Linux Firewall Exploration Lab

The development of this document is/was funded by three grants from the US National Science Foundation: Awards No. 0231122 and 0618680 from TUES/CCLI and Award No. 1017771 from Trustworthy Computing.

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 two types, the *packet filter* and application 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 foward 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. Application firewall works at the application layer. A widely used application firewall is web proxy, which is primarily used for egress filtering of web traffic. In this lab, students will play with both types of firewalls, and also through the implementation of some of the key functionalities, they can understand how firewalls work.

Note for Instructors. If the instructor plans to hold lab sessions for this lab, it is suggested that the following be covered:

- Loadable kernel module.
- The Netfilter mechanism.

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 the 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 also list some commonly used commands in Appendix A.

Before start the task, go to default policy file `/etc/default/ufw`. If `DEFAULT_INPUT_POLICY` is `DROP`, please change it to `ACCEPT`. Otherwise, all the incoming traffic will be dropped by default.

2.2 Task 2: How Firewall Works

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 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 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. 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.

Question 1: What types of hooks does `Netfilter` support, and what can you do with these hooks? Please draw a diagram to show how packets flow through these hooks.

Question 2: Where should you place a hook for ingress filtering, and where should you place a hook for egress filtering?

Question 3: Can you modify packets using `Netfilter`?

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; we also linked some tutorials in the web site. Implementing the dynamic configuration is optional. It is up to your instructor to decide whether you will receive bonus points for this optional task.

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 block 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 (run `sudo service openbsd-inetd start`). 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: `Tools -> Clear Recent History`; otherwise, the cached pages may be displayed. If everything is set up properly, you should not be able to see the 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.
```

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. The following command establish an SSH tunnel from the localhost's port 8000 and machine B,

and 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 replace Machine C with Machine B.

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

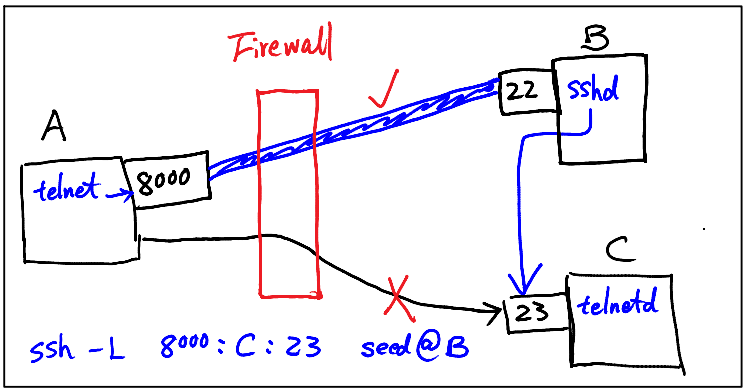

Figure 1: SSH Tunnel Example

After establishing the above tunnel, you can telnet to your localhost using port 8000: `telnet localhost 8000`. SSH will transfer all your TCP packets from your end of the tunnel (localhost:8000) to Machine B, and from there, the packets will be forwarded to Machine_C:23. Replies from Machine C will take a reverse path, and eventually reach your telnet client. This resulting in your telneting 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: Connecting 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 you 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
```

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. The following procedure achieves this:

```
Edit -> Preference -> Advanced tab -> Network tab -> Settings button.

Select Manual proxy configuration
SOCKS Host: 127.0.0.1      Port: 9000
```

```
SOCKS v5
No Proxy for: localhost, 127.0.0.1
```

After the setup is done, please do the followings:

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.

Question 4: If `ufw` blocks the TCP port 22, which is the port used by SSH, can you still set up an SSH tunnel to evade egress filtering?

2.4 Task 4: Web Proxy (Application Firewall)

There is another type of firewalls, which are specific to applications. Instead of inspecting the packets at the transport layer (such as TCP/UDP) and below (such as IP), they look at the application-layer data, and enforce their firewall policies. These firewalls are called application firewalls, which controls input, output, and/or access from, to, or by an application or service. A widely used category of application firewalls is web proxy. which is used to control what their protected browsers can access. This is a typical egress filtering, and it is widely used by companies and schools to block their employees or students from accessing distracting or inappropriate web sites.

In this task, we will set up a web proxy and perform some tasks based on this web proxy. There are a number of web proxy products to choose from. In this lab, we will use a very well-known free software, called `squid`. If you are using our pre-built VM (Ubuntu 12.04), this software is already installed. Otherwise, you can easily run the following command to install it.

```
$ sudo apt-get install squid

Here are several commands that you may need:

$ sudo service squid3 start     // to start the server
$ sudo service squid3 restart   // to restart the server
```

Once you have installed `squid`, you can go to `/etc/squid3`, and locate the configuration file called `squid.conf`. This is where you need to set up your firewall policies. Keep in mind that every time you make a change to `squid.conf`, you need to restart the `squid` server; otherwise, your changes will not take effect.

Task 4.a: Setup. You need to set up two VMs, Machine A and Machine B. Machine A is the one whose browsing behaviors need to restricted, and Machine B is where you run the web proxy. We need to configure A's Firefox browser, so it always use the web proxy server on B. To achieve that, we can tell Firefox to use `B:3128` as its proxy (by default, `squid` uses port `3128`, but this can be changed in `squid.conf`). The following procedure configures `B:3128` as the proxy for Firefox.

```
Edit -> Preferences -> Advanced tab -> Network tab -> Settings button.

Select "Manual proxy configuration"
Fill in the following information:
HTTP Proxy: B's IP address      Port: 3128
```

Note: to ensure that the browser always uses the proxy server, the browser's proxy setting needs to be locked down, so users cannot change it. There are ways for administrators to do that. If you are interested, you can search the Internet for instructions.

After the setup is done, please perform the following tasks:

1. Try to visit some web sites from Machine A's Firefox browser, and describe your observation.
2. By default, all the external web sites are blocked. Please Take a look at the configuration file `squid.conf`, and describe what rules have caused that (hint: search for the `http_access` tag in the file).
3. Make changes to the configuration file, so all the web sites are allowed.
4. Make changes to the configuration file, so only the access to `google.com` is allowed.

You should turn on your Wireshark, capture packets while performing the above tasks. In your report, you should describe what exactly is happening on the wire. Using these observations, you should describe how the web proxy works.

Task 4.b: Using Web Proxy to evade Firewall. Ironically, web proxy, which is widely used to do the egress filtering, is also widely used to bypass egress filtering. Some networks have packet-filter type of firewall, which blocks outgoing packets by looking at their destination addresses and port numbers. For example, in Task 1, we use `ufw` to block the access of Facebook. In Task 3, we have shown that you can use a SSH tunnel to bypass that kind of firewall. In this task, you should do it using a web proxy. Please demonstrate how you can access Facebook even if `ufw` has already blocked it. Please describe how you do that and also include evidence of success in your lab report.

Question 5: If `ufw` blocks the TCP port 3128, can you still use web proxy to evade the firewall?

Task 4.c: URL Rewriting/Redirection. Not only can `squid` block web sites, they can also rewrite URLs or redirect users to another web site. For example, you can set up `squid`, so every time a user tries to visit Facebook, you redirect them to a web page, which shows a big red stop sign. `Squid` allows you to make any arbitrary URL rewriting: all you need to do is to connect `squid` to an URL rewriting program. In this task, you will write a very simple URL-rewriting programming to demonstrate such a functionality of web proxy. You can use any programming language, such as Perl, Java, C/C++, etc.

Assume that you have written a program called `myprog.pl` (a perl program). You need to modify `squid.conf` to connect to this program. You need to add the following to the configuration file:

```
url_rewrite_program /home/seed/myprog.pl
url_rewrite_children 5
```

In case `squid` cannot find `myprog.pl` under `/home/seed`, you can put your url rewrite program under `/etc/squid3`.

This is how it works: when `squid` receives an URL from browsers, it will invoke `myprog.pl`, which can get the URL information from the standard input (the pipe mechanism is used). The rewriting program can do whatever it wants to the URL, and eventually print out a new URL or an empty line to the standard output, which is then piped back to `squid`. `Squid` uses this output as the new URL, if the line is not empty. This is how an URL gets rewritten. The following Perl program gives you an example:

```
#!/usr/bin/perl -w
use strict;
use warnings;

# Forces a flush after every write or print on the STDOUT
select STDOUT; $| = 1;

# Get the input line by line from the standard input.
# Each line contains an URL and some other information.
while (<>)
{
    my @parts = split;
    my $url = $parts[0];
    # If you copy and paste this code from this PDF file,
    # the ~ (tilde) character may not be copied correctly.
    # Remove it, and then type the character manually.
    if ($url =~ /www\.cis\.syr\.edu/) {
        # URL Rewriting
        print "http://www.yahoo.com\n";
    }
    else {
        # No Rewriting.
        print "\n";
    }
}
```

Please conduct the following tasks:

1. Please describe what the above program does (e.g. what URLs got rewritten and what your observations are).

2. Please modify the above program, so it replaces all the Facebook pages with a page that shows a big red stop sign.

3. Please modify the above program, so it replaces all the images (e.g. `.jpg` or `.gif` images) inside any page with a picture of your choice. When an HTML web page contains images, the browser will

identify those image URLs, and send out an URL request for each image. You should be able to see the URLs in your URL rewriting program. You just need to decide whether a URL is trying to fetch an image file of a particular type; if it is, you can replace the URL with another one (of your choice).

Note: If you have a syntax error in the above program, you will not be able to see the error message if you test it via the web proxy. We suggest that you run the above perl program on the command line first, manually type an URL as the input, and see whether the program functions as expected. If there is a syntax error, you will see the error message. Do watch out for the tilde sign if you copy and paste the above program from the PDF file.

Task 4.d (Optional): A Real URL Redirector. The program you wrote above is a toy URL rewriting program. In reality, such a program is much more complicated because of performance issues and many other functionalities. Moreover, they work with real URL datasets (the URLs that need to be blocked and rewritten) that contains many URLs. These datasets can be obtained from various sources (some are free and some are not), and they need to be routinely updated. If you are interested in playing with a real URL rewriting program, you can install `SquidGuard` and also download a blacklist URL dataset that works with it. Here is some related information:

- You can download `SquidGuard` from `http://www.squidguard.org/`, compile and install it. You can get the instruction from the following URL `http://www.squidguard.org/Doc/`.
- You also need to download Oracle Berkeley DB, because `SquidGuard` depends on it. You should do this step first.
- Download a blacklist data set from the following URL:

  `http://www.squidguard.org/blacklists.html`.

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 (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:

```
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:

```
#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).

3.4 The Lightweight Firewall Code

Owen Klan wrote a very nice lightweight firewall program, which we have linked in this lab's web page (`lwfw.tar.gz`). From this sample code, you can see how to write a simple hook for Netfilter, and how to retrieve the IP and TCP/UDP headers inside the hook. You can start with this program, make it work, and gradually expand it to support more sophisticated firewall policies (this sample program only enforces a very simple policy).

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.

Question 6: We can use the SSH and HTTP protocols as tunnels to evade the egress filtering. Can we use the ICMP protocol as a tunnel to evade the egress filtering? Please briefly describe how.

A Firewall Lab CheatSheet

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 diffrent 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
```

Using `squid`**.** The following commands are related to `squid`.

```
$ sudo service squid3 start          // start the squid service
$ sudo service squid3 restart        // restart the squid service
$ sudo service squid3 stop           // stop the squid service

/etc/squid3/squid.conf:  This is the squid configuration file.
```