Linux Firewall Exploration Lab

1 Overview

The learning objective of this lab is for students to gain the insights on how firewalls work by playing with firewall software and implement a simplified packet filtering firewall. Firewalls have several types; in this lab, we focus on *packet filter*. Packet filters inspect packets, and decide whether to drop or forward a packet based on firewall rules. Packet filters are usually *stateless*; they filter each packet based only on the information contained in that packet, without paying attention to whether a packet is part of an existing stream of traffic. Packet filters often use a combination of a packet's source and destination address, its protocol, and, for TCP and UDP traffic, port numbers. In this lab, students will play with this type of firewall, and also through the implementation of some of the key functionalities, they can understand how firewalls work. Moreover, students will learn how to use SSH tunnels to bypass firewalls. This lab covers the following topics:

- Firewall
- Netfilter
- Loadable kernel module
- SSH tunnel

Readings and related topics. Detailed coverage of Firewalls can be found in Chapter 14 of the SEED book, *Computer Security: A Hands-on Approach*, by Wenliang Du. A related lab is the Firewall Bypassing lab, which shows how to use VPN to bypass firewalls.

Lab environment. This lab has been tested on our pre-built Ubuntu 16.04 VM, which can be downloaded from the SEED website.

2 Lab Tasks

2.1 Task 1: Using Firewall

`Linux` has a tool called `iptables`, which is essentially a firewall. It has a nice front end program called `ufw`. In this task, the objective is to use `ufw` to set up some firewall policies, and observe the behaviors of your system after the policies become effective. You need to set up at least two VMs, one called Machine A, and other called Machine B. You run the firewall on your Machine A. Basically, we use `ufw` as a personal firewall. Optionally, if you have more VMs, you can set up a firewall at your router, so it can protect a network, instead of just one single computer. After you set up the two VMs, you should perform the following tasks:

- Prevent A from doing `telnet` to Machine B.
- Prevent B from doing `telnet` to Machine A.
- Prevent A from visiting an external web site. You can choose any web site that you like to block, but keep in mind, some web servers have multiple IP addresses.

You can find the manual of `ufw` by typing "`man ufw`" or search it online. It is pretty straightforward to use. Please remember that the firewall is not enabled by default, so you should run a command to specifically enable it. We list some commonly used commands in Appendix A.

Before starting the task, go to the default policy file `/etc/default/ufw`. find the following entry, and change the rule from `DROP` to `ACCEPT`; otherwise, all the incoming traffic will be dropped by default.

```
# Set the default input policy to ACCEPT, DROP, or REJECT. Please note that if
# you change this you will most likely want to adjust your rules.
DEFAULT_INPUT_POLICY="DROP"
```

2.2 Task 2: Implementing a Simple Firewall

The firewall you used in the previous task is a packet filtering type of firewall. The main part of this type of firewall is the filtering part, which inspects each incoming and outgoing packets, and enforces the firewall policies set by the administrator. Since the packet processing is done within the kernel, the filtering must also be done within the kernel. Therefore, it seems that implementing such a firewall requires us to modify the `Linux` kernel. In the past, this had to be done by modifying and rebuilding the kernel. The modern `Linux` operating systems provide several new mechanisms to facilitate the manipulation of packets without rebuilding the kernel image. These two mechanisms are *Loadable Kernel Module* (`LKM`) and `Netfilter`.

`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. However, this is not enough. In order for the filtering module to block incoming/outgoing packets, the module must be inserted into the packet processing path. This cannot be easily done in the past before the `Netfilter` was introduced into the `Linux`.

`Netfilter` is designed to facilitate the manipulation of packets by authorized users. `Netfilter` achieves this goal by implementing a number of *hooks* in the `Linux` kernel. These hooks are inserted into various places, including the packet incoming and outgoing paths. If we want to manipulate the incoming packets, we simply need to connect our own programs (within LKM) to the corresponding hooks. Once an incoming packet arrives, our program will be invoked. Our program can decide whether this packet should be blocked or not; moreover, we can also modify the packets in the program.

In this task, you need to use LKM and `Netfilter` to implement the packet filtering module. This module will fetch the firewall policies from a data structure, and use the policies to decide whether packets should be blocked or not. To make your life easier, so you can focus on the filtering part, the core of firewalls, we allow you to hardcode your firewall policies in the program. You should support at least five different rules, including the ones specified in the previous task. Guidelines on how to use `Netfilter` can be found in Section 3. In addition, Chapter 14 (§14.4) of the SEED book provides more detailed explanation on `Netfilter`.

Note for Ubuntu 16.04 VM: The code in the SEED book was developed in Ubuntu 12.04. It needs to be changed slightly to work in Ubuntu 16.04. The change is in the definition of the callback function `telnetFilter()`, because the prototype of `Netfilter`'s callback function has been changed in Ubuntu 16.04. See the difference in the following:

```
// In Ubuntu 12.04
unsigned int telnetFilter(unsigned int hooknum, struct sk_buff *skb,
           const struct net_device *in, const struct net_device *out,
           int (*okfn)(struct sk_buff *))

// In Ubuntu 16.04
unsigned int telnetFilter(void *priv, struct sk_buff *skb,
                          const struct nf_hook_state *state)
```

2.3 Task 3: Evading Egress Filtering

Many companies and schools enforce egress filtering, which blocks users inside of their networks from reaching out to certain web sites or Internet services. They do allow users to access other web sites. In many cases, this type of firewalls inspect the destination IP address and port number in the outgoing packets. If a packet matches the restrictions, it will be dropped. They usually do not conduct deep packet inspections (i.e., looking into the data part of packets) due to the performance reason. In this task, we show how such egress filtering can be bypassed using the tunnel mechanism. There are many ways to establish tunnels; in this task, we only focus on SSH tunnels.

You need two VMs A and B for this task (three will be better). Machine A is running behind a firewall (i.e., inside the company or school's network), and Machine B is outside of the firewall. Typically, there is a dedicated machine that runs the firewall, but in this task, for the sake of convenience, you can run the firewall on Machine A. You can use the firewall program that you implemented in the previous task, or directly use `ufw`. You need to set up the following two firewall rules:

- Block all the outgoing traffic to external telnet servers. In reality, the servers that are blocked are usually game servers or other types of servers that may affect the productivity of employees. In this task, we use the telnet server for demonstration purposes. You can run the telnet server on Machine B. If you have a third VM, Machine C, you can run the telnet server on Machine C.
- Block all the outgoing traffic to `www.facebook.com`, so employees (or school kids) are not distracted during their work/school hours. Social network sites are commonly blocked by companies and schools. After you set up the firewall, launch your Firefox browser, and try to connect to Facebook, and report what happens. If you have already visited Facebook before using this browser, you need to clear all the caches using Firefox's menu: `Edit -> Preferences -> Privacy & Security (left pane) -> Clear History (Button on right)`; otherwise, the cached pages may be displayed. If everything is set up properly, you should not be able to see Facebook pages. It should be noted that Facebook has many IP addresses, it can change over the time. Remember to check whether the address is still the same by using `ping` or `dig` command. If the address has changed, you need to update your firewall rules. You can also choose web sites with static IP addresses, instead of using Facebook. For example, most universities' web servers use static IP addresses (e.g. `www.syr.edu`); for demonstration purposes, you can try block these IPs, instead of Facebook.

In addition to set up the firewall rules, you also need the following commands to manage the firewall:

```
$ sudo ufw enable          // this will enable the firewall.
$ sudo ufw disable         // this will disable the firewall.
$ sudo ufw status numbered // this will display the firewall rules.
$ sudo ufw delete 3        // this will delete the 3rd rule.
```

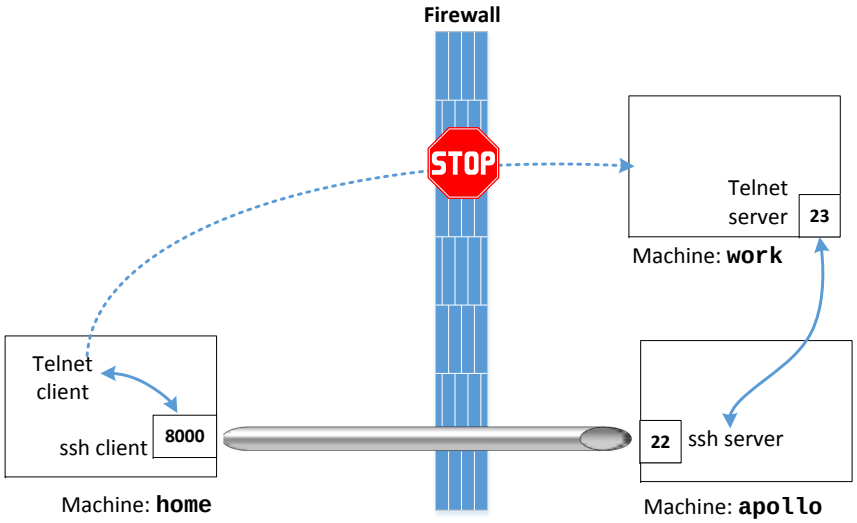

Figure 1: SSH Tunnel Example

Task 3.a: Telnet to Machine B through the firewall To bypass the firewall, we can establish an SSH tunnel between Machine A and B, so all the telnet traffic will go through this tunnel (encrypted), evading the inspection. Figure 1 illustrates how the tunnel works. The following command establishes an SSH tunnel between the localhost (port 8000) and machine B (using the default port 22); when packets come out of B's end, it will be forwarded to Machine C's port 23 (`telnet` port). If you only have two VMs, you can use one VM for both Machine B and Machine C.

```
$ ssh -L 8000:Machine_C_IP:23  seed@Machine_B_IP

// We can now telnet to Machine C via the tunnel:
$ telnet localhost 8000
```

When we `telnet` to `localhost`'s port `8000`, SSH will transfer all our TCP packets from one end of the tunnel on `localhost:8000` to the other end of the tunnel on Machine B; from there, the packets will be forwarded to `Machine_C:23`. Replies from Machine C will take a reverse path, and eventually reach our telnet client. Essentially, we are able to telnet to Machine C. Please describe your observation and explain how you are able to bypass the egress filtering. You should use Wireshark to see what exactly is happening on the wire.

Task 3.b: Connect to Facebook using SSH Tunnel. To achieve this goal, we can use the approach similar to that in Task 3.a, i.e., establishing a tunnel between your localhost:port and Machine B, and ask B to forward packets to Facebook. To do that, you can use the following command to set up the tunnel: "`ssh -L 8000:FacebookIP:80 ...`". We will not use this approach, and instead, we use a more generic approach, called dynamic port forwarding, instead of a static one like that in Task 3.a. To do that, we only specify the local port number, not the final destination. When Machine B receives a packet from the tunnel, it will dynamically decide where it should forward the packet to based on the destination information of the packet.

```
$ ssh -D 9000 -C seed@machine_B
```

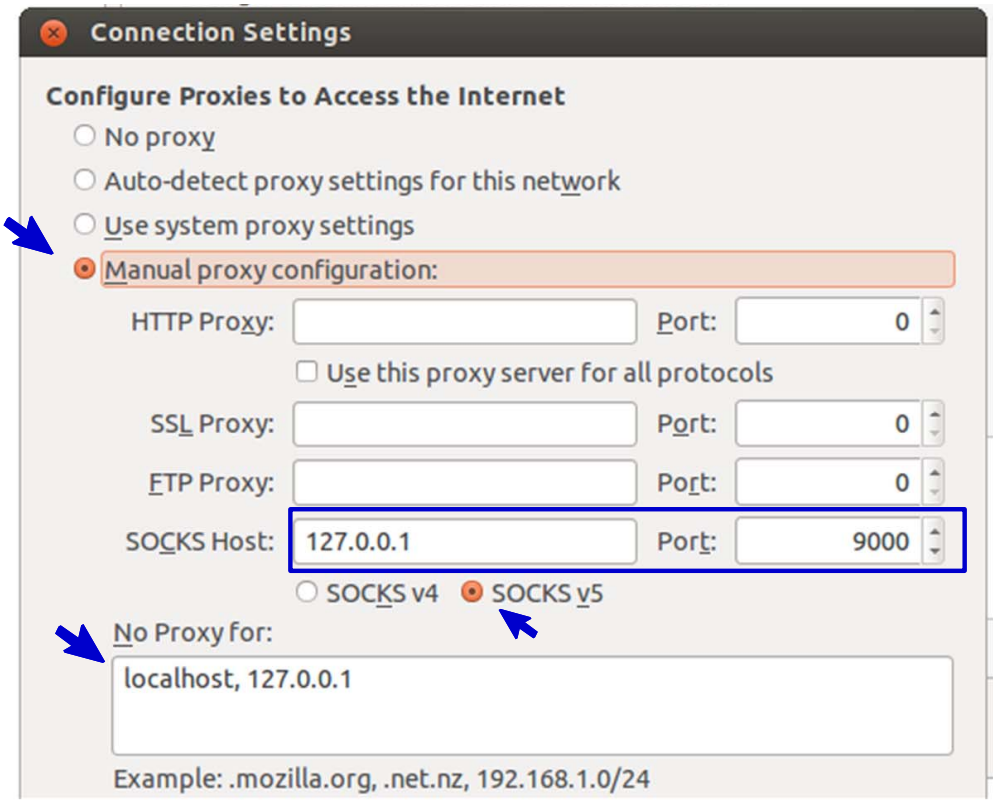

Figure 2: Configure the SOCKS Proxy

Similar to the telnet program, which connects `localhost:9000`, we need to ask Firefox to connect to `localhost:9000` every time it needs to connect to a web server, so the traffic can go through our SSH tunnel. To achieve that, we can tell Firefox to use `localhost:9000` as its proxy. To support dynamic port forwarding, we need a special type of proxy called *SOCKS proxy*, which is supported by most browsers. To set up the proxy in Firefox, go to the menu bar, click `Edit -> Preferences`, scroll down to `Network Proxy`, and click the `Settings` button. Then follow Figure 2. After the setup is done, please do the following:

1. Run Firefox and go visit the Facebook page. Can you see the Facebook page? Please describe your observation.
2. After you get the facebook page, break the SSH tunnel, clear the Firefox cache, and try the connection again. Please describe your observation.
3. Establish the SSH tunnel again and connect to Facebook. Describe your observation.
4. Please explain what you have observed, especially on why the SSH tunnel can help bypass the egress filtering. You should use Wireshark to see what exactly is happening on the wire. Please describe your observations and explain them using the packets that you have captured.

2.4 Task 4: Evading Ingress Filtering

Machine A runs a web server behind a firewall; so only the machines in the internal network can access this web server. You are working from home and needs to access this internal web server. You do not have VPN, but you have SSH, which is considered as a poor man's VPN. You do have an account on Machine A (or another internal machine behind the firewall), but the firewall also blocks incoming SSH connection, so you cannot SSH into any machine on the internal network. Otherwise, you can use the same technique from Task 3 to access the web server. The firewall, however, does not block outgoing SSH connection, i.e., if you want to connect to an outside SSH server, you can still do that.

The objective of this task is to be able to access the web server on Machine A from outside. We will use two machines to emulate the setup. Machine A is the internal computer, running the protected web server; Machine B is the outside machine at home. On Machine A, we block Machine B from accessing its port 80 (web server) and 22 (SSH server). You need to set up a reverse SSH tunnel on Machine A, so once you get home, you can still access the protected web server on A from home.

3 Guidelines

3.1 Loadable Kernel Module

The following is a simple loadable kernel module. It prints out `"Hello World!"` when the module is loaded; when the module is removed from the kernel, it prints out `"Bye-bye World!"`. The messages are not printed out on the screen; they are actually printed into the `/var/log/syslog` file. You can use `dmesg | tail -10` to read the last 10 lines of message.

```
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
    printk(KERN_INFO "Hello World!\n");
    return 0;
}

void cleanup_module(void)
{
    printk(KERN_INFO "Bye-bye World!.\n");
}
```

We now need to create `Makefile`, which includes the following contents (the above program is named `hello.c`). Then just type `make`, and the above program will be compiled into a loadable kernel module.

```
obj-m += hello.o

all:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
        make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Once the module is built by typing `make`, you can use the following commands to load the module, list all modules, and remove the module:

```
$ sudo insmod mymod.ko        (inserting a module)
$ lsmod                       (list all modules)
$ sudo rmmod mymod.ko         (remove the module)
```

Also, you can use `modinfo mymod.ko` to show information about a Linux Kernel module.

3.2 A Simple Program that Uses Netfilter

Using `Netfilter` is quite straightforward. All we need to do is to hook our functions (in the kernel module) to the corresponding `Netfilter` hooks. Here we show an example:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>


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

/* This is the hook function itself */
unsigned int hook_func(void *priv, struct sk_buff *skb,
                       const struct nf_hook_state *state)

{
    /* This is where you can inspect the packet contained in
       the structure pointed by skb, and decide whether to accept
       or drop it. You can even modify the packet */

    // In this example, we simply drop all packets
    return NF_DROP;           /* Drop ALL packets */
}

/* Initialization routine */
int init_module()
{   /* Fill in our hook structure */
    nfho.hook = hook_func;         /* Handler function */
    nfho.hooknum  = NF_INET_PRE_ROUTING; /* First hook for IPv4 */
    nfho.pf       = PF_INET;
    nfho.priority = NF_IP_PRI_FIRST;   /* Make our function first */

    nf_register_hook(&nfho);
    return 0;
}

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

4 Submission and Demonstration

Students need to submit a detailed lab report to describe what they have done, what they have observed, and explanation. Reports should include the evidences to support the observations. Evidences include packet traces, screendumps, etc. Students also need to answer all the questions in the lab description. For the programming tasks, students should list the important code snippets followed by explanation. Simply attaching code without any explanation is not enough.

A Firewall Lab Cheat Sheet

Header Files. You may need to take a look at several header files, including the `skbuff.h`, `ip.h`, `icmp.h`, `tcp.h`, `udp.h`, and `netfilter.h`. They are stored in the following folder:

```
/lib/modules/$(uname -r)/build/include/linux/
```

IP Header. The following code shows how you can get the IP header, and its source/destination IP addresses.

```
struct iphdr *ip_header = (struct iphdr *)skb_network_header(skb);
unsigned int src_ip = (unsigned int)ip_header->saddr;
unsigned int dest_ip = (unsigned int)ip_header->daddr;
```

TCP/UDP Header. The following code shows how you can get the UDP header, and its source/destination port numbers. It should be noted that we use the `ntohs()` function to convert the unsigned short integer from the network byte order to the host byte order. This is because in the 80x86 architecture, the host byte order is the Least Significant Byte first, whereas the network byte order, as used on the Internet, is Most Significant Byte first. If you want to put a short integer into a packet, you should use `htons()`, which is reverse to `ntohs()`.

```
struct udphdr *udp_header = (struct udphdr *)skb_transport_header(skb);
src_port = (unsigned int)ntohs(udp_header->source);
dest_port = (unsigned int)ntohs(udp_header->dest);
```

IP Addresses in different formats. You may find the following library functions useful when you convert IP addresses from one format to another (e.g. from a string `"128.230.5.3"` to its corresponding integer in the network byte order or the host byte order.

```
int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
in_addr_t inet_network(const char *cp);
char *inet_ntoa(struct in_addr in);
struct in_addr inet_makeaddr(int net, int host);
in_addr_t inet_lnaof(struct in_addr in);
in_addr_t inet_netof(struct in_addr in);
```

Using `ufw`. The default firewall configuration tool for Ubuntu is `ufw`, which is developed to ease `iptables` firewall configuration. By default UFW is disabled, so you need to enable it first.

```
$ sudo ufw enable            // Enable the firewall
$ sudo ufw disable           // Disable the firewall
$ sudo ufw status numbered   // Display the firewall rules
$ sudo ufw delete 2          // Delete the 2nd rule
```