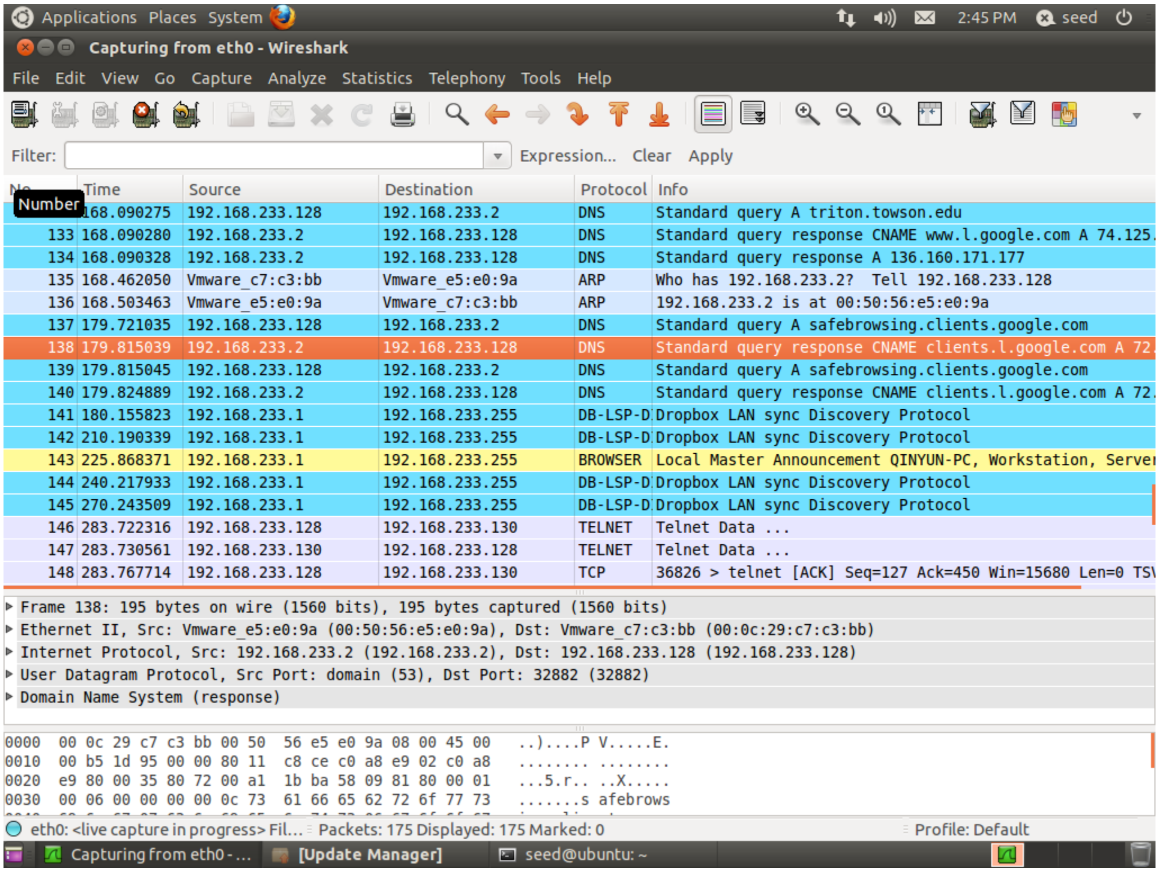
Then I set the policy rule "—out –proto TCP –dstport 80 –action block" through our configuration tool.



Then we can notice that we cannot access to www.google.com any more.

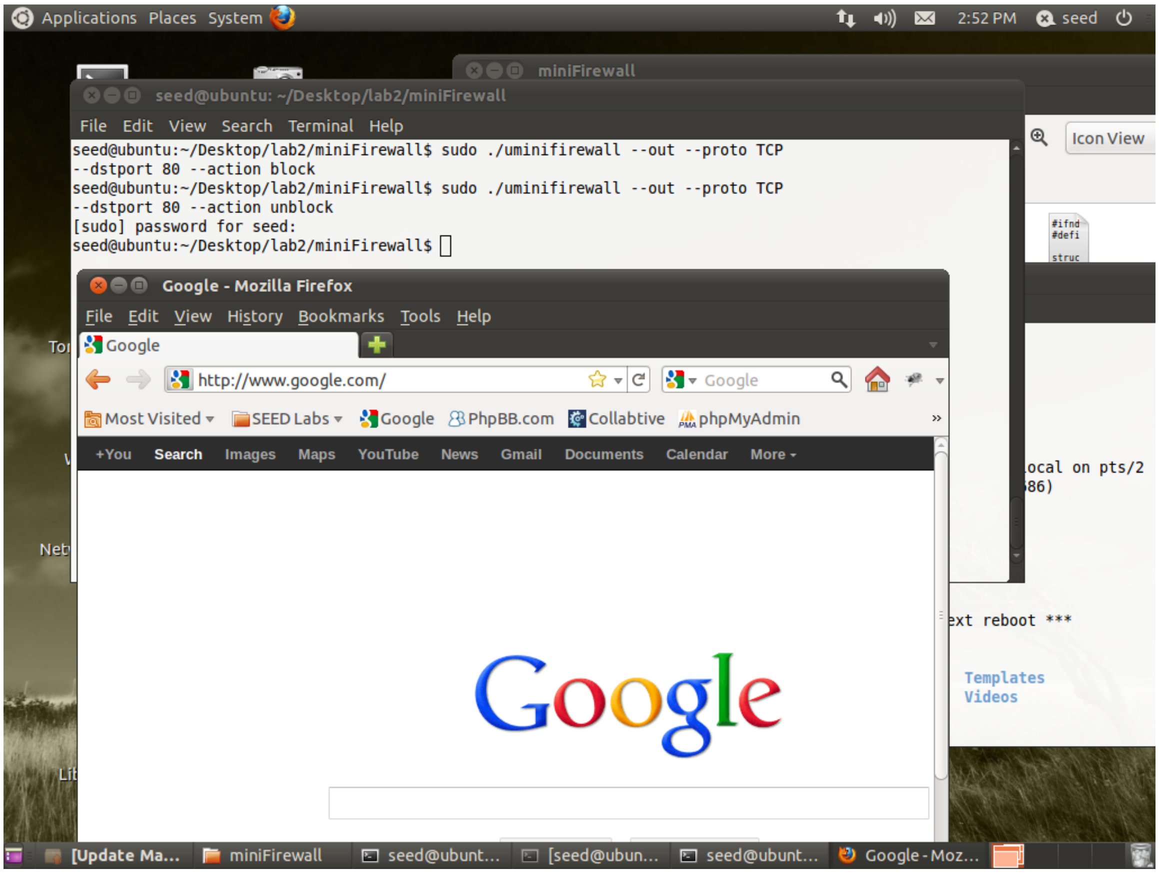


I captured the packets on the witness machine showing that no outgoing packets of heading port 80 from our experiment machine can be detected. However, other TCP communications worked normally.



3.2. Block incoming packets from specific subnet

First, we unblock the packets of last policy rule via the rule "—out --proto TCP –dstport 80 –action unblock". We can see the website can be accessed normally.

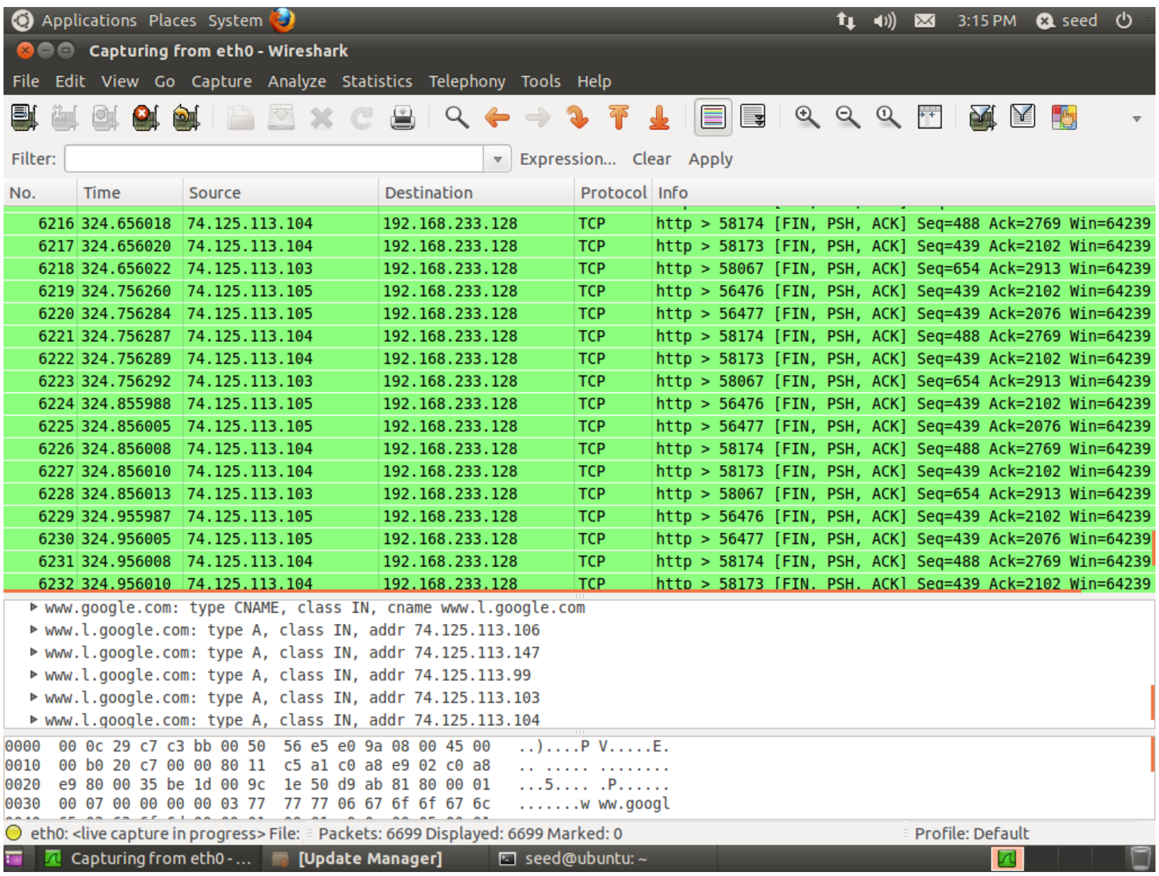


From the DNS responses, we can find that the IPs of server of goole.com are all from the subnet 72.125.113.0/24. Therefore, we can try to block them to test our firewall.

I set the rule "—in –srcmask 255.255.255.0 –srcip 72.125.113.0 –action block".

Then I refreshed the webpage. We can see our browser cannot successfully connect to www.google.com. After I stopped the browser, in our witness machine we saw a lot of traffic from different IPs in 72.125.113.0/24. But they did not get a final ACK from our experiment machine. So the servers kept sending tons of same packets to us.

In the low part of the window, it is the content of a DNS response from a name server of google.

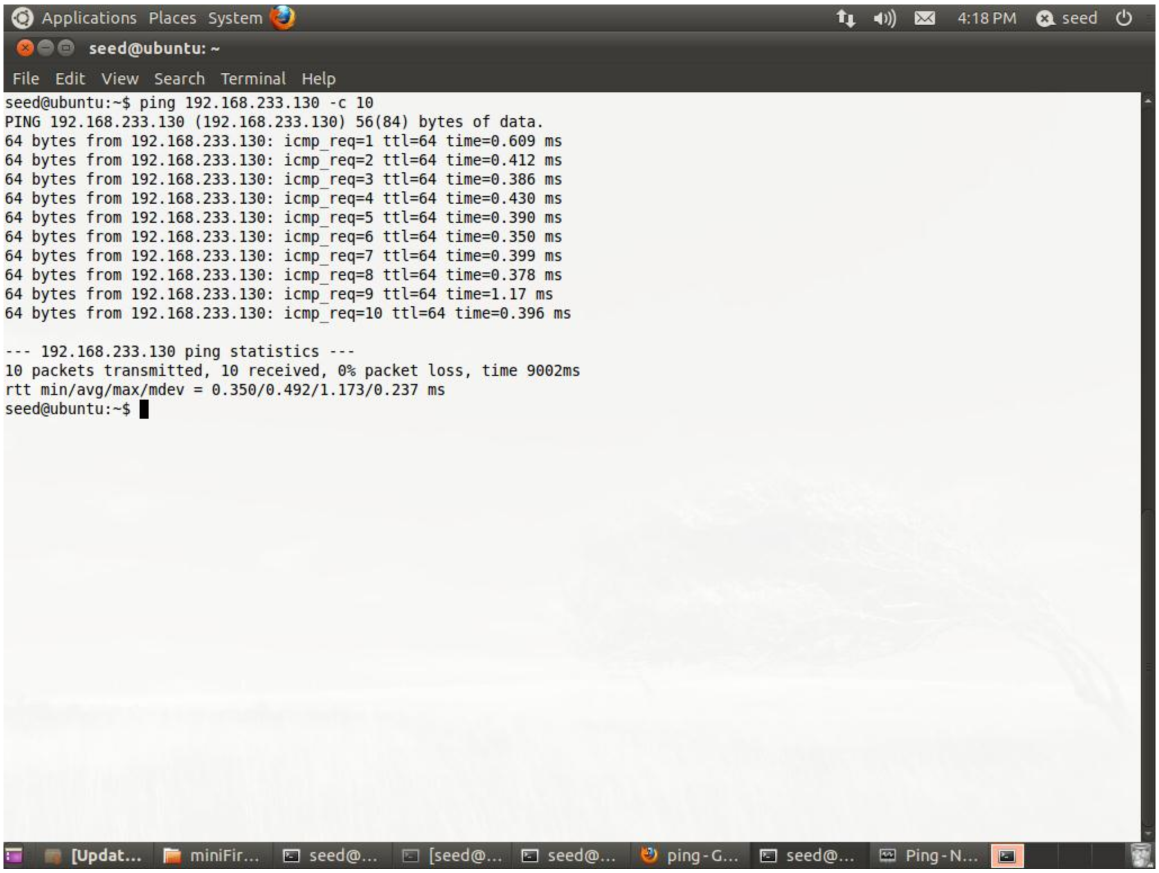


In the end, google's safebrowsing name server sent another set of IPs in different subnet prefix and finished the communication with our machine.

3.3. Block outgoing packets and incoming packets by specific IP and protocol

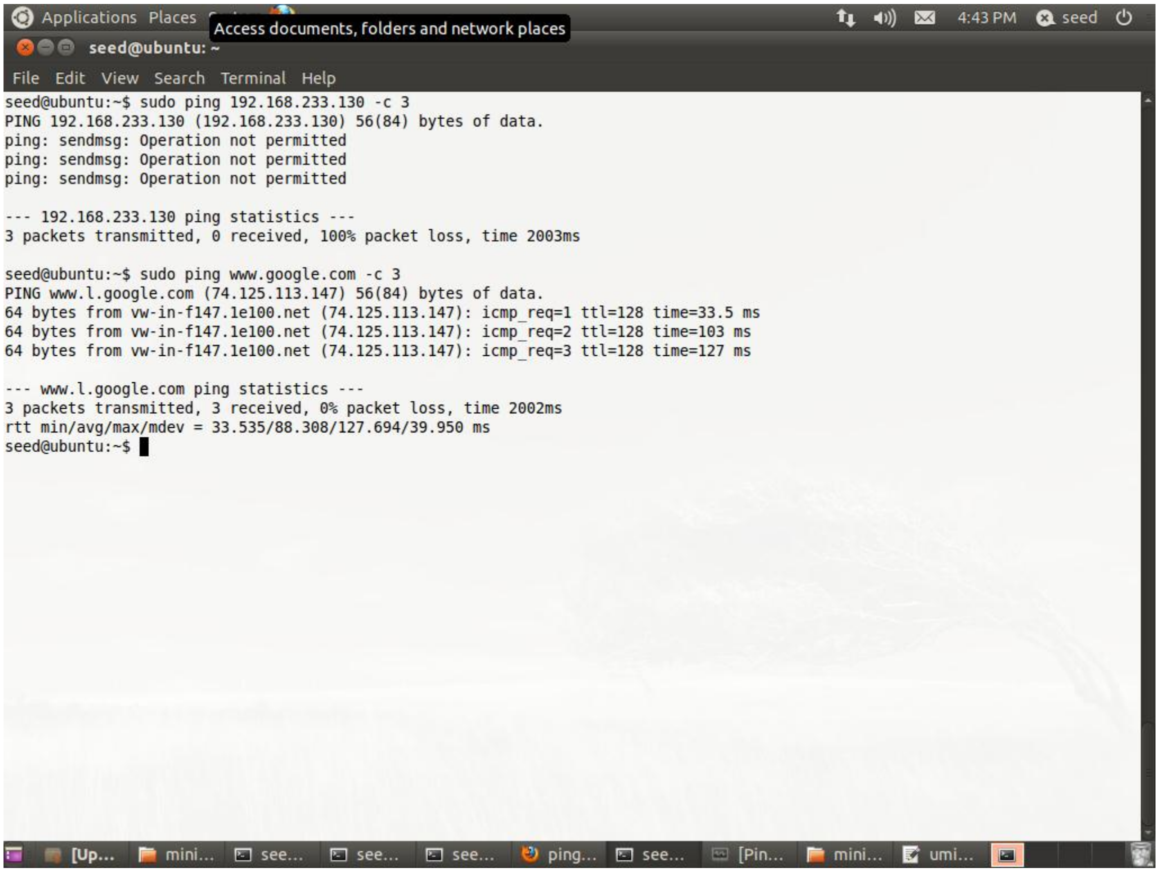
Unblock all packets by rule “—in –proto all –action unblock” and “—out –proto all –action block”.

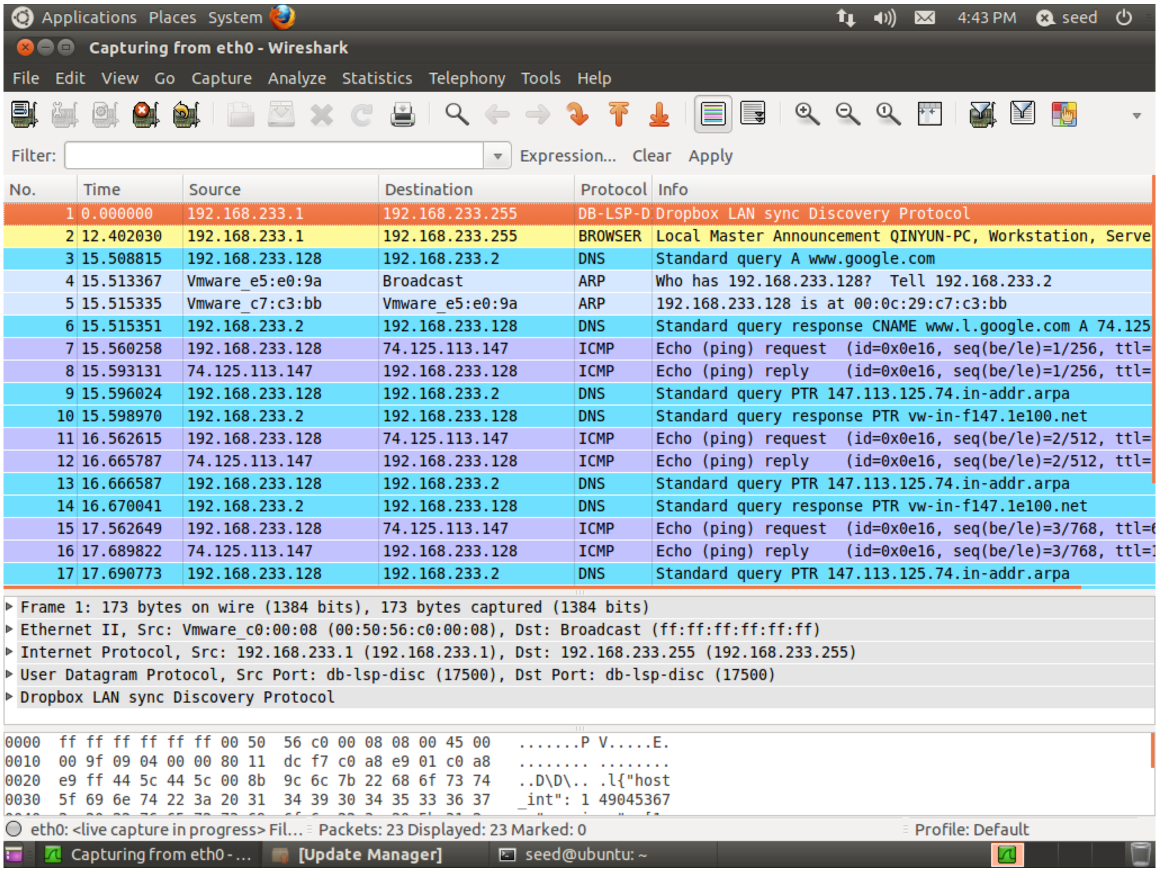
Then I ping the machine 192.168.233.130, the communication is normal.



Then I set the rule "—out –proto ICMP –dstip 192.168.233.130".

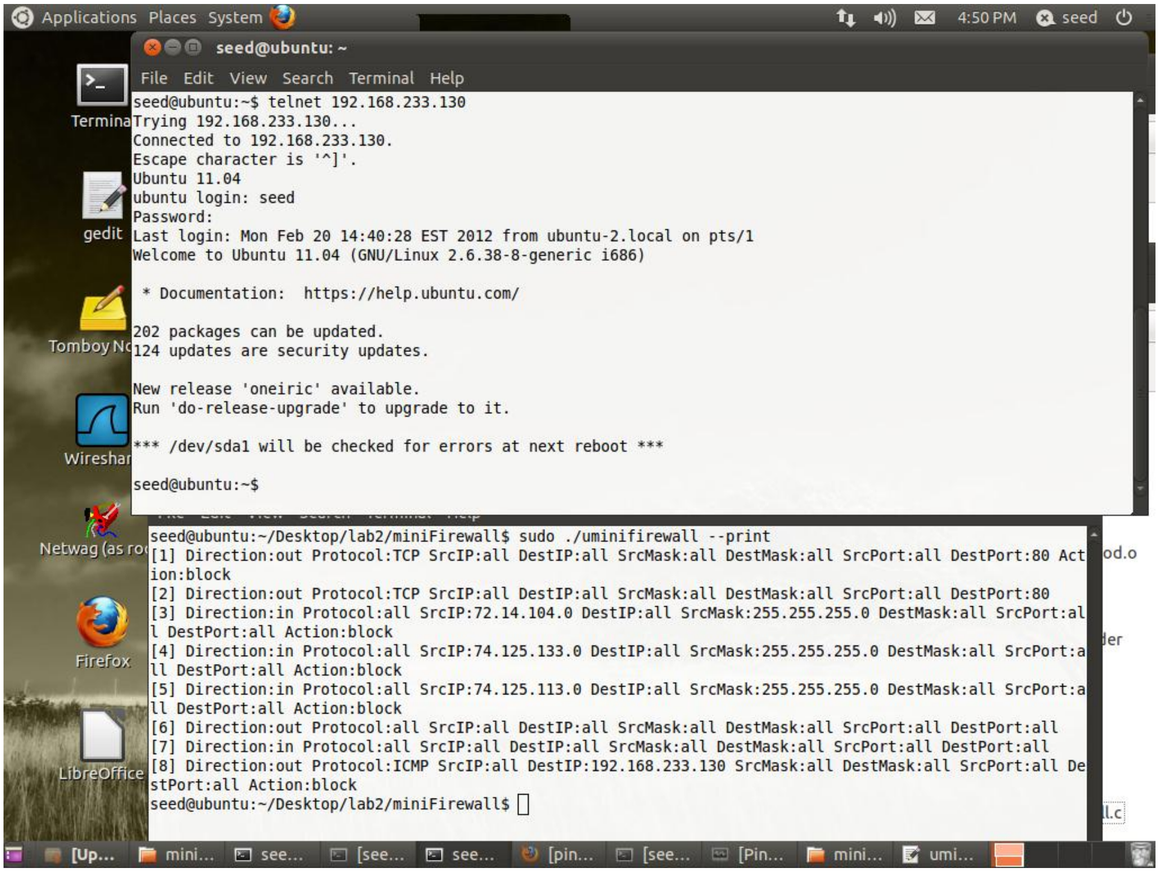
I used command "ping 192.168.233.130 –c 3" and "ping www.google.com –c 3" to test the rule and got the following results.



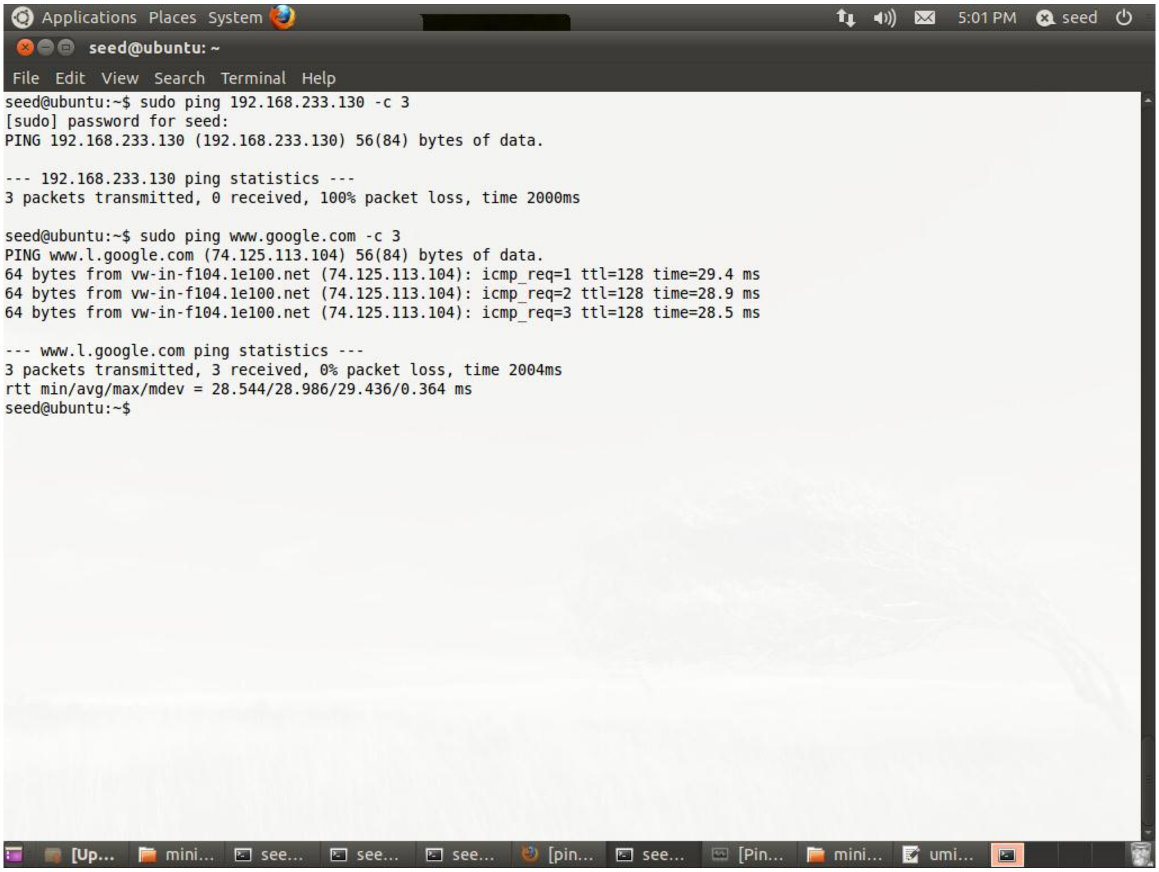


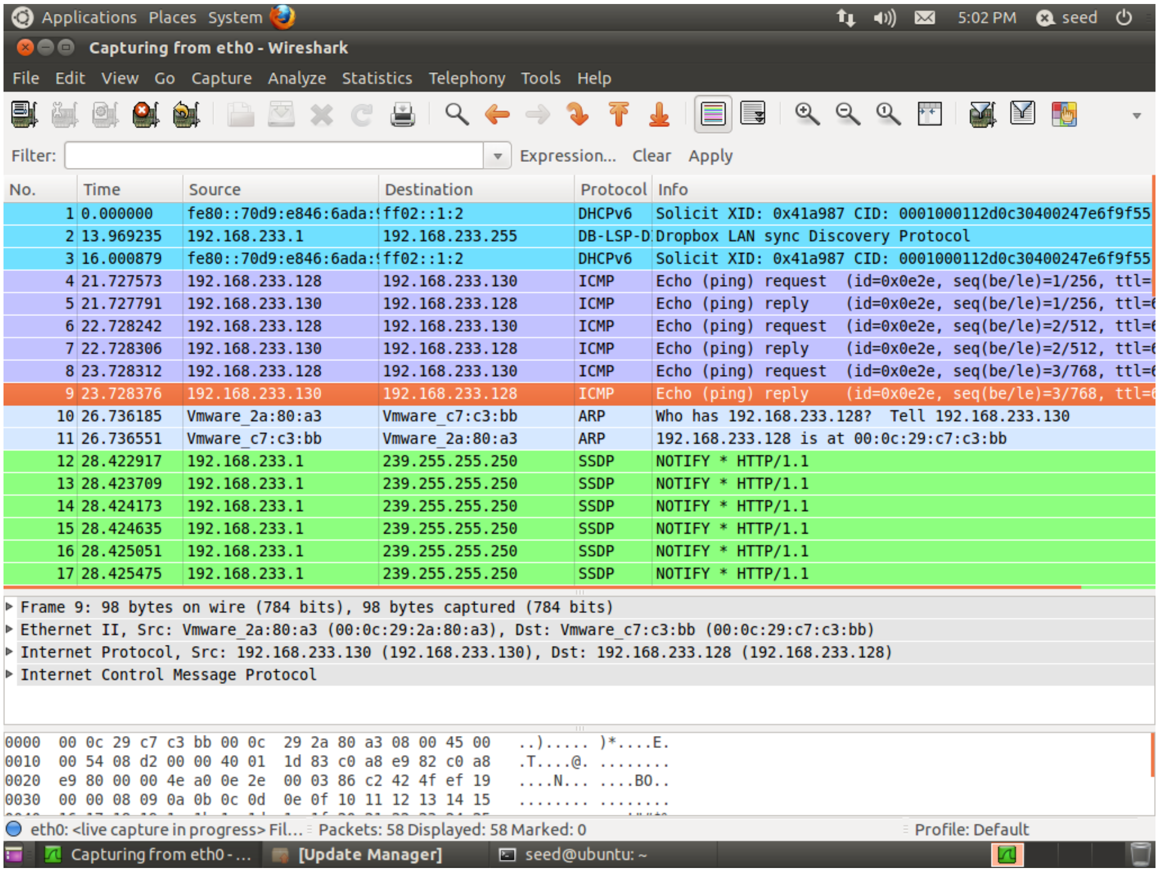
We cannot see any outgoing ICMP packets to 192.168.233.130 but packets to goole.com.

And I can connect to 192.168.233.130 by telnet when the rule is effective (see the last rule showing in the lower window).



Then I unblocked all packets again and set the rule "—in –proto ICMP –srcip 192.168.233.130 –action block". And the results of ping experiment are:

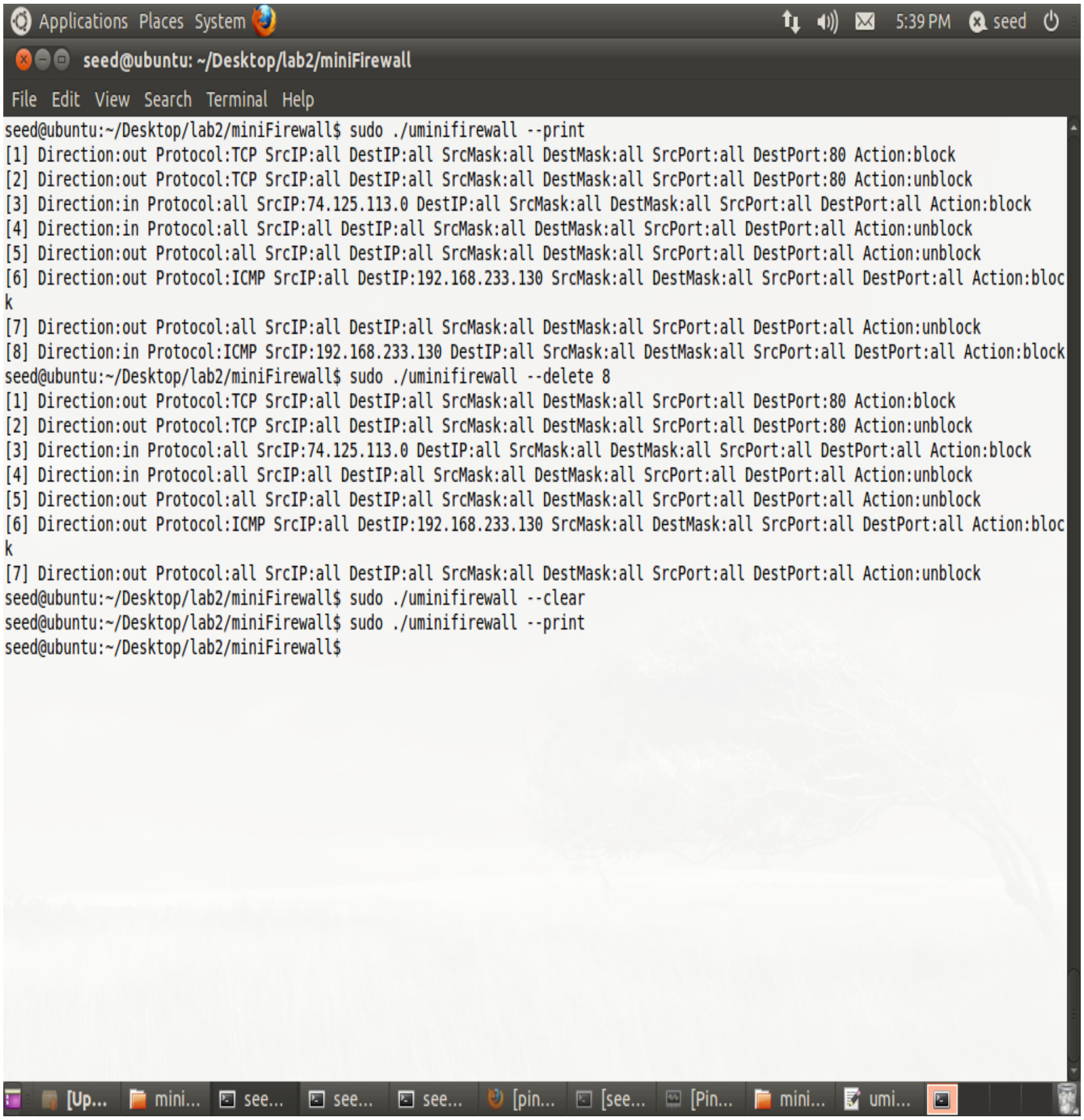

Applications Places System
[redacted]
5:01 PM seed
seed@ubuntu: ~
File Edit View Search Terminal Help
seed@ubuntu:~$ sudo ping 192.168.233.130 -c 3
[sudo] password for seed:
PING 192.168.233.130 (192.168.233.130) 56(84) bytes of data.
--- 192.168.233.130 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2000ms
seed@ubuntu:~$ sudo ping www.google.com -c 3
PING www.l.google.com (74.125.113.104) 56(84) bytes of data.
64 bytes from vw-in-f104.1e100.net (74.125.113.104): icmp_req=1 ttl=128 time=29.4 ms
64 bytes from vw-in-f104.1e100.net (74.125.113.104): icmp_req=2 ttl=128 time=28.9 ms
64 bytes from vw-in-f104.1e100.net (74.125.113.104): icmp_req=3 ttl=128 time=28.5 ms
--- www.l.google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 28.544/28.986/29.436/0.364 ms
seed@ubuntu:~$
[Up... mini... see... see... see... [pin... see... [Pin... mini... umi...



We can know from the results above that all ICMP packets which went to 192.168.233.130 were sent and it did reply it. But our minifirewall blocked the packets.

3.4. Controlling Commands

Following screen dump shows the functions of "--print", "—delete " and "--clear" operation on the policy rules.



First step, the "--print" command let it show all 8 rules I had entered.

Second step, the "—delete 8" command deleted the last rule and showed the 7 rules left.

Third step, the "--clear" command cleared the policy rule list. In other words, it deleted all rules.

Finally, the "--print" command printed all remaining rules, which verified that the previous command indeed cleared the rule list.