CYBR 435: Cyber Risk
Spring 2022

Lab Assignment #5: Firewall Exploration

- Name only: ________________________________
- Release date: Mar 03, 2022 (Thursday), 2:00 pm
- Due date: Mar 10, 2022 (Thursday), 2:00 pm
- Assignment should be **SUBMITTED on Blackboard before Due Date**. Other submission methods will NOT be accepted.
- **LATE Submission will NOT Be Accepted** on Blackboard since the submission link will be closed automatically after due date;
  - o Additional submission for missing answer **will NOT Be Accepted**.
- It should be done INDIVIDUALLY; **Show ALL your work and evidence to support your answers**.
  - o Answer only without evidence receives half credits.
- Total: 10 pts
- The Lab is adopted from Dr. Wenliang Du at Syracuse University.

Overview
The learning objective of this lab is two-fold: learning how firewalls work, and setting up a simple firewall for a network. Students 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. Through this implementation task, students can get the basic ideas on how firewall works.This lab covers the following topics:

- Firewall
- Netfilter
- Loadable kernel module

Software Requirements
This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website (https://seedsecuritylabs.org/labsetup.html), and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

Environment Setup Using Containers
In this lab, we need to use multiple machines. Their setup is depicted in Figure 1. We will use containers to set up this lab environment.

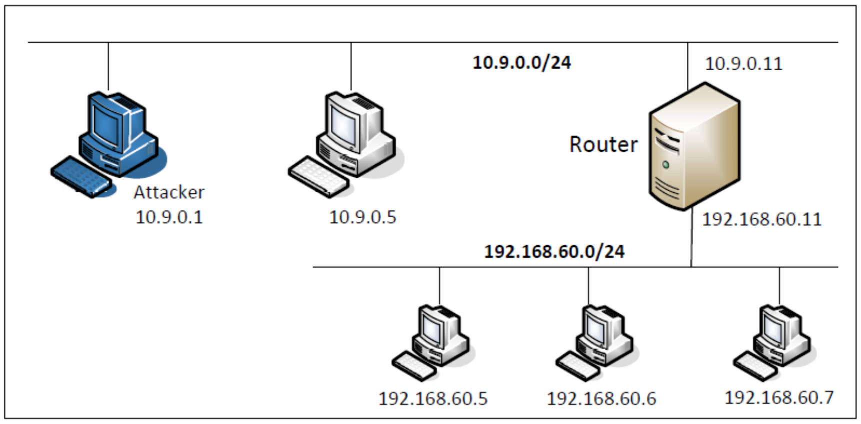

Figure 1: Lab setup

Container Setup and Commands

Please download the Labsetup.zip file (https://seedsecuritylabs.org/Labs_20.04/Networking/Firewall/) to your VM from the lab's website, unzip it, enter the Labsetup folder, and use the docker-compose.yml file to set up the lab environment.

Detailed explanation of the content in this file and all the involved Dockerfile can be found from the user manual, which is linked to the website of this lab. If this is the first time you set up a SEED lab environment using containers, it is very important that you read the user manual.

In the following, we list some of the commonly used commands related to Docker and Compose. Since we are going to use these commands very frequently, we have created aliases for them in the .bashrc file (in our provided SEEDUbuntu 20.04 VM).

```
$ docker-compose build  # Build the container image
$ docker-compose up     # Start the container
$ docker-compose down   # Shut down the container

// Aliases for the Compose commands above
$ dcbuild       # Alias for: docker-compose build
$ dcup          # Alias for: docker-compose up
$ dcdown        # Alias for: docker-compose down
```

All the containers will be running in the background. To run commands on a container, we often need to get a shell on that container. We first need to use the "docker ps" command to find out the ID of the container, and then use "docker exec" to start a shell on that container. We have created aliases for them in the .bashrc file.

```
$ dockps        // Alias for: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// Note: If a docker command requires a container ID, you do not need to
//       type the entire ID string. Typing the first few characters will
//       be sufficient, as long as they are unique among all the containers.
```

If you encounter problems when setting up the lab environment, please read the "Common Problems" section of the manual for potential solutions.

Task 1: Implementing a Simple Firewall

In this task, we will implement a simple packet filtering type of firewall, 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.

Notes about containers:

Since all the containers share the same kernel, kernel modules are global. Therefore, if we set a kernel model from a container, it affects all the containers and the host. For this reason, it does not matter where you set the kernel module. In this lab, we will just set the kernel module from the host VM.

Another thing to keep in mind is that containers' IP addresses are virtual. Packets going to these virtual IP addresses may not traverse the same path as what is described in the Netfilter document. Therefore, in this task, to avoid confusion, we will try to avoid using those virtual addresses. We do most tasks on the host VM. The containers are mainly for the other tasks.

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. In this task, we will get familiar with LKM.

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" to view the messages.

Listing 1: hello.c (included in the lab setup files)

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

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

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

module_init(initialization);
module_exit(cleanup);
```

We now need to create Makefile, which includes the following contents (the file is included in the lab setup files). Just type make, and the above program will be compiled into a loadable kernel module (if you copy and paste the following into Makefile, make sure replace the spaces before the make commands with a tab).

```
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
```

The generated kernel module is in hello.ko. You can use the following commands to load the module, list all modules, and remove the module. Also, you can use "modinfo hello.ko" to show information about a Linux Kernel module.

```
$ sudo insmod hello.ko      (inserting a module)
$ lsmod | grep hello        (list modules)
$ sudo rmmod hello          (remove the module)
$ dmesg                     (check the messages)
```

Question:
Please compile this simple kernel module on your VM, and run it on the VM. For this task, we will not use containers. In your submission, source codes and screenshots of running results are reuqired.) [2 pts]

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.

Netfilter is designed to facilitate the manipulation of packets by authorized users. It 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 a 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. We would like students to focus on the filtering part, the core of firewalls, so students are allowed to hardcode firewall policies in the program.

Hooking to 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 (the code is in Labsetup/packet_filter, but it may not be exactly the same as this example).
The structure of the code follows the structure of the kernel module implemented earlier. When the kernel module is added to the kernel, the registerFilter() function in the code will be invoked. Inside this function, we register two hooks to netfilter.
To register a hook, you need to prepare a hook data structure, and set all the needed parameters, the most important of which are a function name (Line 1) and a hook number (Line 2). The hook number is one of the 5 hooks in netfilter, and the specified function will be invoked when a packet has reached this hook. In this example, when a packet gets to the LOCAL IN hook, the function printInfo() will be invoked (this function will be given later). Once the hook data structure is prepared, we attach the hook to netfilter in Line 3).

Listing 2: Register hook functions to netfilter

```
static struct nf_hook_ops hook1, hook2;

int registerFilter(void) {
   printk(KERN_INFO "Registering filters.\n");

   // Hook 1
   hook1.hook = printInfo;                       ❶
   hook1.hooknum = NF_INET_LOCAL_IN;             ❷
   hook1.pf = PF_INET;
   hook1.priority = NF_IP_PRI_FIRST;
   nf_register_net_hook(&init_net, &hook1);      ❸

   // Hook 2
   hook2.hook = blockUDP;
   hook2.hooknum = NF_INET_POST_ROUTING;
   hook2.pf = PF_INET;
   hook2.priority = NF_IP_PRI_FIRST;
   nf_register_net_hook(&init_net, &hook2);

   return 0;
}

void removeFilter(void) {
   printk(KERN_INFO "The filters are being removed.\n");
   nf_unregister_net_hook(&init_net, &hook1);
   nf_unregister_net_hook(&init_net, &hook2);
}

module_init(registerFilter);
module_exit(removeFilter);
```

Note for Ubuntu 20.04 VM:

The code in the SEED lab was developed in Ubuntu 16.04. It needs to be changed slightly to work in Ubuntu 20.04. The change is in the hook registration and un-registration APIs.

See the difference in the following:

```
// Hook registration:
  nf_register_hook(&nfho);                    // For Ubuntu 16.04 VM
  nf_register_net_hook(&init_net, &nfho);     // For Ubuntu 20.04 VM

// Hook unregistration:
  nf_unregister_hook(&nfho);                  // For Ubuntu 16.04 VM
  nf_unregister_net_hook(&init_net, &nfho);   // For Ubuntu 20.04 VM
```

Hook functions

We give an example of hook function below. It only prints out the packet information. When netfilter invokes a hook function, it passes three arguments to the function, including a pointer to the actual packet (skb). In the following code, Line 1 shows how to retrieve the hook number from the state argument. In Line 2, we use ip hdr() function to get the pointer for the IP header, and then use the %pI4 format string specifier to print out the source and destination IP addresses in Line 3.

Listing 3: An example of hook function

```
unsigned int printInfo(void *priv, struct sk_buff *skb,
                       const struct nf_hook_state *state)
{
   struct iphdr *iph;
   char *hook;

   switch (state->hook){           ❶
     case NF_INET_LOCAL_IN:
          printk("*** LOCAL_IN"); break;
     .. (code omitted) ...
   }

   iph = ip_hdr(skb);              ❷
   printk("    %pI4  --> %pI4\n", &(iph->saddr), &(iph->daddr)); ❸
   return NF_ACCEPT;
}
```

If you need to get the headers for other protocols, you can use the following functions defined in various header files. The structure definition of these headers can be found inside the */lib/modules/5.4.0-54 generic/build/include/uapi/linux* folder, where the version number in the path is the result of "uname -r", so it may be different if the kernel version is different.

```
struct iphdr   *iph   = ip_hdr(skb)    // (need to include <linux/ip.h>)
struct tcphdr  *tcph  = tcp_hdr(skb)   // (need to include <linux/tcp.h>)
struct udphdr  *udph  = udp_hdr(skb)   // (need to include <linux/udp.h>)
struct icmphdr *icmph = icmp_hdr(skb)  // (need to include <linux/icmp.h>)
```

Blocking packets

We also provide a hook function example to show how to block a packet, if it satisfies the specified condition. The following example blocks the UDP packets if their destination IP is 8.8.8.8 and the destination port is 53. This means blocking the DNS query to the nameserver 8.8.8.8.

Listing 4: Code example: blocking UDP

```c
unsigned int blockUDP(void *priv, struct sk_buff *skb,
                      const struct nf_hook_state *state)
{
   struct iphdr *iph;
   struct udphdr *udph;
   u32  ip_addr;
   char ip[16] = "8.8.8.8";

   // Convert the IPv4 address from dotted decimal to a 32-bit number
   in4_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);                      ❶

   iph = ip_hdr(skb);
   if (iph->protocol == IPPROTO_UDP) {
       udph = udp_hdr(skb);
       if (iph->daddr == ip_addr && ntohs(udph->dest) == 53){          ❷
            printk(KERN_DEBUG "****Dropping %pI4 (UDP), port %d\n",
                               &(iph->daddr), port);
            return NF_DROP;                                            ❸
        }
   }
   return NF_ACCEPT;                                                   ❹
}
```

In the code above, Line 1 shows, inside the kernel, how to convert an IP address in the dotted decimal format (i.e., a string, such as 1.2.3.4) to a 32-bit binary (0x01020304), so it can be compared with the binary number stored inside packets. Line 2 compares the destination IP address and port number with the values in our specified rule. If they match the rule, the NF DROP will be returned to netfilter, which will drop the packet. Otherwise, the NF ACCEPT will be returned, and netfilter will let the packet continue its journey (NF ACCEPT only means that the packet is accepted by this hook function; it may still be dropped by other hook functions).

The complete sample code is called seedFilter.c, which is included in the lab setup files (inside the Files/packet_filter folder). Please do the following tasks (do each of them separately):

1. Compile the sample code using the provided Makefile. Load it into the kernel, and demonstrate that the firewall is working as expected. You can use the following command to generate UDP packets to 8.8.8.8, which is Google's DNS server. If your firewall works, your request will be blocked; otherwise, you will get a response.

   ```
   dig @8.8.8.8 www.example.com
   ```

   Question:
   In your submission, source codes and screenshots of running results are reuqired.) [2 pts]

2. Hook the printInfo function to all of the netfilter hooks. Here are the macros of the hook numbers. Using your experiment results to help explain at what condition will each of the hook function be invoked.

   ```
   NF_INET_PRE_ROUTING
   NF_INET_LOCAL_IN
   NF_INET_FORWARD
   NF_INET_LOCAL_OUT
   NF_INET_POST_ROUTING
   ```

   Question:
   In your submission, source codes, screenshots of running results along with writing explanation are reuqired.) [2 pts]

3. Implement two more hooks to achieve the following: (1) preventing other computers to ping the VM, and (2) preventing other computers to telnet into the VM. Please implement two different hook functions, but register them to the same netfilter hook. You should decide what hook to use. Telnet's default port is TCP port 23. To test it, you can start the containers, go to 10.9.0.5, run the following commands (10.9.0.1 is the IP address assigned to the VM; for the sake of simplicity, you can hardcode this IP address in your firewall rules):

   ```
   ping 10.9.0.1
   telnet 10.9.0.1
   ```

   Question:
   In your submission, source codes and screenshots of running results are reuqired.) [4 pts]

Your submission should be a WORD file containing:

1. Answers for Task 1.A: Implement a Simple Kernel Module (source codes and screenshots of running results are required)
2. Answers for Task 1.B: Implement a Simple Firewall Using Netfilter
   1. Compile the sample code using the provided Makefile……. (source codes and screenshots of running results are required)
   2. Hook the printInfo function to all of the netfilter hooks……. (source codes and screenshots of running results along with writing explanation are required)
   3. Implement two more hooks……(source codes and screenshots of running results are required)

Important note:
Since we make changes to the kernel, there is a high chance that you would crash the kernel. Make sure you back up your files frequently, so you don't lose them. One of the common reasons for system crash is that you forget to unregister hooks. When a module is removed, these hooks will still be triggered, but the module is no longer present in the kernel. That will cause system crash. To avoid this, make sure for each hook you add to your module, add a line in removeFilter to unregister it, so when the module is removed, those hooks are also removed.

Happy Blocking!