TCP Spoofing

Ethical Hacking Challenge #2

Francesco Marchiori

ID = 2020389

A.Y. 2021/2022

Contents

List of Figures

1 Introduction

In this laboratory session we will first implement a simple stateless packet-filtering firewall, which inspects packets, and decides whether to drop or forward a packet based on firewall rules. Then, we will use `iptables` to set up a firewall in order to also be able to make changes to packets

2 Environment Setup

The environment, as suggested in the instructions, consists in a virtual machine running `SEED-Ubuntu 20.04` on which are present some Docker containers, built and run using the `docker-compose.yml` file provided in the material.

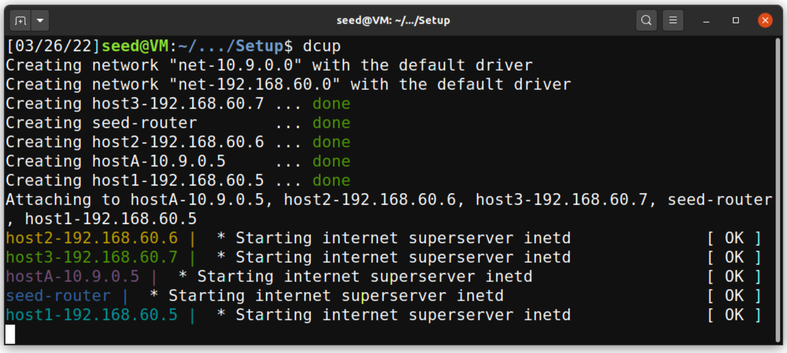

Figure 1: Setup of the Environment.

3 Task 1: Implement a simple Firewall

3.1 Task 1.A: Implement a Simple Kernel Module

LKM allows us to add a new module to the kernel at the runtime. This new module enables us to extend the functionalities of the kernel, without rebuilding the kernel or even rebooting the computer. The packet filtering part of a firewall can be implemented as an LKM. First, we need to `make` the `Makefile` in order to compile the `hello.c` program into a **Loadable Kernel Module**. After that, we load the module, list all the modules and remove it. In this way we should be able to see the printk command inside the program. Let's execute the command and see what happens.

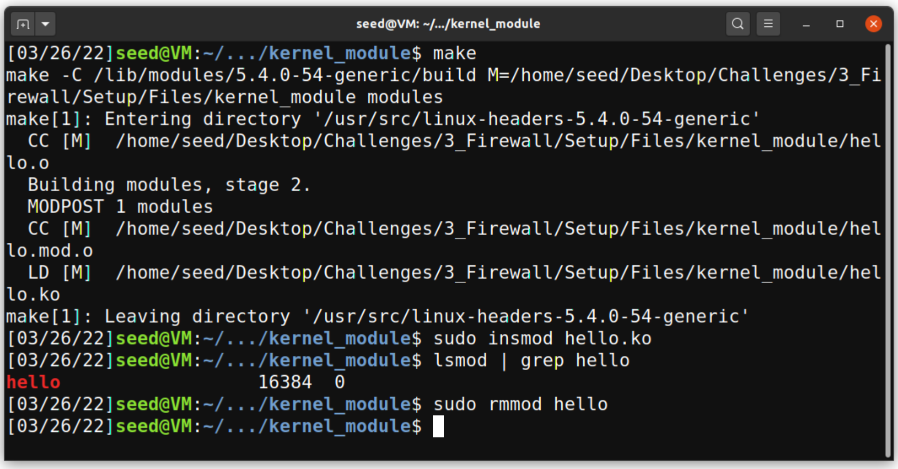

Figure 2: Loading, listing and removing the LKM.

So far so good. Let's now see if the information is printed in the `/var/log/syslog` file using the `dmesg` command.

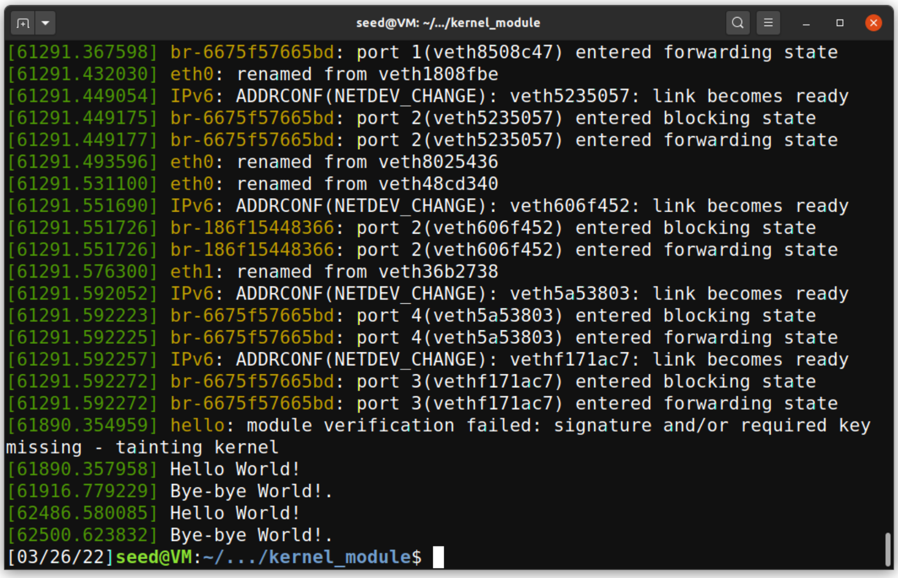

Figure 3: Messages print by `hello.c`.

As we can see, messages are printed correctly, indicating the success of this task. As a note, they are printed twice because the program has been run for the first time in an attempt and run another time in order to take screenshots of it.

3.2 Task 1.B: Implement a Simple Firewall Using Netfilter

In this task, we will write our packet filtering program as an LKM, and then insert in into the packet processing path inside the kernel. This cannot be easily done in the past before the `netfilter` was introduced into the Linux. First of all, we compile the sample code using the provided `Makefile` using the same commands as before (omitting the last `sudo rmmod seedFilter` otherwise we wouldn't have any difference). We can see that the code has been compiled correctly in the following Figure.

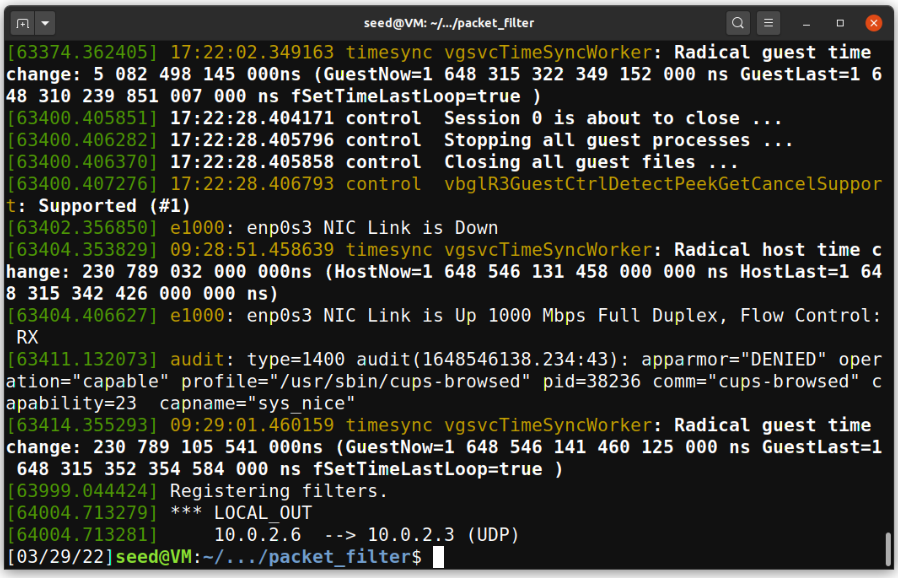

Figure 4: Compiling `seedFilter.c`.

Now that we have compiled everything correctly, let's try to perform a UDP connection to 8.8.8.8, which in this case should be blocked.

Figure 5: Connection timed out when `netfilter` is active.

As we can see in Fig. (5), the connection has been timed out and no servers could be reached because out program was active.
The five hooks defined by `netfilter` are called in the following conditions:

1. `NF_INET_PRE_ROUTING`: before routing decisions.
2. `NF_INET_LOCAL_IN`: after routing decisions if the packet destination address is the same as the host.
3. `NF_INET_FORWARD`: if the packet is addressed to another interface.
4. `NF_INET_LOCAL_OUT`: for packets that have as source address the host address.
5. `NF_INET_POST_ROUTING`: after the routing engine has determined that a packet is destined for another host.

Let's now implement two new hooks. First of all, we will implement a hook that will prevent other computers to ping the VM, which means that we will drop every incoming ICMP packet but allow any outgoing one. To achieve this, we write a new hook function called `blockICMP` and we implement it in the following way inside the same `seedFilter.c` file:

```c
unsigned int blockICMP(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)
{
   struct iphdr *iph;

   char ip[16] = "10.9.0.1";
   u32  ip_addr;

   if (!skb) return NF_ACCEPT;

   iph = ip_hdr(skb);
   // Convert the IPv4 address from dotted decimal to 32-bit binary
   in4_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);

   if (iph->protocol == IPPROTO_ICMP) {
       if (iph->daddr == ip_addr){
            printk(KERN_WARNING "*** Dropping %pI4 (ICMP)", &(iph->daddr));
            return NF_DROP;
        }
   }
   return NF_ACCEPT;
   // ...
   int registerFilter(void) {
       printk(KERN_INFO "Registering filters.\n");

       hook1.hook = blockICMP;
       hook1.hooknum = NF_INET_LOCAL_IN;
       hook1.pf = PF_INET;
       hook1.priority = NF_IP_PRI_FIRST;
       nf_register_net_hook(&init_net, &hook1);

       return 0;
    }
}
```

After repeating the same process as before to enable the firewall, we try to ping the VM machine from another container in the docker and see if we are able to ping it, and then see if we are able to ping anything else. Results are shown in Fig. (6).

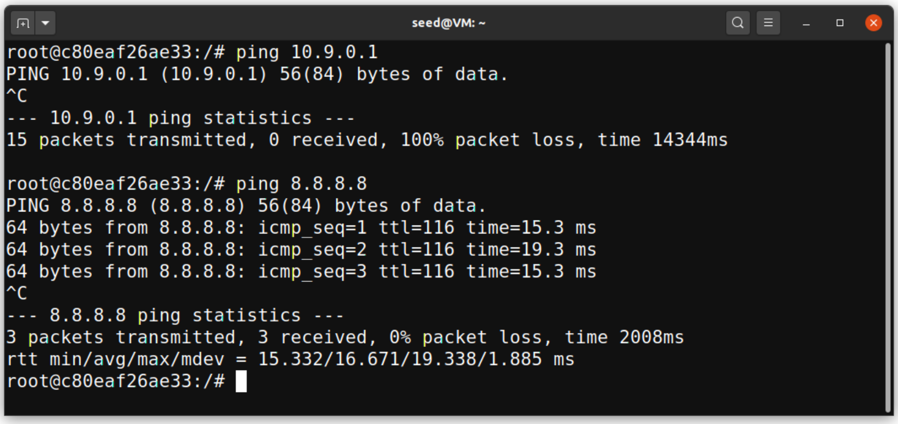

Figure 6: Successfully preventing other computers to ping the VM.

To prevent other computers to telnet into the VM instead, we should block incoming TCP packets on port 23, and this is done analogously as before with the following code:

```
unsigned int blockTelnet(void *priv, struct sk_buff *skb, const struct nf_hook_state *state)
{
    struct iphdr *iph;
    struct tcphdr *tcph;

    u16  port   = 23;
    char ip[16] = "10.9.0.1";
    u32  ip_addr;

    if (!skb) return NF_ACCEPT;

    iph = ip_hdr(skb);
    // Convert the IPv4 address from dotted decimal to 32-bit binary
    in4_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);

    if (iph->protocol == IPPROTO_TCP) {
        tcph = tcp_hdr(skb);
         if (iph->daddr == ip_addr && ntohs(tcph->dest) == port){
             printk(KERN_WARNING "*** Dropping %pI4 (TCP), port %d\n", &(iph->daddr), port);
             return NF_DROP;
         }
    }
    return NF_ACCEPT;
}
    // ...
    int registerFilter(void) {
        printk(KERN_INFO "Registering filters.\n");

        hook1.hook = blockTelnet;
        hook1.hooknum = NF_INET_LOCAL_IN;
        hook1.pf = PF_INET;
        hook1.priority = NF_IP_PRI_FIRST;
        nf_register_net_hook(&init_net, &hook1);

        return 0;
     }
}
```

Now let's try to first ping the machine and then try to enable a telnet connection. Of course, the previous filter has been removed, otherwise we couldn't be able to even ping the machine since any ICMP packet would have been blocked.

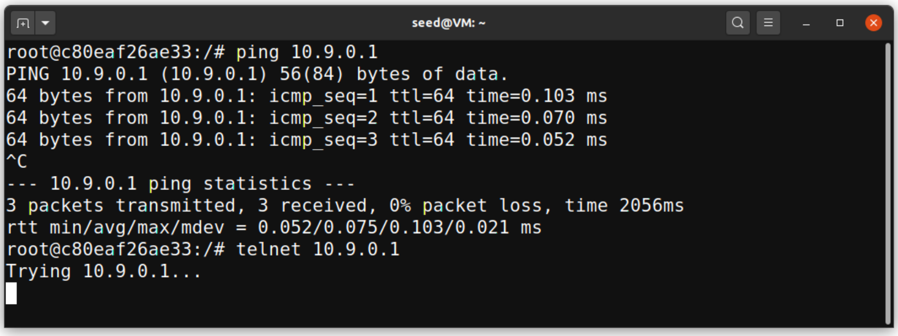

Figure 7: Successfully preventing other computers to telnet the VM.

4 Task 2: Experimenting with Stateless Firewall Rules

In the previous task, we had a chance to build a simple firewall using `netfilter`. Actually, Linux already has a built-in firewall, also based on `netfilter`. This firewall is called `iptables`. Technically, the kernel part implementation of the firewall is called `Xtables`, while `iptables` is a user-space program to configure the firewall. However, `iptables` is often used to refer to both the kernel-part implementation and the user-space program.

4.1 Task 2.A: Protecting the Router

In this task, we will set up rules to prevent outside machines from accessing the router machine, except ping. In particular, we will execute these four commands on the router container:

1. `iptables -A INPUT -p icmp -icmp-type echo-request -j ACCEPT`: accept ICMP requests in input.
2. `iptables -A OUTPUT -p icmp -icmp-type echo-reply -j ACCEPT`: accept ICMP replies in output.
3. `iptables -P OUTPUT DROP`: drop every outgoing packet.
4. `iptables -P INPUT DROP`: drop every ingoing packet.

This series of commands works for our purpose because iptables entries are applied in order. Therefore if an incoming packet is an ICMP request it will be accepted because the rule for accepting ICMP packets comes before the rule for dropping every incoming packets. We can see that it works in the following Figure.

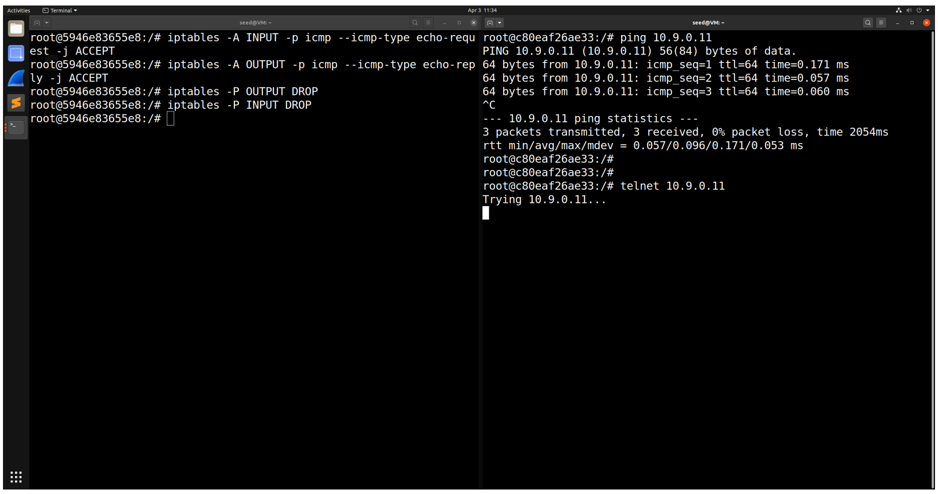

Figure 8: Preventing router connection except for ping.

4.2 Task 2.B: Protecting the Internal Network

In this task, we will set up firewall rules on the router to protect the internal network 192.168.60.0/24. We need to use the `FORWARD` chain for this purpose. More specifically, we need to enforce the following restrictions on the ICMP traffic:

1. Outside hosts cannot ping internal hosts.
2. Outside hosts can ping the router.
3. Internal hosts can ping outside hosts.
4. All other packets between the internal and external networks should be blocked.

The commands that we are going to use here are the following:

1. `iptables -A FORWARD -i eth0 -p icmp -icmp-type echo-request -j DROP`: with this command we are dropping every ICMP packet coming from the `eth0` interface of the router. In this way we are only blocking the incoming packets and not the outgoing packets, but the router is still pingable from every direction.
2. `iptables -A FORWARD ! -p icmp -j DROP`: with this command we are dropping every packet that is not an ICMP.

In this way outside hosts cannot ping internal hosts but can ping the router, internal hosts can ping outside hosts and all other packets are blocked. This can be proven as in Fig. (9), where we are first trying to ping the router from the external network and then we try to ping a machine in the internal network.

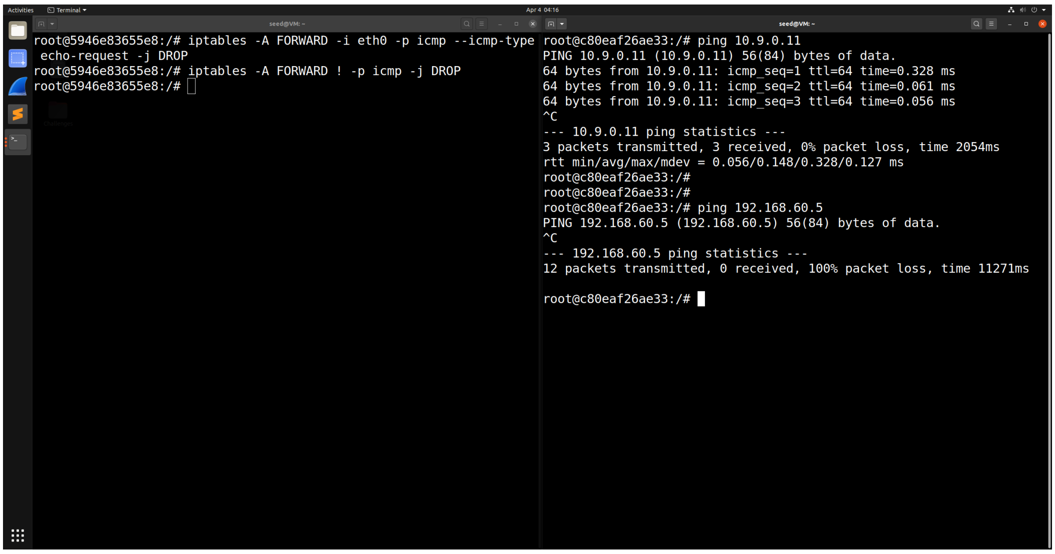

Figure 9: Successfully protecting the internal network.

4.3 Task 2.C: Protecting Internal Server

In this task, we want to protect the TCP servers inside the internal network (192.168.60.0/24). More specifically, we would like to achieve the following objectives:

1. All the internal hosts run a telnet server (listening to port 23). Outside hosts can only access the `telnet` server on 192.168.60.5, not the other internal hosts.
2. Outside hosts cannot access other internal servers.
3. Internal hosts can access all the internal servers.
4. Internal hosts cannot access external servers.
5. In this task, the connection tracking mechanism is not allowed.

The commands that we are going to use here are the following:

1. `iptables -A FORWARD -i eth0 -p tcp -d 192.168.60.5 -dport 23 -j ACCEPT`: with this command we are satisfying requirement 1, so now external machines can connect only to 192.168.60.5 through `telnet`.

2. `iptables -A FORWARD ! -i eth1 -o eth1 -p tcp -j DROP`: with this command we are dropping every packet that is not a `telnet` request from an internal machine to another internal machine.

We can see that this approach works in Fig. (10). On the left we have the router shell, on the right we have a shell of an external machine and on the bottom we have a shell of an internal machine. As we can see, an external machine cannot telnet to any internal machine with the only exception of 192.168.60.5, while internal machines can connect to whatever internal machine they would like.

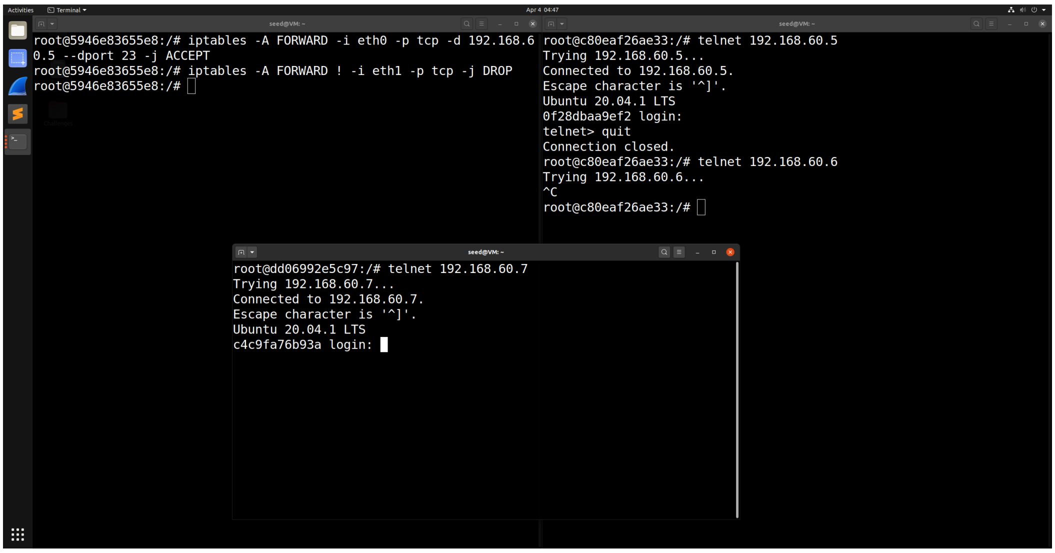

Figure 10: Successfully protecting the internal server.