Linux Firewall Lab

Copyright © 2006 - 2011 Wenliang Du, Syracuse University.
The development of this document is funded by the National Science Foundation's Course, Curriculum, and Laboratory Improvement (CCLI) program under Award No. 0618680 and 0231122. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation. A copy of the license can be found at http://www.gnu.org/licenses/fdl.html.

1 Overview

The learning objective of this lab is for students to gain the insights on how firewalls work by designing and implementing a simple personal firewall for `Linux`. A personal firewall controls network traffic to and from a computer, permitting or denying communications based on the security policies set by the administrators.

Firewalls have several types; in this lab, we focus on a very simple type, the *packet filter*. Packet filters act by inspecting the packets; if a packet matches the packet filter's set of rules, the packet filter will drop the packet either silently or send an "error responses" to the source. 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, the port number.

2 Lab Tasks

In this lab, students need to implement a packet filter for `Linux`. We will call it `miniFirewall`. This firewall consists of two components: policy configuration and packet filtering.

2.1 Task 1: Firewall Policies

The policy configuration module is intended for allowing system administrators to set up the firewall policies. There are many types of policies that can be supported by personal firewalls, starting from very simple to fairly complex. For `miniFirewall`, the minimal requirements on policies are described in the following, but you are encouraged (and will be rewarded) if your firewall can support more sophisticated policies. Basically, your firewall should be able to block or unblock incoming and outgoing packets based on the following criteria:

1. *Protocol:* It specifies which protocol a policy applies to. The protocol can be TCP, UDP, or ICMP.
2. *Source and Destination address:* Match packets with source and destination addresses. As used by many packet filters, address/netmask combination is often used to block an address range.
3. *Source and Destination port number:* Match packets with source and destination port numbers.
4. *Action:* Specify the actions when a packet matches with a rule. Common actions include
   - `BLOCK`: block packets.
   - `UNBLOCK`: used in conjunction with BLOCK to allow packets from just one address through while the entire network is blocked.

Configuration Tools. You need to implement a tool to allow the administrator to configure the firewall policies. Let us call this tool `minifirewall`. We give a few examples on how this tool can be used. However, feel free to change the syntax according to your own preference.

- `minifirewall --in --proto ALL --action BLOCK`
  Block all incoming packets.
- `minifirewall --in --proto TCP --action UNBLOCK`
  Allow only TCP incoming packets.
- `minifirewall --in --srcip 172.16.75.43 --proto ALL --action BLOCK`
  Block all the packets from the given IP address.
- `minifirewall --out --destip 172.20.33.22 --proto UDP --action UNBLOCK`
  Unblock the outgoing UDP packets if the destination is 172.20.33.22
- `minifirewall --in --srcip 172.16.0.0 --srcnetmask 255.255.0.0 --destport 80 --proto TCP --action BLOCK`
  Block all incoming TCP packets from the `172.16.0.0/16` network if the packets are directed towards port 80.
- `minifirewall --print`
  Print all rules.
- `minifirewall --delete 3`
  Delete the 3rd rule.

2.2 Task 2: Packet Filtering

The main part of firewall is the filtering part, which 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. This requires us to modify the `Linux` kernel. In the past, this has to be done by modifying the kernel code, and rebuild 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 on 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 our `miniFirewall` can be implemented as an LKM. However, this is not enough. In order for our module to block incoming/outgoing packets, our module must be inserted into the packet processing path. This cannot be easily done in the past before the `Netfilter` is 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. You should be able to support a dynamic configuration, i.e., the administrator can dynamically change the firewall policies, your packet filtering module must automatically enforce the updated policies.

Storage of policies. Since your configuration tool runs in the user space, the tool has to send the data to the kernel space, where your packet filtering module, which is 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.

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 Interacting with Loadable Kernel Module

In our `miniFirewall`, 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:

```
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;
}
```

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 $cookie\_pot$.

```
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 from an online tutorial, which is available at the following URL:

`http://www.topsight.net/article.php/2003050621055083/print`

```
#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 *))
{
    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).

3.4 The `iptables` and `ufw` programs

`Linux` has a tool called `iptables`, which is essentially a firewall built upon the `Netfilter` mechanism. In addition, `Linux` has another tool called `ufw`, which a front end to `iptables` (it is easier to use than `iptables`). You can consider `miniFirewall` as a mini-version of `iptables` or `ufw`. You are encourage to play with these programs to gain some inspiration for your own design. However, copying the code from them is strictly forbidden. Moreover, you may find some tutorials that also provide sample code for simple firewalls. You can learn from those tutorials and play with the sample code, but you have to write your own code. In your lab reports, you need to submit your code, and explain the key parts of your code.

3.5 Parsing Command Line Arguments

Because our `miniFirewall` program needs to recognize command line arguments, we need to parse these arguments. If the syntax for the command line arguments is simple enough, we can directly write code to parse them. However, our `miniFirewall` has to recognized options with a a fairly sophisticated syntax. We can use `getopt()` and `getopt_long()` to systematically parse command line arguments. Please read the tutorial in the following URL:

`http://www.gnu.org/s/libc/manual/html_node/Getopt.html`

4 Submission and Demonstration

You should submit a detailed lab report to describe your design and implementation. You should also describe how you test the functionalities and security of your system. Please list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.