CENG489 HW3-B: Writing a Linux Firewall

1 Overview

The learning objective of this assignment is to gain the insights on how firewalls work by implementing a simplified packet filtering firewall. Packet filters act by inspecting the packets; if a packet matches the packet filter's set of rules, the packet filter will either drop the packet or forward it, depending on what the rules say. 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 the packet's source and destination address, its protocol, and, for TCP and UDP traffic, port numbers.

2 Your Task: Implement a Simple Packet Filtering Firewall

The firewall you will implement in this task is a packet filtering 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 the kernel code, and rebuilding the entire kernel image. The modern Linux operating system provides several new mechanisms to facilitate the manipulation of packets without requiring the kernel image to be rebuilt. These two mechanisms are *Loadable Kernel Module* (`LKM`) and `Netfilter`.

`LKM` allows us to add a new module to the kernel at 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 firewalls 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.

`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, we allow you to hardcode your firewall policies in the program. Setup two VMs A and B, and run your firewall on A. You should support at least five different rules, including the ones below.

- Prevent A from doing a telnet to Machine B.
- Prevent B from doing a telnet to Machine A.
- Prevent A from visiting an external web site (e.g. Facebook). You can choose any web site that you like to block, but keep in mind, some web servers have multiple IP addresses.
- Prevent all traffic that has a destination port of 443 and show that you are not able to access a server using HTTPS after implementing this rule.
- Prevent all ICMP traffic and show that you are not able to ping another machine.

Optional: A real firewall should support a dynamic configuration, i.e., the administrator can dynamically change the firewall policies. The firewall configuration tool runs in the user space, but it has to send the data to the kernel space, where your packet filtering module (a LKM) can get the data. The policies must be stored in the kernel memory. You cannot ask your LKM to get the policies from a file, because that will significantly slow down your firewall. This involves interactions between a user-level program and the kernel module, which is not very difficult to implement. We have provided some guidelines in Section 3. Implementing the dynamic configuration is optional.

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.

```
[frame=single]
#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 Interacting with Loadable Kernel Module (for the Optional Task)

In firewall implementation, the packet filtering part is implemented in the kernel, but the policy setting is done at the user space. We need a mechanism to pass the policy information from a user-space program to the kernel module. There are several ways to do this; a standard approach is to use `/proc`. Please read the article from `http://www.ibm.com/developerworks/linux/library/l-proc.html` for detailed instructions. Once we set up a `/proc` file for our kernel module, we can use the standard `write()` and `read()` system calls to pass data to and from the kernel module.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/string.h>
#include <linux/vmalloc.h>
#include <asm/uaccess.h>
```

```
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Fortune Cookie Kernel Module");
MODULE_AUTHOR("M. Tim Jones");

#define MAX_COOKIE_LENGTH       PAGE_SIZE

static struct proc_dir_entry *proc_entry;
static char *cookie_pot;  // Space for fortune strings
static int cookie_index;  // Index to write next fortune
static int next_fortune;  // Index to read next fortune

ssize_t fortune_write( struct file *filp, const char __user *buff,
                       unsigned long len, void *data );

int fortune_read( char *page, char **start, off_t off,
                  int count, int *eof, void *data );

int init_fortune_module( void )
{
  int ret = 0;

  cookie_pot = (char *)vmalloc( MAX_COOKIE_LENGTH );

  if (!cookie_pot) {
    ret = -ENOMEM;
  } else {
    memset( cookie_pot, 0, MAX_COOKIE_LENGTH );
    proc_entry = create_proc_entry( "fortune", 0644, NULL );
    if (proc_entry == NULL) {
      ret = -ENOMEM;
      vfree(cookie_pot);
      printk(KERN_INFO "fortune: Couldn't create proc entry\n");
    } else {
      cookie_index = 0;
      next_fortune = 0;
      proc_entry->read_proc = fortune_read;
      proc_entry->write_proc = fortune_write;

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

  return ret;
}
```

```
void cleanup_fortune_module( void )
{
  remove_proc_entry("fortune", NULL);
  vfree(cookie_pot);
  printk(KERN_INFO "fortune: Module unloaded.\n");
}

module_init( init_fortune_module );
module_exit( cleanup_fortune_module );
```

The function to read a fortune is shown as following:

```
[frame=single]
int fortune_read( char *page, char **start, off_t off,
                  int count, int *eof, void *data )
{
  int len;

  if (off > 0) {
    *eof = 1;
    return 0;
  }

  /* Wrap-around */
  if (next_fortune >= cookie_index) next_fortune = 0;
  len = sprintf(page, "%s\n", &cookie_pot[next_fortune]);
  next_fortune += len;

  return len;
}
\end{Verbatim}

The function to write a fortune is shown as following. Note that
we use $copy\_from\_user$ to copy the user buffer directly into the $cookie\_pot$.

\begin{verbatim}
ssize_t fortune_write( struct file *filp, const char __user *buff,
                       unsigned long len, void *data )
{
  int space_available = (MAX_COOKIE_LENGTH-cookie_index)+1;

  if (len > space_available) {
    printk(KERN_INFO "fortune: cookie pot is full!\n");
    return -ENOSPC;
  }
```

```
  if (copy_from_user( &cookie_pot[cookie_index], buff, len )) {
    return -EFAULT;
  }

  cookie_index += len;
  cookie_pot[cookie_index-1] = 0;
  return len;
}
```

3.3 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(unsigned int hooknum,
                       struct sk_buff *skb,
                       const struct net_device *in,
                       const struct net_device *out,
                       int (*okfn)(struct sk_buff *))
{
    /* 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);
}
```

When compiling some of the examples from the tutorial, you might see an error that says that
`NF_IP_PRE_ROUTING` is undefined. Most likely, this example is written for the older Linux kernel. Since version 2.6.25, kernels have been using `NF_INET_PRE_ROUTING`. Therefore, replace `NF_IP_PRE_ROUTING` with `NF_INET_PRE_ROUTING`, this error will go away (the replacement is already done in the code above).

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);
```