

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 to 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 among 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;
}

printf("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).

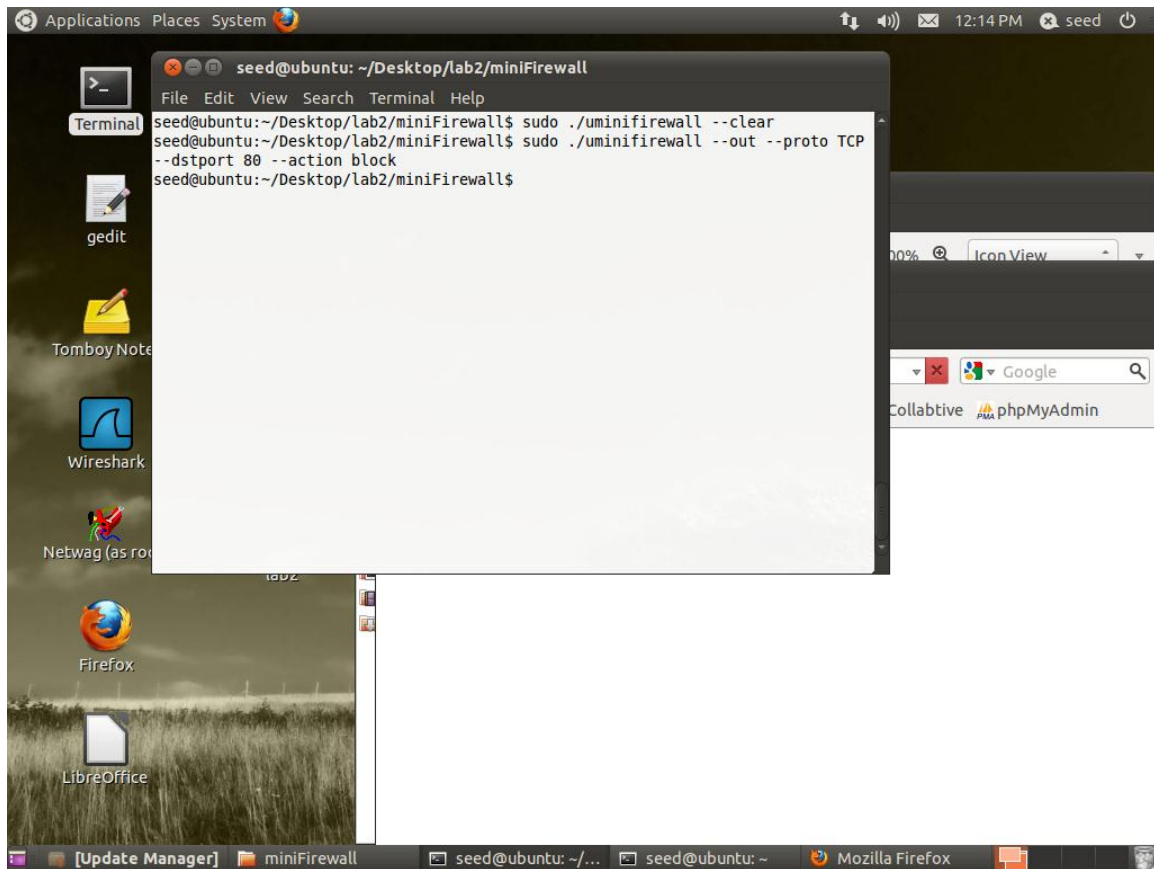
The image shows a Wireshark packet capture window titled 'capture1 - Wireshark'. The interface includes a menu bar (File, Edit, View, Go, Capture, Analyze, Statistics, Telephony, Tools, Help), a toolbar with various icons, and a filter bar. The main display area shows a list of captured packets with columns for No., Time, Source, Destination, Protocol, and Info. Packet 23 is highlighted, showing a DNS standard query for www.google.com. Below the packet list, the packet details pane shows the structure of packet 23: Ethernet II, Internet Protocol, User Datagram Protocol, and Domain Name System (query). The packet bytes pane shows the raw data in hexadecimal and ASCII.

No.	Time	Source	Destination	Protocol	Info
21	8.138848	91.189.90.132	192.168.233.129	TCP	http > 43476 [FIN, PSH, ACK] Seq=222 Ack=226 Win=64239
22	8.138920	192.168.233.129	91.189.90.132	TCP	43476 > http [ACK] Seq=226 Ack=223 Win=15544 Len=0
23	11.026825	192.168.233.128	192.168.233.2	DNS	Standard query A www.google.com
24	11.030945	Vmware_e5:e0:9a	Broadcast	ARP	Who has 192.168.233.128? Tell 192.168.233.2
25	11.031172	Vmware_c7:c3:bb	Vmware_e5:e0:9a	ARP	192.168.233.128 is at 00:0c:29:c7:c3:bb
26	11.031209	192.168.233.2	192.168.233.128	DNS	Standard query response CNAME www.l.google.com A 74.125.
27	11.116467	192.168.233.128	74.125.113.147	TCP	53712 > http [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK
28	11.220237	74.125.113.147	192.168.233.128	TCP	http > 53712 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=
29	11.220818	192.168.233.128	74.125.113.147	TCP	53712 > http [ACK] Seq=1 Ack=1 Win=14600 Len=0
30	11.221110	192.168.233.128	74.125.113.147	HTTP	GET / HTTP/1.1
31	11.221175	74.125.113.147	192.168.233.128	TCP	http > 53712 [ACK] Seq=1 Ack=585 Win=64240 Len=0
32	11.347521	74.125.113.147	192.168.233.128	TCP	[TCP segment of a reassembled PDU]
33	11.347555	74.125.113.147	192.168.233.128	TCP	[TCP segment of a reassembled PDU]
34	11.347559	74.125.113.147	192.168.233.128	TCP	[TCP segment of a reassembled PDU]
35	11.347949	192.168.233.128	74.125.113.147	TCP	53712 > http [ACK] Seq=585 Ack=1461 Win=17520 Len=0
36	11.347957	192.168.233.128	74.125.113.147	TCP	53712 > http [ACK] Seq=585 Ack=2921 Win=20440 Len=0
37	11.347959	192.168.233.128	74.125.113.147	TCP	53712 > http [ACK] Seq=585 Ack=3690 Win=23360 Len=0

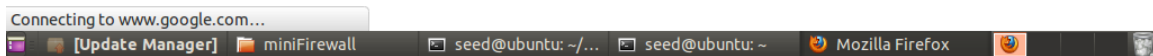
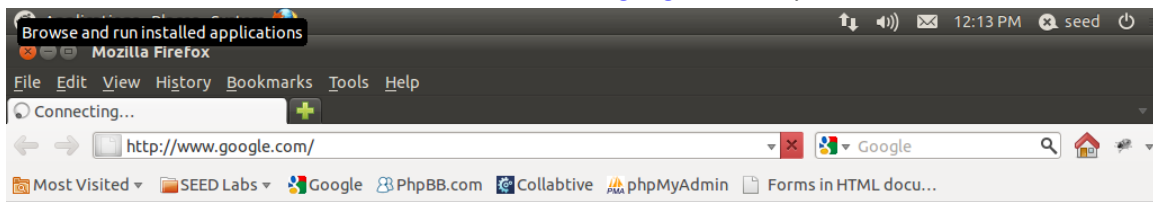
Frame 23: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
Ethernet II, Src: Vmware_c7:c3:bb (00:0c:29:c7:c3:bb), Dst: Vmware_e5:e0:9a (00:50:56:e5:e0:9a)
Internet Protocol, Src: 192.168.233.128 (192.168.233.128), Dst: 192.168.233.2 (192.168.233.2)
User Datagram Protocol, Src Port: 41909 (41909), Dst Port: domain (53)
Domain Name System (query)

0000 00 50 56 e5 e0 9a 00 0c 29 c7 c3 bb 00 00 45 00 .PV.....).....E.
0010 00 3c 5a a2 40 00 00 11 8c 3a c0 a8 e9 80 c0 a8 .<Z.@.@.
0020 e9 02 a3 b5 00 35 00 28 a8 c9 d0 6d 01 00 00 015.(...m....
0030 00 00 00 00 00 00 03 77 77 77 06 6f 6f 6f 6cw ww.googl

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.

Applications Places System

Capturing from eth0 - Wireshark

File Edit View Go Capture Analyze Statistics Telephony Tools Help

Filter: Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Info
132	168.090275	192.168.233.128	192.168.233.2	DNS	Standard query A triton.towson.edu
133	168.090280	192.168.233.2	192.168.233.128	DNS	Standard query response CNAME www.l.google.com A 74.125.
134	168.090328	192.168.233.2	192.168.233.128	DNS	Standard query response A 136.160.171.177
135	168.462050	Vmware_c7:c3:bb	Vmware_e5:e0:9a	ARP	Who has 192.168.233.2? Tell 192.168.233.128
136	168.503463	Vmware_e5:e0:9a	Vmware_c7:c3:bb	ARP	192.168.233.2 is at 00:50:56:e5:e0:9a
137	179.721035	192.168.233.128	192.168.233.2	DNS	Standard query A safebrowsing.clients.google.com
138	179.815039	192.168.233.2	192.168.233.128	DNS	Standard query response CNAME clients.l.google.com A 72.
139	179.815045	192.168.233.128	192.168.233.2	DNS	Standard query A safebrowsing.clients.google.com
140	179.824889	192.168.233.2	192.168.233.128	DNS	Standard query response CNAME clients.l.google.com A 72.
141	180.155823	192.168.233.1	192.168.233.255	DB-LSP-D	Dropbox LAN sync Discovery Protocol
142	210.190339	192.168.233.1	192.168.233.255	DB-LSP-D	Dropbox LAN sync Discovery Protocol
143	225.868371	192.168.233.1	192.168.233.255	BROWSER	Local Master Announcement QINYUN-PC, Workstation, Server
144	240.217933	192.168.233.1	192.168.233.255	DB-LSP-D	Dropbox LAN sync Discovery Protocol
145	270.243509	192.168.233.1	192.168.233.255	DB-LSP-D	Dropbox LAN sync Discovery Protocol
146	283.722316	192.168.233.128	192.168.233.130	TELNET	Telnet Data ...
147	283.730561	192.168.233.130	192.168.233.128	TELNET	Telnet Data ...
148	283.767714	192.168.233.128	192.168.233.130	TCP	36826 > telnet [ACK] Seq=127 Ack=450 Win=15680 Len=0 TSV

Frame 138: 195 bytes on wire (1560 bits), 195 bytes captured (1560 bits)

Ethernet II, Src: Vmware_e5:e0:9a (00:50:56:e5:e0:9a), Dst: Vmware_c7:c3:bb (00:0c:29:c7:c3:bb)

Internet Protocol, Src: 192.168.233.2 (192.168.233.2), Dst: 192.168.233.128 (192.168.233.128)

User Datagram Protocol, Src Port: domain (53), Dst Port: 32882 (32882)

Domain Name System (response)

0000 00 0c 29 c7 c3 bb 00 50 56 e5 e0 9a 08 00 45 00 ..)....P V.....E.

0010 00 b5 1d 95 00 00 80 11 c8 ce c0 a8 e9 02 c0 a8X.....

0020 e9 80 00 35 80 72 00 a1 1b ba 58 09 81 80 00 01 ...S.r...X.....

0030 00 06 00 00 00 00 0c 73 61 66 65 62 72 6f 77 73s afebrows

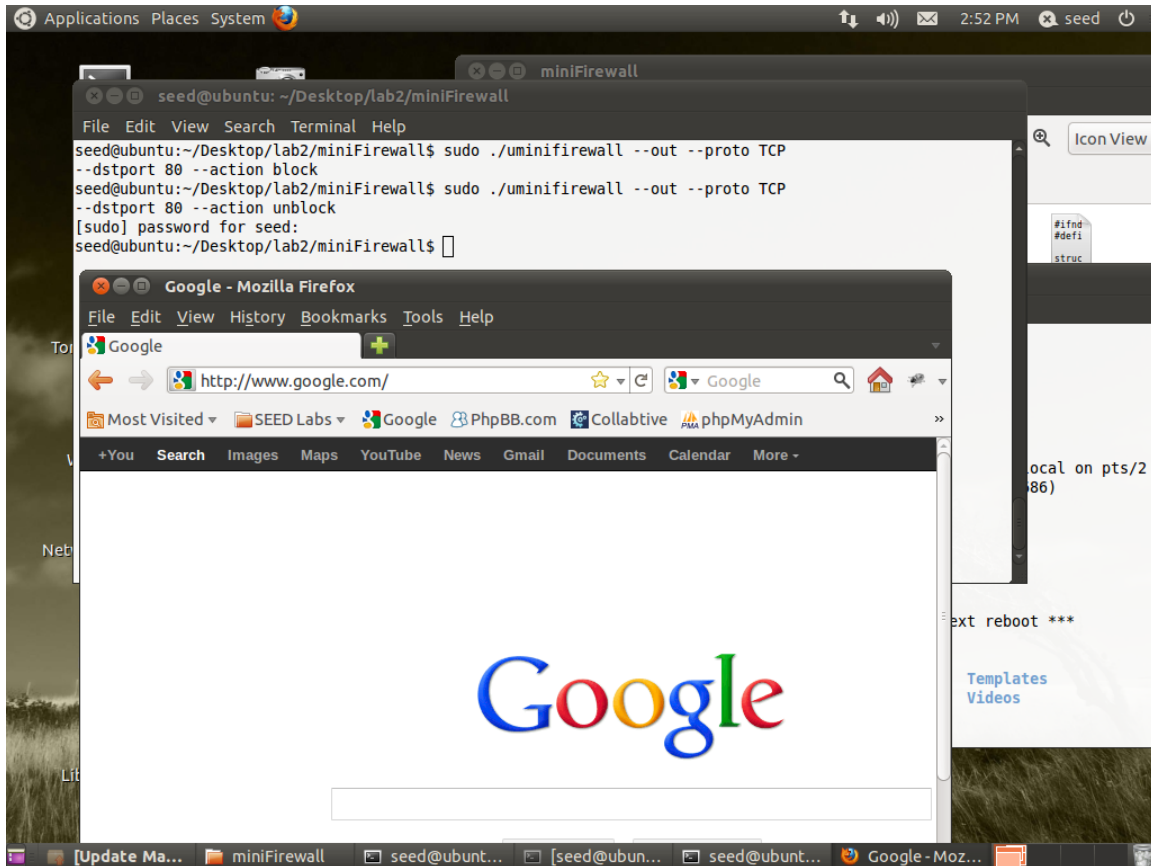
eth0: <live capture in progress> Fil... Packets: 175 Displayed: 175 Marked: 0

Profile: Default

Capturing from eth0 - ... [Update Manager] seed@ubuntu: ~

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 google.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.

The image shows a Wireshark packet capture window titled "Capturing from eth0 - Wireshark". The interface includes a menu bar (File, Edit, View, Go, Capture, Analyze, Statistics, Telephony, Tools, Help), a toolbar with various icons, and a filter bar. The main display area shows a list of captured packets with columns for No., Time, Source, Destination, Protocol, and Info. The packets are filtered by "Filter: Expression... Clear Apply".

No.	Time	Source	Destination	Protocol	Info
6216	324.656018	74.125.113.104	192.168.233.128	TCP	http > 58174 [FIN, PSH, ACK] Seq=488 Ack=2769 Win=64239
6217	324.656020	74.125.113.104	192.168.233.128	TCP	http > 58173 [FIN, PSH, ACK] Seq=439 Ack=2102 Win=64239
6218	324.656022	74.125.113.103	192.168.233.128	TCP	http > 58067 [FIN, PSH, ACK] Seq=654 Ack=2913 Win=64239
6219	324.756260	74.125.113.105	192.168.233.128	TCP	http > 56476 [FIN, PSH, ACK] Seq=439 Ack=2102 Win=64239
6220	324.756284	74.125.113.105	192.168.233.128	TCP	http > 56477 [FIN, PSH, ACK] Seq=439 Ack=2076 Win=64239
6221	324.756287	74.125.113.104	192.168.233.128	TCP	http > 58174 [FIN, PSH, ACK] Seq=488 Ack=2769 Win=64239
6222	324.756289	74.125.113.104	192.168.233.128	TCP	http > 58173 [FIN, PSH, ACK] Seq=439 Ack=2102 Win=64239
6223	324.756292	74.125.113.103	192.168.233.128	TCP	http > 58067 [FIN, PSH, ACK] Seq=654 Ack=2913 Win=64239
6224	324.855988	74.125.113.105	192.168.233.128	TCP	http > 56476 [FIN, PSH, ACK] Seq=439 Ack=2102 Win=64239
6225	324.856005	74.125.113.105	192.168.233.128	TCP	http > 56477 [FIN, PSH, ACK] Seq=439 Ack=2076 Win=64239
6226	324.856008	74.125.113.104	192.168.233.128	TCP	http > 58174 [FIN, PSH, ACK] Seq=488 Ack=2769 Win=64239
6227	324.856010	74.125.113.104	192.168.233.128	TCP	http > 58173 [FIN, PSH, ACK] Seq=439 Ack=2102 Win=64239
6228	324.856013	74.125.113.103	192.168.233.128	TCP	http > 58067 [FIN, PSH, ACK] Seq=654 Ack=2913 Win=64239
6229	324.955987	74.125.113.105	192.168.233.128	TCP	http > 56476 [FIN, PSH, ACK] Seq=439 Ack=2102 Win=64239
6230	324.956005	74.125.113.105	192.168.233.128	TCP	http > 56477 [FIN, PSH, ACK] Seq=439 Ack=2076 Win=64239
6231	324.956008	74.125.113.104	192.168.233.128	TCP	http > 58174 [FIN, PSH, ACK] Seq=488 Ack=2769 Win=64239
6232	324.956010	74.125.113.104	192.168.233.128	TCP	http > 58173 [FIN, PSH, ACK] Seq=439 Ack=2102 Win=64239

Below the packet list, the details pane shows the content of a DNS response from a name server of google. The response is for the query "www.google.com" and contains several A records with IP addresses in the 72.125.113.0/24 subnet.

```
▶ www.google.com: type CNAME, class IN, cname www.l.google.com
▶ www.l.google.com: type A, class IN, addr 74.125.113.106
▶ www.l.google.com: type A, class IN, addr 74.125.113.147
▶ www.l.google.com: type A, class IN, addr 74.125.113.99
▶ www.l.google.com: type A, class IN, addr 74.125.113.103
▶ www.l.google.com: type A, class IN, addr 74.125.113.104
```

The packet data is shown in hexadecimal and ASCII format:

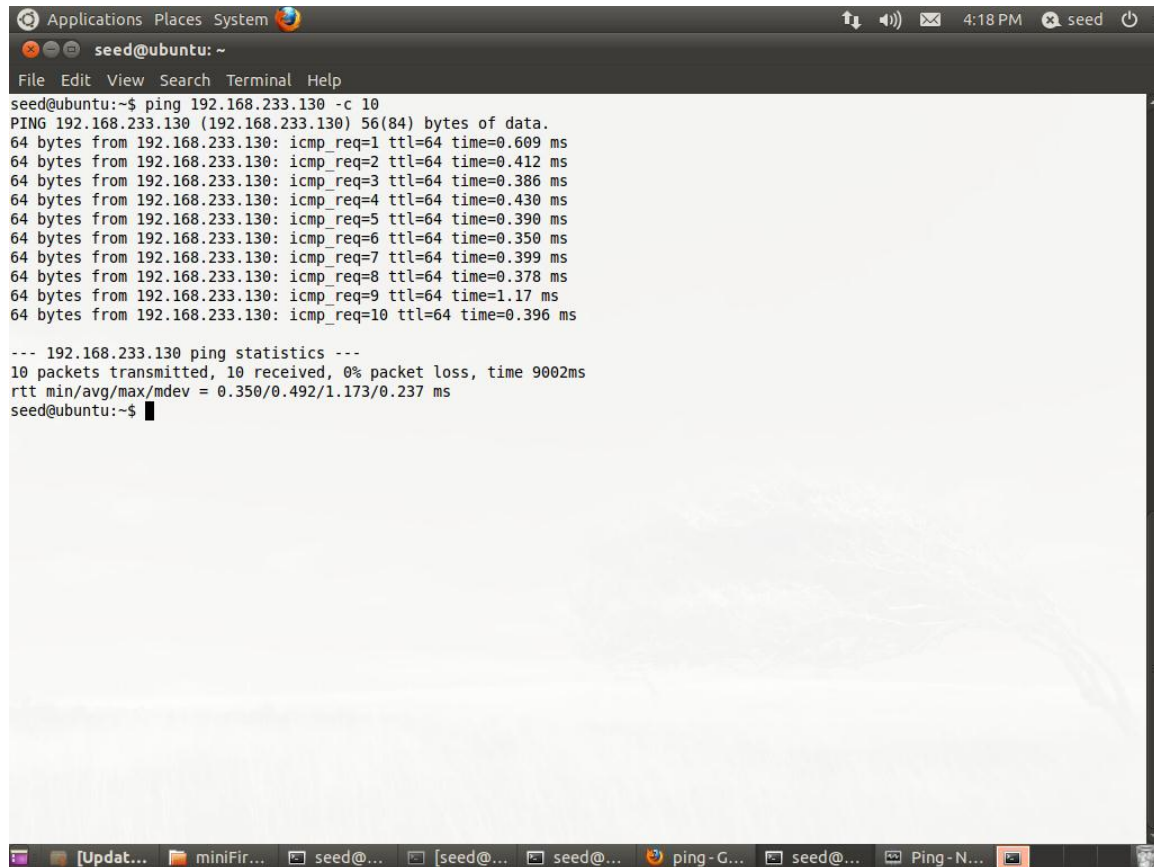
```
0000 00 0c 29 c7 c3 bb 00 50 56 e5 e0 9a 08 00 45 00  ..)....P V....E.
0010 00 b0 20 c7 00 00 80 11 c5 a1 c0 a8 e9 02 c0 a8  .. .....
0020 e9 80 00 35 be 1d 00 9c 1e 50 d9 ab 81 80 00 01  ...5.... .P.....
0030 00 07 00 00 00 00 03 77 77 77 06 67 6f 6f 67 6c  ....w ww.googl
```

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.

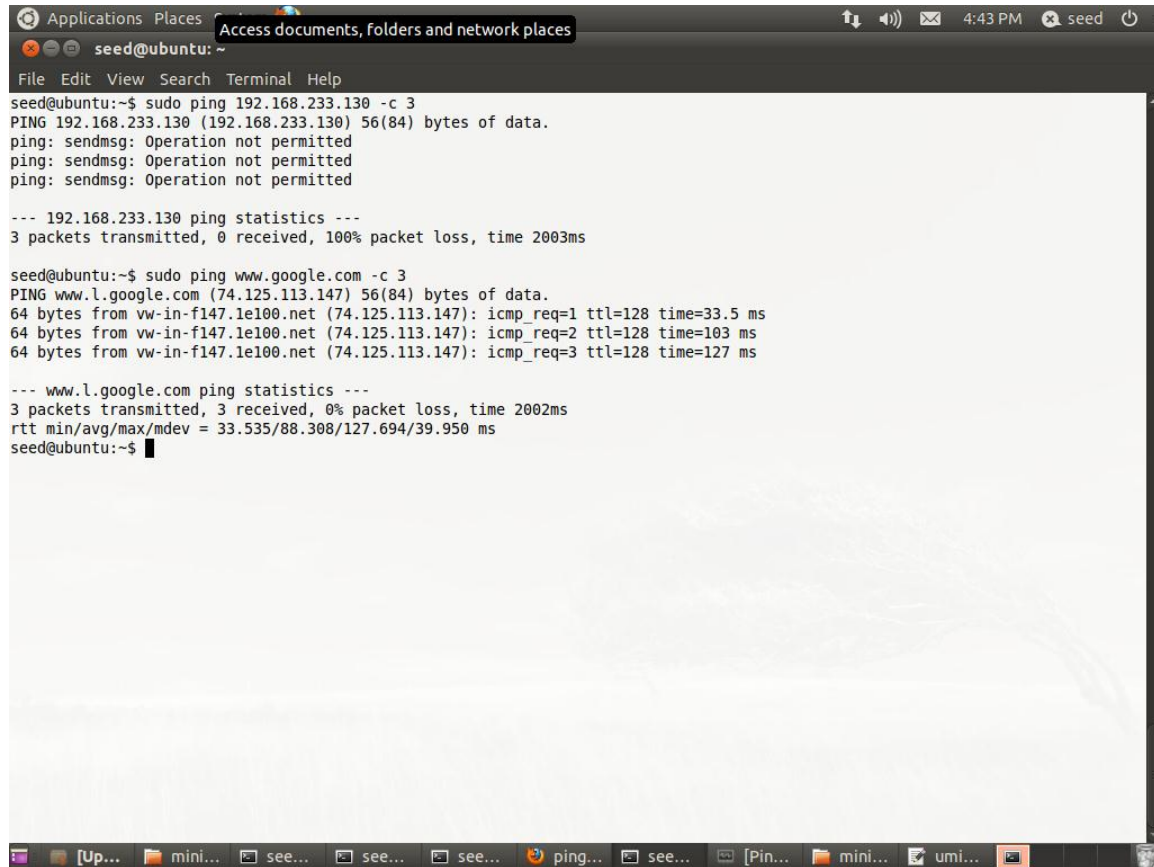


```
Applications Places System
seed@ubuntu: ~
File Edit View Search Terminal Help
seed@ubuntu:~$ ping 192.168.233.130 -c 10
PING 192.168.233.130 (192.168.233.130) 56(84) bytes of data.
64 bytes from 192.168.233.130: icmp_req=1 ttl=64 time=0.609 ms
64 bytes from 192.168.233.130: icmp_req=2 ttl=64 time=0.412 ms
64 bytes from 192.168.233.130: icmp_req=3 ttl=64 time=0.386 ms
64 bytes from 192.168.233.130: icmp_req=4 ttl=64 time=0.430 ms
64 bytes from 192.168.233.130: icmp_req=5 ttl=64 time=0.390 ms
64 bytes from 192.168.233.130: icmp_req=6 ttl=64 time=0.350 ms
64 bytes from 192.168.233.130: icmp_req=7 ttl=64 time=0.399 ms
64 bytes from 192.168.233.130: icmp_req=8 ttl=64 time=0.378 ms
64 bytes from 192.168.233.130: icmp_req=9 ttl=64 time=1.17 ms
64 bytes from 192.168.233.130: icmp_req=10 ttl=64 time=0.396 ms

--- 192.168.233.130 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9002ms
rtt min/avg/max/mdev = 0.350/0.492/1.173/0.237 ms
seed@ubuntu:~$
```

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.




```
Applications Places Access documents, folders and network places
seed@ubuntu: ~
File Edit View Search Terminal Help
seed@ubuntu:~$ sudo ping 192.168.233.130 -c 3
PING 192.168.233.130 (192.168.233.130) 56(84) bytes of data.
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted

--- 192.168.233.130 ping statistics ---
3 packets transmitted, 0 received, 100% packet loss, time 2003ms

seed@ubuntu:~$ sudo ping www.google.com -c 3
PING www.l.google.com (74.125.113.147) 56(84) bytes of data.
64 bytes from vw-in-f147.1e100.net (74.125.113.147): icmp_req=1 ttl=128 time=33.5 ms
64 bytes from vw-in-f147.1e100.net (74.125.113.147): icmp_req=2 ttl=128 time=103 ms
64 bytes from vw-in-f147.1e100.net (74.125.113.147): icmp_req=3 ttl=128 time=127 ms

--- www.l.google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 33.535/88.308/127.694/39.950 ms
seed@ubuntu:~$
```

Applications Places System  4:43 PM seed

Capturing from eth0 - Wireshark

File Edit View Go Capture Analyze Statistics Telephony Tools Help

Filter: Expression... Clear Apply

No.	Time	Source	Destination	Protocol	Info
1	0.000000	192.168.233.1	192.168.233.255	DB-LSP-D	Dropbox LAN sync Discovery Protocol
2	12.402030	192.168.233.1	192.168.233.255	BROWSER	Local Master Announcement QINYUN-PC, Workstation, Serve
3	15.508815	192.168.233.128	192.168.233.2	DNS	Standard query A www.google.com
4	15.513367	Vmware_e5:e0:9a	Broadcast	ARP	Who has 192.168.233.128? Tell 192.168.233.2
5	15.515335	Vmware_c7:c3:bb	Vmware_e5:e0:9a	ARP	192.168.233.128 is at 00:0c:29:c7:c3:bb
6	15.515351	192.168.233.2	192.168.233.128	DNS	Standard query response CNAME www.l.google.com A 74.125
7	15.560258	192.168.233.128	74.125.113.147	ICMP	Echo (ping) request (id=0x0e16, seq(be/le)=1/256, ttl=
8	15.593131	74.125.113.147	192.168.233.128	ICMP	Echo (ping) reply (id=0x0e16, seq(be/le)=1/256, ttl=
9	15.596024	192.168.233.128	192.168.233.2	DNS	Standard query PTR 147.113.125.74.in-addr.arpa
10	15.598970	192.168.233.2	192.168.233.128	DNS	Standard query response PTR vw-in-f147.1e100.net
11	16.562615	192.168.233.128	74.125.113.147	ICMP	Echo (ping) request (id=0x0e16, seq(be/le)=2/512, ttl=
12	16.665787	74.125.113.147	192.168.233.128	ICMP	Echo (ping) reply (id=0x0e16, seq(be/le)=2/512, ttl=
13	16.666587	192.168.233.128	192.168.233.2	DNS	Standard query PTR 147.113.125.74.in-addr.arpa
14	16.670041	192.168.233.2	192.168.233.128	DNS	Standard query response PTR vw-in-f147.1e100.net
15	17.562649	192.168.233.128	74.125.113.147	ICMP	Echo (ping) request (id=0x0e16, seq(be/le)=3/768, ttl=
16	17.689822	74.125.113.147	192.168.233.128	ICMP	Echo (ping) reply (id=0x0e16, seq(be/le)=3/768, ttl=
17	17.690773	192.168.233.128	192.168.233.2	DNS	Standard query PTR 147.113.125.74.in-addr.arpa

▶ Frame 1: 173 bytes on wire (1384 bits), 173 bytes captured (1384 bits)
 ▶ Ethernet II, Src: Vmware_c0:00:08 (00:50:56:c0:00:08), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
 ▶ Internet Protocol, Src: 192.168.233.1 (192.168.233.1), Dst: 192.168.233.255 (192.168.233.255)
 ▶ User Datagram Protocol, Src Port: db-lsp-disc (17500), Dst Port: db-lsp-disc (17500)
 ▶ Dropbox LAN sync Discovery Protocol

```

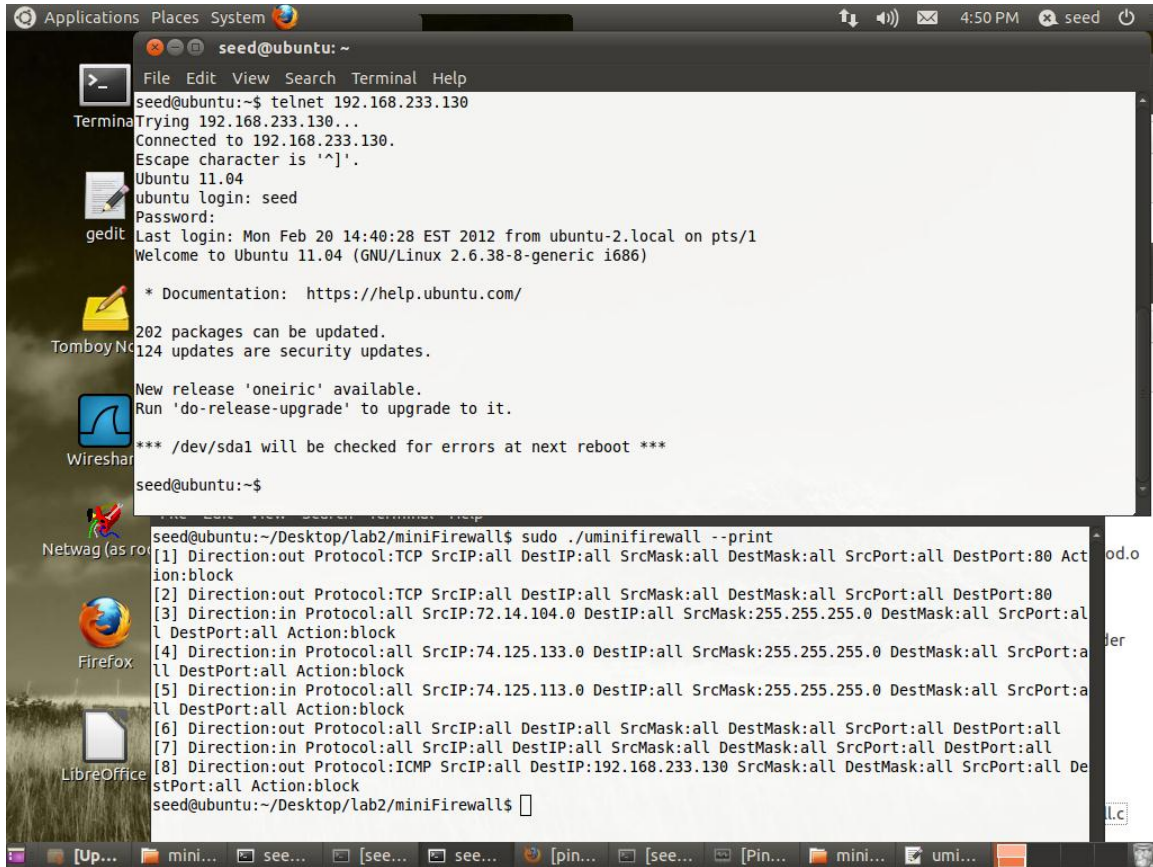
0000  ff ff ff ff ff ff 00 50 56 c0 00 08 08 00 45 00  .....P V....E.
0010  00 9f 09 04 00 00 80 11 dc f7 c0 a8 e9 01 c0 a8  .....
0020  e9 ff 44 5c 44 5c 00 8b 9c 6c 7b 22 68 6f 73 74  ..D\D\.. \{"host
0030  5f 69 6e 74 22 3a 20 31 34 39 30 34 35 33 36 37  _int": 1 49045367
  
```

eth0: <live capture in progress> Fil... Packets: 23 Displayed: 23 Marked: 0 Profile: Default

Capturing from eth0 - ... [Update Manager] seed@ubuntu: ~

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).



```
seed@ubuntu: ~  
File Edit View Search Terminal Help  
seed@ubuntu:~$ telnet 192.168.233.130  
Trying 192.168.233.130...  
Connected to 192.168.233.130.  
Escape character is '^'.  
Ubuntu 11.04  
ubuntu login: seed  
Password:  
Last login: Mon Feb 20 14:40:28 EST 2012 from ubuntu-2.local on pts/1  
Welcome to Ubuntu 11.04 (GNU/Linux 2.6.38-8-generic i686)  
  
* Documentation: https://help.ubuntu.com/  
  
202 packages can be updated.  
124 updates are security updates.  
  
New release 'oneiric' available.  
Run 'do-release-upgrade' to upgrade to it.  
  
*** /dev/sda1 will be checked for errors at next reboot ***  
seed@ubuntu:~$  
  
seed@ubuntu:~/Desktop/lab2/miniFirewall$ sudo ./uminiFirewall --print  
[1] Direction:out Protocol:TCP SrcIP:all DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:80 Action:block  
[2] Direction:out Protocol:TCP SrcIP:all DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:80 Action:block  
[3] Direction:in Protocol:all SrcIP:72.14.104.0 DestIP:all SrcMask:255.255.255.0 DestMask:all SrcPort:all DestPort:all Action:block  
[4] Direction:in Protocol:all SrcIP:74.125.133.0 DestIP:all SrcMask:255.255.255.0 DestMask:all SrcPort:all DestPort:all Action:block  
[5] Direction:in Protocol:all SrcIP:74.125.113.0 DestIP:all SrcMask:255.255.255.0 DestMask:all SrcPort:all DestPort:all Action:block  
[6] Direction:out Protocol:all SrcIP:all DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:all Action:block  
[7] Direction:in Protocol:all SrcIP:all DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:all Action:block  
[8] Direction:out Protocol:ICMP SrcIP:all DestIP:192.168.233.130 SrcMask:all DestMask:all SrcPort:all DestPort:all Action:block  
seed@ubuntu:~/Desktop/lab2/miniFirewall$
```

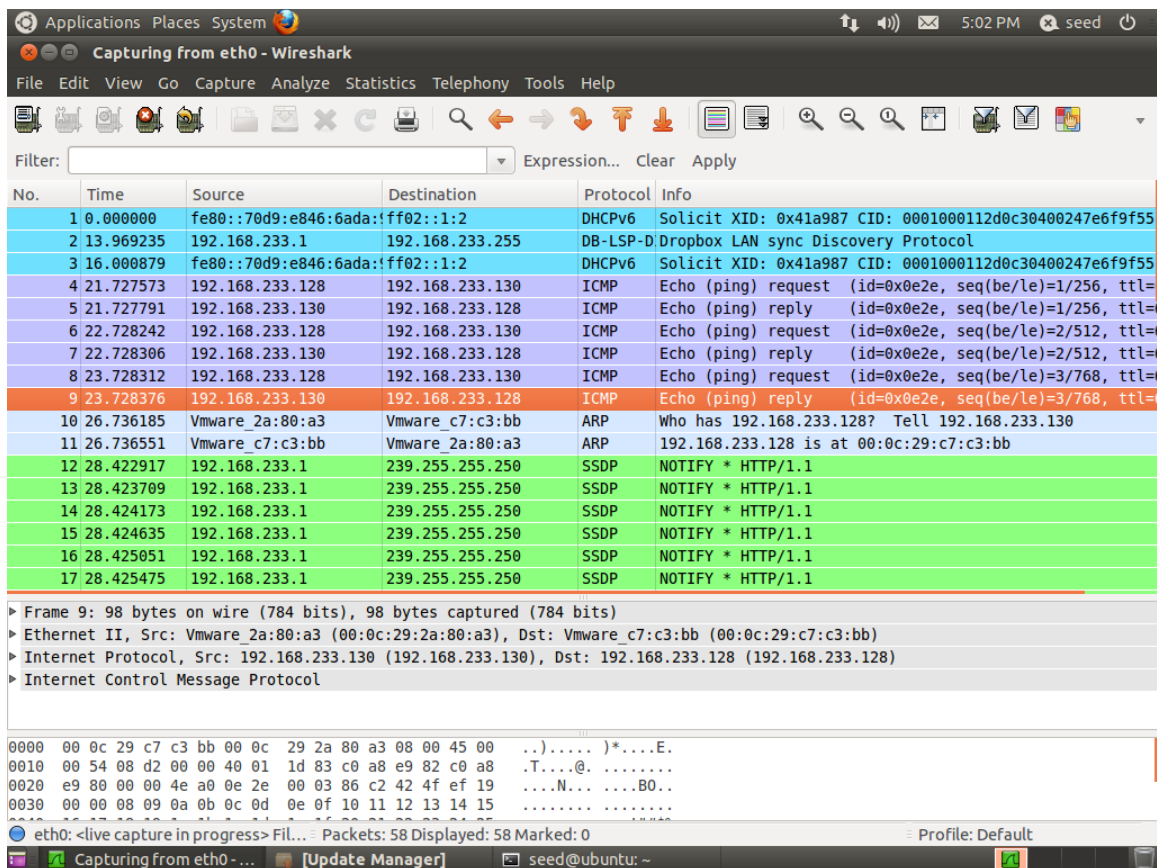
Then I unblocked all packets again and set the rule “`—in —proto ICMP —srcip 192.168.233.130 — action block`”. And the results of ping experiment are:

```
Applications Places System
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

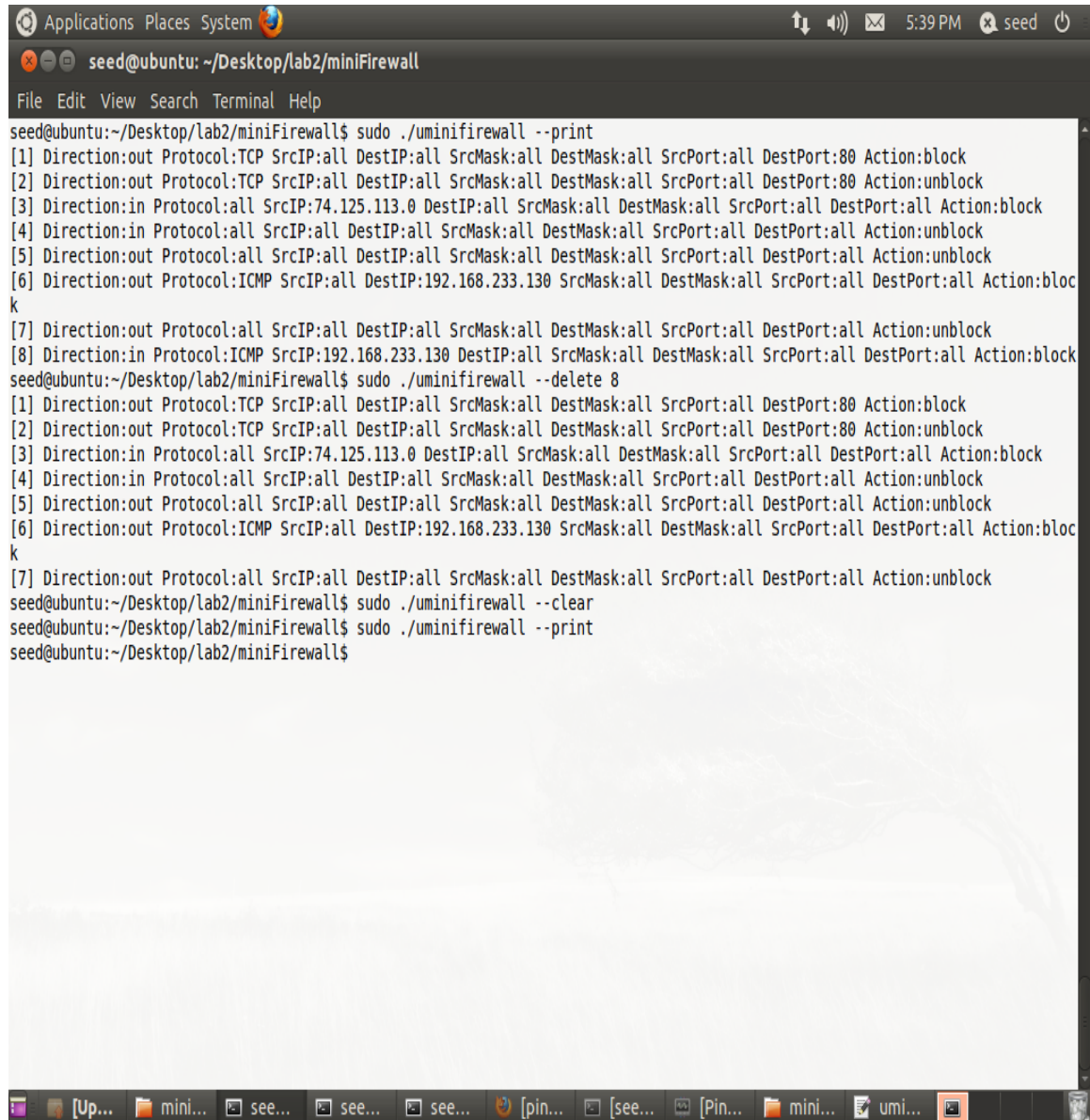
--- www.l.google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 28.544/28.986/29.436/0.364 ms
seed@ubuntu:~$
```



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.

A screenshot of a Linux terminal window titled 'seed@ubuntu: ~/Desktop/lab2/miniFirewall'. The window shows the execution of the 'miniFirewall' application. The user runs 'sudo ./uminifirewall --print', which lists 8 rules. Then, they run 'sudo ./uminifirewall --delete 8', which removes the 8th rule. Finally, they run 'sudo ./uminifirewall --clear', which removes all rules, and then 'sudo ./uminifirewall --print' again to confirm the list is empty. The terminal output is as follows:

```
seed@ubuntu:~/Desktop/lab2/miniFirewall$ sudo ./uminifirewall --print
[1] Direction:out Protocol:TCP SrcIP:all DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:80 Action:block
[2] Direction:out Protocol:TCP SrcIP:all DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:80 Action:unblock
[3] Direction:in Protocol:all SrcIP:74.125.113.0 DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:all Action:block
[4] Direction:in Protocol:all SrcIP:all DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:all Action:unblock
[5] Direction:out Protocol:all SrcIP:all DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:all Action:unblock
[6] Direction:out Protocol:ICMP SrcIP:all DestIP:192.168.233.130 SrcMask:all DestMask:all SrcPort:all DestPort:all Action:block
[7] Direction:out Protocol:all SrcIP:all DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:all Action:unblock
[8] Direction:in Protocol:ICMP SrcIP:192.168.233.130 DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:all Action:block
seed@ubuntu:~/Desktop/lab2/miniFirewall$ sudo ./uminifirewall --delete 8
[1] Direction:out Protocol:TCP SrcIP:all DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:80 Action:block
[2] Direction:out Protocol:TCP SrcIP:all DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:80 Action:unblock
[3] Direction:in Protocol:all SrcIP:74.125.113.0 DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:all Action:block
[4] Direction:in Protocol:all SrcIP:all DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:all Action:unblock
[5] Direction:out Protocol:all SrcIP:all DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:all Action:unblock
[6] Direction:out Protocol:ICMP SrcIP:all DestIP:192.168.233.130 SrcMask:all DestMask:all SrcPort:all DestPort:all Action:block
[7] Direction:out Protocol:all SrcIP:all DestIP:all SrcMask:all DestMask:all SrcPort:all DestPort:all Action:unblock
seed@ubuntu:~/Desktop/lab2/miniFirewall$ sudo ./uminifirewall --clear
seed@ubuntu:~/Desktop/lab2/miniFirewall$ sudo ./uminifirewall --print
seed@ubuntu:~/Desktop/lab2/miniFirewall$
```

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.