Firewall Exploration Lab

1 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.

Actually, Linux already has a built-in firewall, also based on `netfilter`. This firewall is called `iptables`. Students will be given a simple network topology, and are asked to use `iptables` to set up firewall rules to protect the network. Students will also be exposed to several other interesting applications of `iptables`. This lab covers the following topics:

- Firewall
- Netfilter
- Loadable kernel module
- Using `iptables` to set up firewall rules
- Various applications of `iptables`

Readings and videos. Detailed coverage of firewalls can be found in the following:

- Chapter 17 of the SEED Book, *Computer & Internet Security: A Hands-on Approach*, 2nd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.
- Section 9 of the SEED Lecture, *Internet Security: A Hands-on Approach*, by Wenliang Du. See details at `https://www.handsonsecurity.net/video.html`.

Lab environment. This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, 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.

2 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.

2.1 Container Setup and Commands

Please download the `Labsetup.zip` file 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

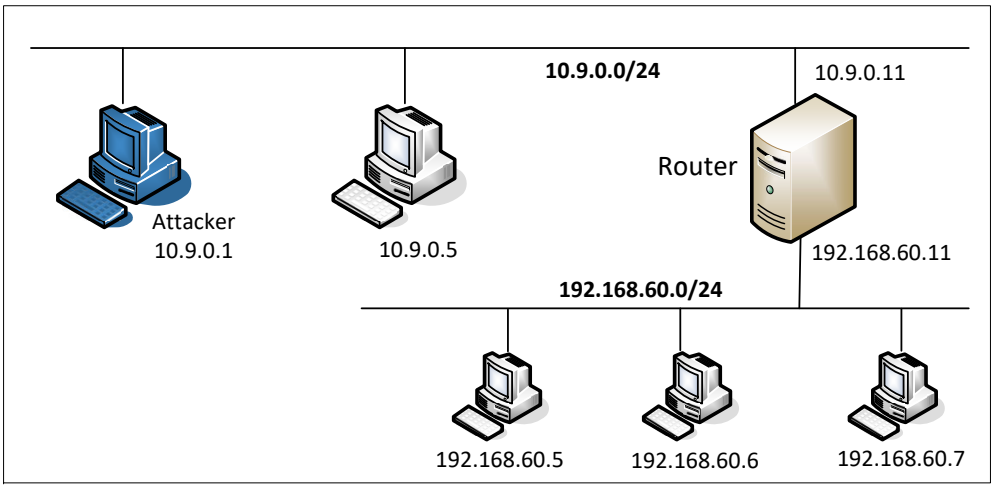


Figure 1: Lab setup

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.

3 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.

3.1 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)
```

Task. Please compile this simple kernel module on your VM, and run it on the VM. For this task, we will not use containers. Please show your running results in the lab report.

3.2 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. Detailed guidelines on how to use `Netfilter` can be found in Chapter 17 of the SEED book. We will provide some guidelines in here as well.

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 ❶) and a hook number (Line ❷). 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 ❸).

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 book 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 ❶ shows how to retrieve the hook number from the `state` argument. In Line ❷, 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 ❸.

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

```
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 ❶ 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 ❷ 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).

Tasks. 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
   ```

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
   ```

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
```

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.

4 Task 2: Experimenting with Stateless Firewall Rules

In the previous task, we had a chance to build a simple firewall using `netfilter`. Actually, `Linux` already has a built-in firewall, also based on `netfilter`. This firewall is called `iptables`. Technically, the kernel part implementation of the firewall is called `Xtables`, while `iptables` is a user-space program to configure the firewall. However, `iptables` is often used to refer to both the kernel-part implementation and the user-space program.

4.1 Background of `iptables`

In this task, we will use `iptables` to set up a firewall. The `iptables` firewall is designed not only to filter packets, but also to make changes to packets. To help manage these firewall rules for different purposes, `iptables` organizes all rules using a hierarchical structure: table, chain, and rules. There are several tables, each specifying the main purpose of the rules as shown in Table 1. For example, rules for packet filtering should be placed in the `filter` table, while rules for making changes to packets should be placed in the `nat` or `mangle` tables.

Each table contains several chains, each of which corresponds to a `netfilter` hook. Basically, each chain indicates where its rules are enforced. For example, rules on the `FORWARD` chain are enforced at the `NF_INET_FORWARD` hook, and rules on the `INPUT` chain are enforced at the `NF_INET_LOCAL_IN` hook.

Each chain contains a set of firewall rules that will be enforced. When we set up firewalls, we add rules to these chains. For example, if we would like to block all incoming `telnet` traffic, we would add a rule to the `INPUT` chain of the `filter` table. If we would like to redirect all incoming `telnet` traffic to a different port on a different host, basically doing port forwarding, we can add a rule to the `INPUT` chain of the `mangle` table, as we need to make changes to packets.

4.2 Using `iptables`

To add rules to the chains in each table, we use the `iptables` command, which is a quite powerful command. Students can find the manual of `iptables` by typing `"man iptables"` or easily find many tutorials from online. What makes `iptables` complicated is the many command-line arguments that we need to provide when using the command. However, if we understand the structure of these command-line arguments, we will find out that the command is not that complicated.

In a typical `iptables` command, we add a rule to or remove a rule from one of the chains in one of the tables, so we need to specify a table name (the default is `filter`), a chain name, and an operation on the chain. After that, we specify the rule, which is basically a pattern that will be matched with each of the packets passing through. If there is a match, an action will be performed on this packet. The general structure of the command is depicted in the following:

Table 1: `iptables` Tables and Chains

Table	Chain	Functionality

filter	`INPUT`
	`FORWARD`
	`OUTPUT`
		Packet filtering

nat	`PREROUTING`
	`INPUT`
	`OUTPUT`
	`POSTROUTING`
		Modifying source or destination network addresses

mangle	`PREROUTING`
	`INPUT`
	`FORWARD`
	`OUTPUT`
	`POSTROUTING`
		Packet content modification

```
iptables -t <table> -<operation> <chain>  <rule>   -j <target>
         ---------- --------------------  -------  -----------
            Table          Chain           Rule      Action
```

The rule is the most complicated part of the `iptables` command. We will provide additional information later when we use specific rules. In the following, we list some commonly used commands:

```
// List all the rules in a table (without line number)
iptables -t nat -L -n

// List all the rules in a table (with line number)
iptables -t filter -L -n --line-numbers


// Delete rule No. 2 in the INPUT chain of the filter table
iptables -t filter -D INPUT 2

// Drop all the incoming packets that satisfy the <rule>
iptables -t filter -A INPUT <rule>  -j DROP
```

Note. Docker relies on `iptables` to manage the networks it creates, so it adds many rules to the `nat` table. When we manipulate `iptables` rules, we should be careful not to remove Docker rules. For example, it will be quite dangerous to run the "`iptables -t nat -F`" command, because it removes all the rules in the `nat` table, including many of the Docker rules. That will cause trouble to Docker containers. Doing this for the `filter` table is fine, because Docker does not touch this table.

4.3 Task 2.A: Protecting the Router

In this task, we will set up rules to prevent outside machines from accessing the router machine, except ping. Please execute the following `iptables` command on the router container, and then try to access it from `10.9.0.5`. (1) Can you ping the router? (2) Can you telnet into the router (a telnet server is running on all the containers; an account called `seed` was created on them with a password `dees`). Please report your observation and explain the purpose for each rule.

```
iptables -A INPUT  -p icmp --icmp-type echo-request -j ACCEPT
iptables -A OUTPUT -p icmp --icmp-type echo-reply   -j ACCEPT
iptables -P OUTPUT DROP     ← Set default rule for OUTPUT
iptables -P INPUT  DROP     ← Set default rule for INPUT
```

Cleanup. Before moving on to the next task, please restore the `filter` table to its original state by running the following commands:

```
iptables -F
iptables -P OUTPUT ACCEPT
iptables -P INPUT  ACCEPT
```

Another way to restore the states of all the tables is to restart the container. You can do it using the following command (you need to find the container's ID first):

```
$ docker restart <Container ID>
```

4.4 Task 2.B: Protecting the Internal Network

In this task, we will set up firewall rules on the router to protect the internal network `192.168.60.0/24`. We need to use the FORWARD chain for this purpose.

The directions of packets in the INPUT and OUTPUT chains are clear: packets are either coming into (for INPUT) or going out (for OUTPUT). This is not true for the FORWARD chain, because it is bi-directional: packets going into the internal network or going out to the external network all go through this chain. To specify the direction, we can add the interface options using `"-i xyz"` (coming in from the `xyz` interface) and/or `"-o xyz"` (going out from the `xyz` interface). The interfaces for the internal and external networks are different. You can find out the interface names via the `"ip addr"` command.

In this task, we want to implement a firewall to protect the internal network. More specifically, we need to enforce the following restrictions on the ICMP traffic:

1. Outside hosts cannot ping internal hosts.
2. Outside hosts can ping the router.
3. Internal hosts can ping outside hosts.
4. All other packets between the internal and external networks should be blocked.

You will need to use the `"-p icmp"` options to specify the match options related to the ICMP protocol. You can run `"iptables -p icmp -h"` to find out all the ICMP match options. The following example drops the ICMP echo request.

```
iptables -A FORWARD -p icmp --icmp-type echo-request -j DROP
```

In your lab report, please include your rules and screenshots to demonstrate that your firewall works as expected. When you are done with this task, please remember to clean the table or restart the container before moving on to the next task.

4.5 Task 2.C: Protecting Internal Servers

In this task, we want to protect the TCP servers inside the internal network (`192.168.60.0/24`). More specifically, we would like to achieve the following objectives.

1. All the internal hosts run a telnet server (listening to port `23`). Outside hosts can only access the telnet server on `192.168.60.5`, not the other internal hosts.
2. Outside hosts cannot access other internal servers.
3. Internal hosts can access all the internal servers.
4. Internal hosts cannot access external servers.
5. In this task, the connection tracking mechanism is not allowed. It will be used in a later task.

You will need to use the `"-p tcp"` options to specify the match options related to the TCP protocol. You can run `"iptables -p tcp -h"` to find out all the TCP match options. The following example allows the TCP packets coming from the interface `eth0` if their source port is `5000`.

```
iptables -A FORWARD -i eth0 -p tcp --sport 5000  -j ACCEPT
```

When you are done with this task, please remember to clean the table or restart the container before moving on to the next task.

5 Task 3: Connection Tracking and Stateful Firewall

In the previous task, we have only set up stateless firewalls, which inspect each packet independently. However, packets are usually not independent; they may be part of a TCP connection, or they may be ICMP packets triggered by other packets. Treating them independently does not take into consideration the context of the packets, and can thus lead to inaccurate, unsafe, or complicated firewall rules. For example, if we would like to allow TCP packets to get into our network only if a connection was made first, we cannot achieve that easily using stateless packet filters, because when the firewall examines each individual TCP packet, it has no idea whether the packet belongs to an existing connection or not, unless the firewall maintains some state information for each connection. If it does that, it becomes a stateful firewall.

5.1 Task 3.A: Experiment with the Connection Tracking

To support stateful firewalls, we need to be able to track connections. This is achieved by the `conntrack` mechanism inside the kernel. In this task, we will conduct experiments related to this module, and get familiar with the connection tracking mechanism. In our experiment, we will check the connection tracking information on the router container. This can be done using the following command:

```
# conntrack -L
```

The goal of the task is to use a series of experiments to help students understand the connection concept in this tracking mechanism, especially for the ICMP and UDP protocols, because unlike TCP, they do not have connections. Please conduct the following experiments. For each experiment, please describe your observation, along with your explanation.

- ICMP experiment: Run the following command and check the connection tracking information on the router. Describe your observation. How long is the ICMP connection state be kept?

  ```
  // On 10.9.0.5, send out ICMP packets
  # ping 192.168.60.5
  ```

- UDP experiment: Run the following command and check the connection tracking information on the router. Describe your observation. How long is the UDP connection state be kept?

```
// On 192.168.60.5, start a netcat UDP server
# nc -lu 9090

// On 10.9.0.5, send out UDP packets
# nc -u 192.168.60.5 9090
<type something, then hit return>
```

- TCP experiment: Run the following command and check the connection tracking information on the router. Describe your observation. How long is the TCP connection state be kept?

```
// On 192.168.60.5, start a netcat TCP server
# nc -l 9090

// On 10.9.0.5, send out TCP packets
# nc 192.168.60.5 9090
<type something, then hit return>
```

5.2 Task 3.B: Setting Up a Stateful Firewall

Now we are ready to set up firewall rules based on connections. In the following example, the "`-m conntrack`" option indicates that we are using the `conntrack` module, which is a very important module for `iptables`; it tracks connections, and `iptables` replies on the tracking information to build stateful firewalls. The `--ctsate ESTABLISHED,RELATED` indicates that whether a packet belongs to an `ESTABLISHED` or `RELATED` connection. The rule allows TCP packets belonging to an existing connection to pass through.

```
iptables -A FORWARD -p tcp -m conntrack          \
         --ctstate ESTABLISHED,RELATED -j ACCEPT
```

The rule above does not cover the SYN packets, which do not belong to any established connection. Without it, we will not be able to create a connection in the first place. Therefore, we need to add a rule to accept incoming SYN packet:

```
iptables -A FORWARD -p tcp -i eth0 --dport 8080 --syn  \
         -m conntrack --ctstate NEW -j ACCEPT
```

Finally, we will set the default policy on FORWARD to drop everything. This way, if a packet is not accepted by the two rules above, they will be dropped.

```
iptables -P FORWARD DROP
```

Please rewrite the firewall rules in Task 2.C, but this time, **we will add a rule allowing internal hosts to visit any external server** (this was not allowed in Task 2.C). After you write the rules using the connection tracking mechanism, think about how to do it without using the connection tracking mechanism (you do not need to actually implement them). Based on these two sets of rules, compare these two different approaches, and explain the advantage and disadvantage of each approach. When you are done with this task, remember to clear all the rules.

6 Task 4: Limiting Network Traffic

In addition to blocking packets, we can also limit the number of packets that can pass through the firewall. This can be done using the `limit` module of `iptables`. In this task, we will use this module to limit how many packets from `10.9.0.5` are allowed to get into the internal network. You can use "`iptables -m limit -h`" to see the manual.

```
$ iptables -m limit -h
limit match options:
--limit avg             max average match rate: default 3/hour
                        [Packets per second unless followed by
                        /sec /minute /hour /day postfixes]
--limit-burst number    number to match in a burst, default 5
```

Please run the following commands on router, and then ping `192.168.60.5` from `10.9.0.5`. Describe your observation. Please conduct the experiment with and without the second rule, and then explain whether the second rule is needed or not, and why.

```
iptables -A FORWARD -s 10.9.0.5 -m limit \
         --limit 10/minute --limit-burst 5 -j ACCEPT

iptables -A FORWARD -s 10.9.0.5 -j DROP
```

7 Task 5: Load Balancing

The `iptables` is very powerful. In addition to firewalls, it has many other applications. We will not be able to cover all its applications in this lab, but we will experimenting with one of the applications, load balancing. In this task, we will use it to load balance three UDP servers running in the internal network. Let's first start the server on each of the hosts: `192.168.60.5`, `192.168.60.6`, and `192.168.60.7` (the `-k` option indicates that the server can receive UDP datagrams from multiple hosts):

```
nc -luk 8080
```

We can use the `statistic` module to achieve load balancing. You can type the following command to get its manual. You can see there are two modes: `random` and `nth`. We will conduct experiments using both of them.

```
$ iptables -m statistic -h
statistic match options:
 --mode mode         Match mode (random, nth)
 random mode:
[!] --probability p  Probability
 nth mode:
[!] --every n        Match every nth packet
 --packet p          Initial counter value (0 <= p <= n-1, default 0)
```

Using the `nth` mode (round-robin). On the router container, we set the following rule, which applies to all the UDP packets going to port `8080`. The `nth` mode of the `statistic` module is used; it implements a round-robin load balancing policy: for every three packets, pick the packet 0 (i.e., the first one), change its

destination IP address and port number to `192.168.60.5` and `8080`, respectively. The modified packets will continue on its journey.

```
iptables -t nat -A PREROUTING -p udp --dport 8080      \
         -m statistic --mode nth --every 3 --packet 0  \
         -j DNAT --to-destination 192.168.60.5:8080
```

It should be noted that those packets that do not match the rule will continue on their journeys; they will not be modified or blocked. With this rule in place, if you send a UDP packet to the router's `8080` port, you will see that one out of three packets gets to `192.168.60.5`.

```
// On 10.9.0.5
echo hello | nc -u 10.9.0.11 8080
<hit Ctrl-C>
```

Please add more rules to the router container, so all the three internal hosts get the equal number of packets. Please provide some explanation for the rules.

Using the `random` mode. Let's use a different mode to achieve the load balancing. The following rule will select a matching packet with the probability `P`. You need to replace `P` with a probability number.

```
iptables -t nat -A PREROUTING -p udp --dport 8080   \
         -m statistic --mode random --probability P \
         -j DNAT --to-destination 192.168.60.5:8080
```

Please use this mode to implement your load balancing rules, so each internal server get roughly the same amount of traffic (it may not be exactly the same, but should be close when the total number of packets is large). Please provide some explanation for the rules.

8 Submission and Demonstration

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.

Firewall

N7.1. What is `netfilter` and what are its benefits?

N7.2. What are the five `netfilter` hooks for IPv4? What are their purposes?

N7.3. Please answer the following questions about netfilter:

1. A packet from S to D arrives at D, which netfilter hooks on D will this packet pass through?
2. A packet from S to D arrives at a router R, which netfilter hooks on R will this packet pass through?
3. A packet is created on host S, and it will be sent to D. Which netfilter hooks on S will this packet pass through?

N7.4. Why do we need to build a kernel module in order to use the `netfilter` hooks?

N7.5. The following code tries to block the computer from accessing the web server (HTTP) running on host `10.0.2.5`. Please complete the code by replacing `@@@@@@@` with actual code.

```
static struct nf_hook_ops filterHook;
int setUpFilter(void){
  filterHook.hook = @@@@@@;                    ①
  filterHook.hooknum = NF_INET_POST_ROUTING;
  filterHook.pf = PF_INET;
  filterHook.priority = NF_IP_PRI_FIRST;
  nf_register_hook(&@@@@@@@);                  ②
  return 0;
}
void removeFilter(void){
  nf_unregister_hook(&@@@@@@@);                ③
}
module_init(@@@@@@@);                          ④
module_exit(@@@@@@@);                          ⑤


unsigned int block(void *priv, struct sk_buff *skb,
                 const struct nf_hook_state *state)
{
  if(!skb){
    printk(KERN_INFO,"packet receive not correct\n");
    return NF_DROP;
  }

  struct iphdr *iph;
  struct tcphdr *tcph;
```

```
  iph = ip_hdr(@@@@@@);                          ⑥
  tcph = (void *)iph+iph->ihl*4;

  __u32 sou_ip = iph->saddr;
  __u32 des_ip = iph->daddr;
  __u16 sou_port = tcph->source;
  __u16 des_port = tcph->dest;

  if(des_ip==in_aton("@@@@@@") &&                ⑦
     ntohs(des_port)== @@@@@@) {                 ⑧
    return @@@@@@;                               ⑨
  }
  return NF_ACCEPT;
}
```

N7.6. Based on the `netfilter` diagram (can be found in the book), please describe which filter is best for enforcing the following rules:

- Restricting what comes into a computer
- Restricting what goes out of a computer

N7.7. Other than being used to implement firewalls to block packets, can `netfilter` be used to modify packets? What are the other applications of `netfilter`?

N7.8. Three functions, F1, F2, and F3, are registered to the netfilter hooks. F1 is registered to `NF_INET_POST_ROUTING` with a priority `-110`. F2 is registered to `NF_INET_LOCAL_OUT` with a priority `-120`. F3 is registered to `NF_INET_LOCAL_OUT` with a priority `-100`. F4 is registered to `NF_INET_FORWARD` with a priority `-110`. When we send out an ICMP echo request packet from this machine, which functions will be invoked, and in what order?

N7.9. If a hook function returns `NF_ACCEPT` for a packet, this packet will be accepted. Is this true or false, why?

N7.10. Three functions are registered to a netfilter hook with the following order: F1 $\rightarrow$ F2 $\rightarrow$ F3. (1) If function F2 returns `NF_ACCEPT`, will function F3 be invoked or not? (2) If function F2 returns `NF_DROP`, will function F3 be invoked or not?

N7.11. Which netfilter hook do the following `iptables` chain correspond to, respectively: (1) the `filter` table's `INPUT` chain, (2) the `nat` table's `OUTPUT` chain, and the `mangle` table's `POSTROUTING` chain?

N7.12. ★
The `SYNPROXY` is a firewall to filter out SYN flooding attack packets. Please find articles from the Internet about `SYNPROXY`, and explain at high-level how it works.

N7.13. What are the benefits of stateful firewalls that support connection-based firewall rules? Please use examples to illustrate the benefit.

N7.14. In Ubuntu, a program is called `ufw`, which stands for Uncomplicated Firewall. Is this a real firewall?

N7.15. Add a rule in iptables to accept packets from a trusted network `192.168.10.0/24`

N7.16. A machine has an IP address `10.0.20.5`. On this machine, you need to block incoming connections to its ports 22, 23, 80, and 443. What will you do?

N7.17. Assuming that we have four identical UDP services running on four different machines (all listening to port 9000), and we want to distribute the load, so each machine takes one fourth of the incoming requests. How do we do this? Please provide the concrete `iptables` rules (you can use A, B, C, and D to represent the IP address of these four machines).

N7.18. ICMP and UDP do not have connections, but Linux's connection tracking does track ICMP and UDP. What do the "connections" mean for ICMP and UDP?

N7.19. When we run `conntrack -L`, we get the following results. How long will each of the connection last before it times out in the connection tracking?

```
tcp      6 431752 ESTABLISHED src=10.0.5.5 dst=52.89.15.44 ...
udp      17 1 src=10.0.5.5 dst=10.0.5.3 sport=68 dport=67 ...
icmp     1 29 src=10.0.5.5 dst=1.1.1.1 type=8 code=0 id=16 ..
```

Firewall Evasion Lab

1 Overview

There are situations where firewalls are too restrictive, making it inconvenient for users. For example, many companies and schools enforce egress filtering, which blocks users inside of their networks from reaching out to certain websites or Internet services, such as game and social network sites. There are many ways to evade firewalls. A typical approach is to use the tunneling technique, which hides the real purposes of network traffic. There are a number of ways to establish tunnels. The two most common tunneling techniques are Virtual Private Network (VPN) and port forwarding. The goal of this lab is to help students gain hands-on experience on these two tunneling techniques. The lab covers the following topics:

- Firewall evasion
- VPN
- Port forwarding
- SSH tunneling

Readings and videos. Detailed coverage of the tunneling technology and how to use it to evade firewalls can be found in the following:

- Chapter 9 of the SEED Book, *Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. See details at `https://www.handsonsecurity.net`.

Lab environment. This lab has been tested on our pre-built Ubuntu 20.04 VM, which can be downloaded from the SEED website. Since we use containers to set up the lab environment, this lab does not depend much on the SEED VM. You can do this lab using other VMs, physical machines, or VMs on the cloud.

2 Task 0: Get Familiar with the Lab Setup

We will conduct a series of experiments in this chapter. These experiments need to use several computers in two separate networks. The experiment setup is depicted in Figure 1. We use docker containers for these machines. Readers can find the container setup file from the website of this lab. In this lab, the network `10.8.0.0/24` serves as an external network, while `192.168.20.0/24` serves as the internal network.

The host `10.8.0.1` is not a container; this IP address is given to the host machine (i.e., the VM in our case). This machine is the gateway to the Internet. To reach the Internet from the hosts in both `192.168.20.0/24` and `10.8.0.0/24` networks, packets must be routed to `10.8.0.1`. The routing has already been set up.

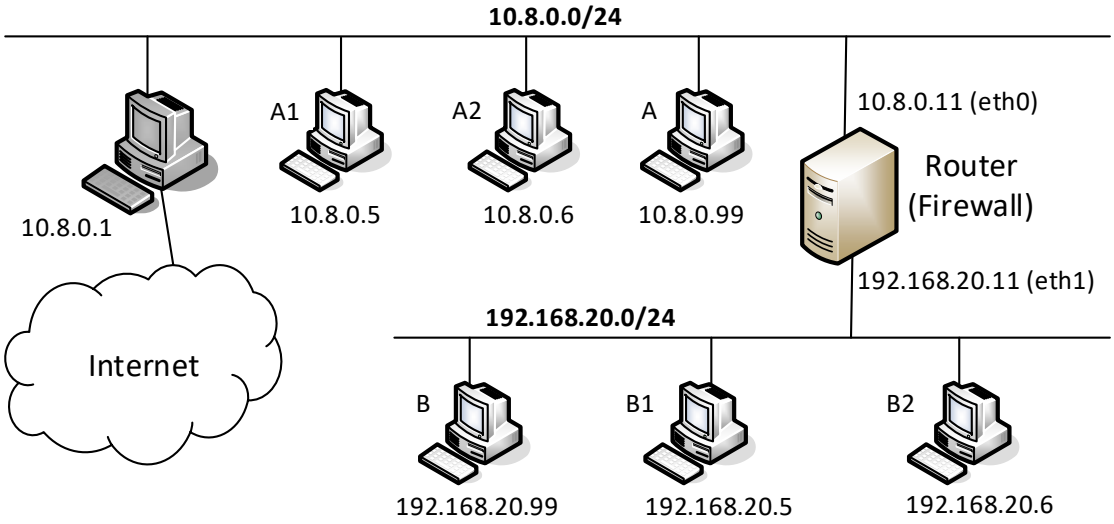


Figure 1: Network setup

Router configuration: setting up NAT. The following `iptables` command is included in the router configuration inside the `docker-compose.yml` file. This command sets up a NAT on the router for the traffic going out from its `eth0` interface, except for the packets to `10.8.0.0/24`. With this rule, for packets going out to the Internet, their source IP address will be replaced by the router's IP address `10.8.0.11`. Packets going to `10.8.0.0/24` will not go through NAT.

```
iptables -t nat -A POSTROUTING ! -d 10.8.0.0/24 -j MASQUERADE -o eth0
```

In the above command, we assume that `eth0` is the name assigned to the interface connecting the router to the `10.8.0.0/24` network. This is not guaranteed. The router has two Ethernet interfaces; when the router container is created, the name assigned to this interface might be `eth1`. You can find out the correct interface name using the following command. If the name is not `eth0`, you should make a change to the command above inside the `docker-compose.yml` file, and then restart the containers.

```
# ip -br address
lo               UNKNOWN        127.0.0.1/8
eth1@if1907      UP             192.168.20.11/24
eth0@if1909      UP             10.8.0.11/24
```

Router configuration: Firewall rules. We have also added the following firewall rules on the router. Please make sure that `eth0` is the interface connected to the `10.8.0.0/24` network and that `eth1` is the one connected to `192.168.20.0/24`. If not, make changes accordingly.

```
// Ingress filtering: only allows SSH traffic
iptables -A FORWARD -i eth0 -p tcp -m conntrack \
         --ctstate ESTABLISHED,RELATED -j ACCEPT
iptables -A FORWARD -i eth0 -p tcp --dport 22 -j ACCEPT
iptables -A FORWARD -i eth0 -p tcp -j DROP

// Egress filtering: block www.example.com
iptables -A FORWARD -i eth1 -d 93.184.216.0/24 -j DROP
```

The first rule allows TCP packets to come in if they belong to an established or related connection. This is a stateful firewall rule. The second rule allows SSH, and the third rule drops all other TCP packets if they do not satisfy the first or the second rule. The fourth rule is an egress firewall rule, and it prevents the internal hosts from sending packets to `93.184.216.0/24`, which is the network for `www.example.com`.

Lab task. Please block two more websites and add the firewall rules to the setup files. The choice of websites is up to you. We will use them in one of the tasks. Keep in mind that most popular websites have multiple IP addresses that can change from time to time. After adding the rules, start the containers, and verify that all the ingress and egress firewall rules are working as expected.

3 Task 1: Static Port Forwarding

The firewall in the lab setup prevents outside machines from connecting to any TCP server on the internal network, other than the SSH server. In this task, we would like to use static port forwarding to evade this restriction. More specifically, we will use `ssh` to create a static port forwarding tunnel between host A (on the external network) and host B (on the internal network), so whatever data received on A's port X will be sent to B, from where the data is forwarded to the target T's port Y. In the following command, we use `ssh` to create such a tunnel.

```
$ ssh -4NT -L <A's IP>:<A's port X>:<T's IP>:<T's port Y>  <user id>@<B's IP>

// -4: use IPv4 only, or we will see some error message.
// -N: do not execute a remote command.
// -T: disable pseudo-terminal allocation (save resources).
```

Regarding A's IP, typically we use `0.0.0.0`, indicating that our port forwarding will listen to the connection from all the interfaces on A. If want to limit the connection from a particular interface, we should use that interface's IP address. For example, if we want to limit the connection to the loopback interface, so only the program on the local host can use this port forwarding, we can use `127.0.0.1:<port>` or simply omit the IP address (the default IP address is `127.0.0.1`).

Lab task. Please use static port forwarding to create a tunnel between the external network and the internal network, so we can telnet into the server on `B1`. Please demonstrate that you can do such telnet from hosts `A`, `A1` and `A2`. Moreover, please answer the following questions: (1) How many TCP connections are involved in this entire process. You should run wireshark or `tcpdump` to capture the network traffic, and then point out all the involved TCP connections from the captured traffic. (2) Why can this tunnel successfully help users evade the firewall rule specified in the lab setup?

4 Task 2: Dynamic Port Forwarding

In the static port forwarding, each port-forwarding tunnel forwards the data to a particular destination. If we want to forward data to multiple destinations, we need to set up multiple tunnels. For example, using port forwarding, we can successfully visit the blocked `example.com` website, but what if the firewall blocks many other sites, how do we avoid tediously establishing one SSH tunnel for each site? We can use dynamic port forwarding to solve this problem.

In the lab setup, the router already blocks `example.com`, so hosts on the internal network cannot access the `example.com` website. Please add firewall rules to the router, so two more websites are blocked.

The choice of the websites is up to individual students. Please provide evidences to show that the websites are indeed blocked.

4.1 Task 2.1: Setting Up Dynamic Port Forwarding

We can use `ssh` to create a dynamic port-forwarding tunnel between `B` and `A`. We run the following command on host `B`. In dynamic port forwarding, `B` is often called proxy.

```
$ ssh -4NT -D  <B's IP>:<B's port X>   <user id>@<A's IP>
```

Regarding B's IP, typically we use `0.0.0.0`, indicating that our port forwarding will listen to the connection from all the interfaces on B. After the tunnel is set up, we can test it using the `curl` command. We specify a proxy option, so `curl` will send its HTTP request to the proxy B, which listens on port `X`. The proxy forwards the data received on this port to the other end of the tunnel (host A), from where the data will be further forwarded to the target website. The type of proxy is called SOCKS version 5, so that is why we specify `socks5h`.

```
$ curl --proxy socks5h://<B's IP>:<B's port>  <blocked URL>
```

Lab task. Please demonstrate that you can visit all the blocked websites using `curl` from hosts `B`, `B1`, and `B2` on the internal network. Please also answer the following questions: (1) Which computer establishes the actual connection with the intended web server? (2) How does this computer know which server it should connect to?

4.2 Task 2.2: Testing the Tunnel Using Browser

We can also test the tunnel using a real browser, instead of using `curl`. Although it is hard to run a browser inside a container, in the docker setup, by default, the host machine is always attached to any network created inside docker, and the first IP address on that network is assigned to the host machine. For example, in our setup, the host machine is the SEED VM; its IP address on the internal network `192.168.20.0/24` is `192.168.20.1`.

To use the dynamic port forwarding, we need to configure Firefox's proxy setting. To get to the setting page, we can type `about:preferences` in the URL field or click the `Preference` menu item. On the `General` page, find the `"Network Settings"` section, click the `Settings` button, and a window will pop up. Follow Figure 2 to set up the SOCKS proxy.

Lab task. Once the proxy is configured, we can then browse any website. The requests and replies will go through the SSH tunnel. Since the host VM can reach the Internet directly, to make sure that our web browsing traffic has gone through the tunnel, you should do the following: (1) run `tcpdump` on the router/-firewall, and point out the traffic involved in the entire port forwarding process. (2) Break the SSH tunnel, and then try to browse a website. Describe your observation.

Cleanup. After this task, please make sure to remove the proxy setting from Firefox by checking the `"No proxy"` option. Without a proper cleanup, future labs may be affected.

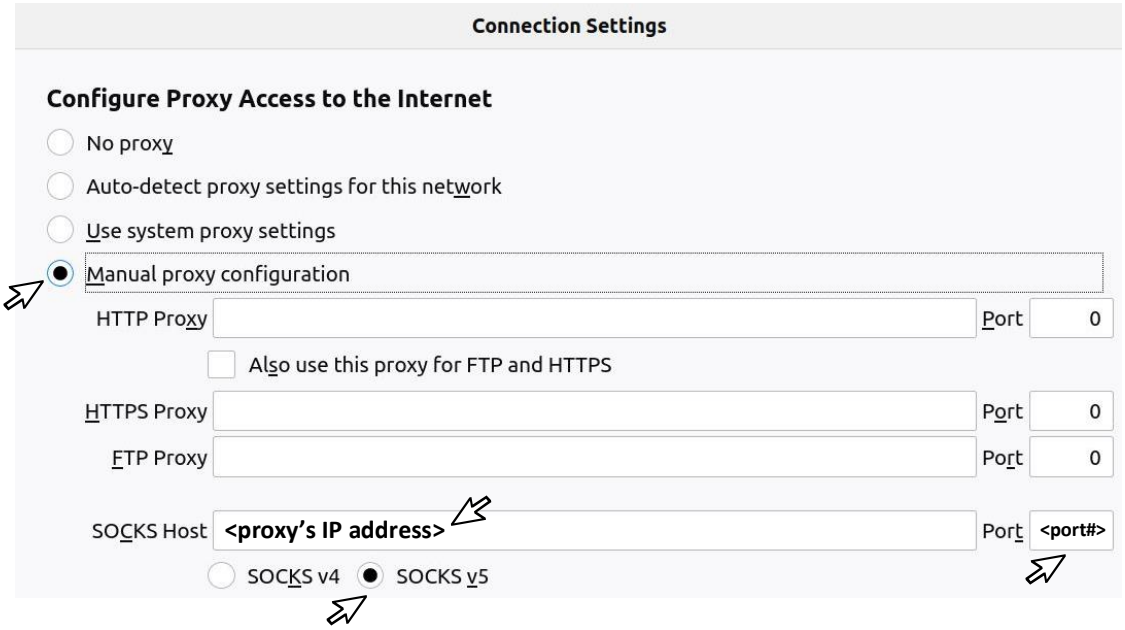


Figure 2: Configure the SOCKS Proxy

4.3 Task 2.3: Writing a SOCKS Client Using Python

For port forwarding to work, we need to specify where the data should be forwarded to (the final destination). In the static case, this piece of information is provided when we set up the tunnel, i.e., it is hard-wired into the tunnel setup. In the dynamic case, the final destination is dynamic, not specified during the setup, so how can the proxy know where to forward the data?

Applications using a dynamic port forwarding proxy must tell the proxy where to forward their data. This is done through an additional protocol between the application and the proxy. A common protocol for such a purpose is the SOCKS (Socket Secure) protocol, which becomes a de facto proxy standard.

Since the application needs to interact with the proxy using the SOCKS protocol, the application software must have a native SOCKS support in order to use SOCKS proxies. Both Firefox and `curl` have such a support, but we cannot directly use this type of proxy for the telnet program, because it does not provide a native SOCKS support. In this task, we implement a very simple SOCKS client program using Python.

```
#!/bin/env python3
import socks

s = socks.socksocket()
s.set_proxy(socks.SOCKS5, "<proxy's IP>", <proxy's port>)
s.connect(("<server's IP or hostname>", <server's port>))

hostname = "www.example.com"
req = b"GET / HTTP/1.0\r\nHost: " + hostname.encode('utf-8') + b"\r\n\r\n"
s.sendall(req)
response = s.recv(2048)
while response:
  print(response.split(b"\r\n"))
  response = s.recv(2048)
```

Lab task. Please complete this program, and use it to access `http://www.example.com` from hosts `B`, `B1`, and `B2`. The code given above is only for sending HTTP requests, not HTTPS requests (sending HTTPS requests is much more complicated due to the TLS handshake). For this task, students only need to send HTTP requests.

5 Task 3: Virtual Private Network (VPN)

VPN is often used to bypass firewall. In this task, we will use VPN to bypass ingress and egress firewalls. OpenVPN is a powerful tool that we can use, but in this task, we will simply use SSH, which is often called the poor man's VPN. We need to change some default SSH settings on the server to allow VPN creation. The changes made in `/etc/ssh/sshd_config` are listed in the following. They are already enabled inside the containers.

```
PermitRootLogin yes
PermitTunnel    yes
```

5.1 Task 3.1: Bypassing Ingress Firewall

To create a VPN tunnel from a client to a server, we run the following ssh command. This command creates a TUN interface `tun0` on the VPN client and server machines, and then connect these two TUN interfaces using an encrypted TCP connection. Both zeros in option `0:0` means `tun0`. Detailed explanation of the `-w` option can be found in the manual of SSH.

```
# ssh -w 0:0 root@<VPN Server's IP>
```

It should also be noted that creating TUN interfaces requires the root privilege, so we need to have the root privilege on both ends of the tunnel. That is why we run it inside the root account, and also SSH into the root account on the server. The above command only creates a tunnel; further configuration is needed on both ends of the tunnel. The following improved command include some of the configuration commands:

```
# ssh -w 0:0 root@<VPN Server's IP> \
      -o "PermitLocalCommand=yes" \
      -o "LocalCommand= ip addr add 192.168.53.88/24 dev tun0 && \
                        ip link set tun0 up" \
      -o "RemoteCommand=ip addr add 192.168.53.99/24 dev tun0 && \
                        ip link set tun0 up"
root@<VPN Server's IP> password: ****  ← Password:  dees
```

The `LocalCommand` entry specifies the command running on the VPN client side. It configures the client-side TUN interface: assigning the `192.168.53.88/24` address to the interface and bringing it up. The `RemoteCommand` entry specifies the command running on the VPN server side. It configures the server-side TUN interface. The configuration is incomplete, and further configuration is still needed.

Lab task. Please create a VPN tunnel between `A` and `B`, with `B` being the VPN server. Then conduct all the necessary configuration. Once everything is set up, please demonstrate that you can telnet to `B`, `B1`, and `B2` from the outside network. Please capture the packets trace, and explain why the packets are not blocked by the firewall.

5.2 Task 3.2: Bypassing Egress Firewall

In this task, we will use VPN to bypass egress firewall. In our setup, we have blocked three external websites, so the hosts on the `192.168.20.0/24` cannot access these websites. The objective of this task is to use the VPN tunneling technique to bypass these rules. This objective of is the same as that of Task 2, except that this time, we use VPN, instead of dynamic port forwarding. The command for creating VPN tunnels is similar to that in Task 3.1. In this task, we use `B` as the VPN client and `A` as the VPN server.

It should be noted that when a packet generated on the VPN client is sent to the VPN server via the tunnel, the source IP address of the packet will be `192.168.53.88` according to our setup. When this packet goes out, it will go through VirtualBox's NAT (Network Address Translation) server, where the source IP address will be replaced by the IP address of the host computer. The packet will eventually arrive at `example.com`, and the reply packet will come back to our host computer, and then be given to the same NAT server, where the destination address is translated back `192.168.53.88`. This is where the problem comes up.

VirtualBox's NAT server knows nothing about the `192.168.53.0/24` network, because this is the one that we create internally for our TUN interface, and VirtualBox has no idea how to route to this network, much less knowing that the packet should be given to VPN server. As a result, the reply packet from `example.com` will be dropped.

To solve this problem, we will set up our own NAT server on VPN server, so when packets from `192.168.53.88` go out, their source IP addresses are always replaced by the VPN server `A`'s IP address (`10.8.0.99`). We can use the following command to create a NAT server on the `eth0` interface of the VPN server.

```
# iptables -t nat -A POSTROUTING -j MASQUERADE -o eth0
```

Lab task. Please set up a VPN tunnel between `B` and `A`, with `A` being the VPN server. Please demonstrate that you can use this VPN tunnel to successfully reach the blocked websites from hosts `B`, `B1` and `B2`. Please capture the packets trace, and explain why the packets are not blocked by the firewall.

6 Task 4: Comparing SOCKS5 Proxy and VPN

Both SOCKS5 proxy (dynamic port forwarding) and VPN are commonly used in creating tunnels to bypass firewalls, as well as to protect communications. Many VPN service providers provide both types of services. Sometimes, when a VPN service provider tells you that it provides the VPN service, but in reality, it is just a SOCKS5 proxy. Although both technologies can be used to solve the same problem, they do have significant differences. Based on your experience from this lab, please compare these two technologies, describing their differences, pros and cons.

7 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.

Tunneling and Firewall Evasion

N9.1. Host H is on an internal network `10.7.2.0/24`, which is protected by a firewall (only the ssh connection to H is allowed). Alice tries to access the other hosts on the internal network. She has an account on host H, so she uses SSH to create a VPN between her outside machine and the host H. After the VPN is established, she needs to configure both outside machine and Host H. Please describe in plain English what exact configuration she needs to do to make this work.

N9.2. Why can VPN be used to bypass geo-restriction enforced by some firewalls?

N9.3. A TCP server is running on a remote machine called `sirius` using "`nc -lv 9090`". This machine is on a planet outside the Solar system. An alien named Alice living on the Earth wants to communicate with the TCP server on `sirius`, but unfortunately, the Earth has a firewall that prevents all computers on the Earth from accessing any machine outside the Solar system. Alice does have a computer on Mars, which does not have such a restrict firewall rule. Alice's computer on Mars is called `mars`, and her account name is called `alien`. (1) Please describe how Alice can use an SSH tunnel to bypass Earth's firewall, so she can talk to `sirius`. (2) Without the firewall, if Alice wants to communicate with the TCP server on `sirius`, she can use the "`nc sirius 9090`" command. Now, with the SSH tunnel and the firewall, what command should Alice run to access the server?

N9.4. This problem is based on Problem N9.3. After Alice has established an SSH tunnel between her local computer `earth` and `mars`, she can use the `nc` command to communicate with the `netcat` server on `sirius`. Please describe how the TCP packets flow, from the `netcat` client program to the destination `netcat` server.

N9.5. This problem is based on Problem N9.3. The alien Alice has many friends outside the Solar system, and she wants to connect with them by visiting various social network sites hosted on the planets where her friends live. Establishing one SSH tunnel for each social network site is tedious. Can you help Alice set up one single SSH tunnel, which she can use to stay connected with her friends?

N9.6. This problem is based on Problem N9.3. Alice also runs a web server on her machine `earth` on the Earth, and she would like her friends from her home planet to visit this web server. Unfortunately, the Earth has a firewall the prevents computers outside the Solar system from accessing any computer on the Earth. Alice would like to use her computer on Mars to set up a port forwarding using SSH, so instead of visiting Alice's machine on Earth, her friends can point their browsers to `mars`, which automatically forward the traffic to Alice's machine on the Earth. This kind of port forwarding is called *remote port forwarding*, which is not covered in the book. Please read about this kind of port forwarding from the Internet, and then describe how to use it to help Alice.

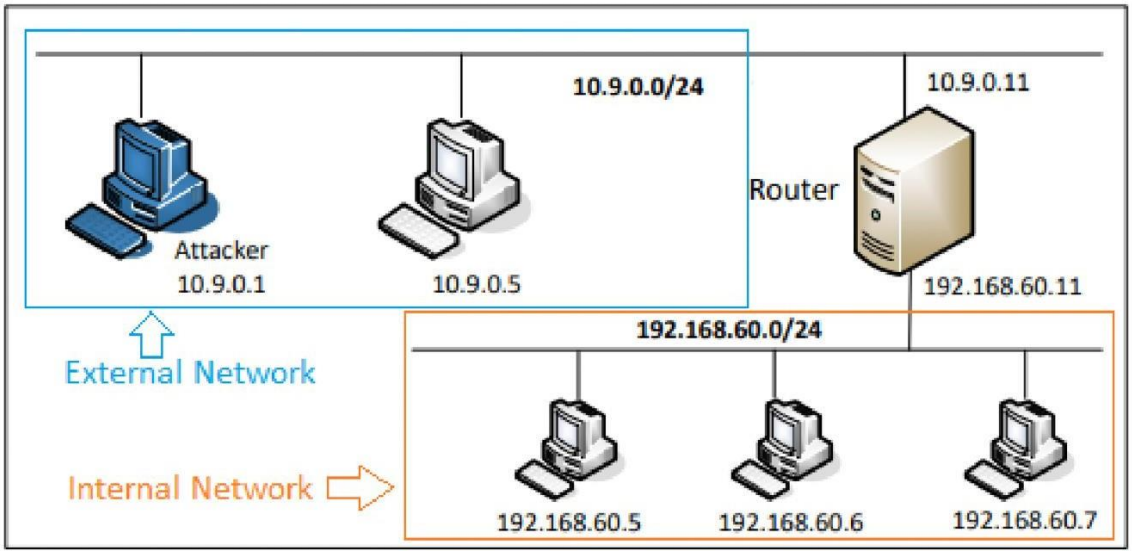
N9.7. VPN tunnel and SOCKS5-based dynamic port forwarding can both be used to create tunnels. What are their main differences?

CSE 406
Security Online – Firewall

Suppose you are the attacker (10.9.0.1) in the network below. You want to hide your identity from other users. You decided if you get at least one **PING** message and at least one **TCP SYN** packet in any order, you will drop all the packets from that IP.

You must use *netfilter* to implement this. You can't use *iptables*.

Here, is a diagram of the network given for your reference.

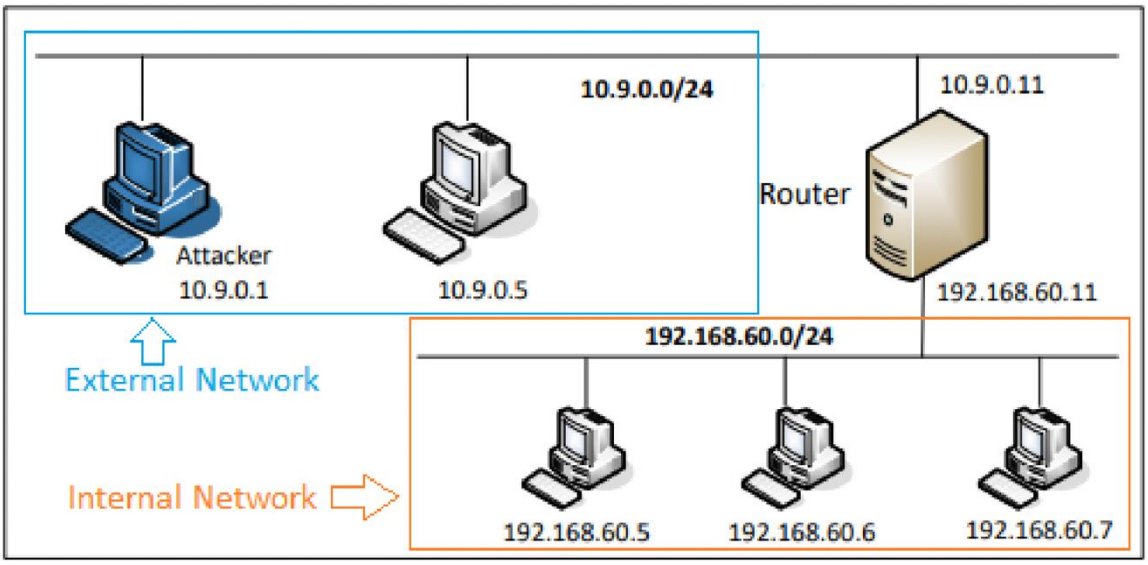


CSE 406
Security Online – Firewall

Suppose you are a network administrator. Suddenly you noticed that there is a huge number of incoming ping message to the machine with IP address with 192.168.60.5. Now, your job is to write a protection mechanism for the specified machine so that no other machine (external or internal) can send more than ***p*** ping messages to the machine.

You must use *netfilter* to implement this. You can't use *iptables*. Also, you only have the shell access for the machine with IP address with 192.168.60.5. For testing purpose, you can assume the value for *p=5.*

Here, is a diagram of the network given for your reference.



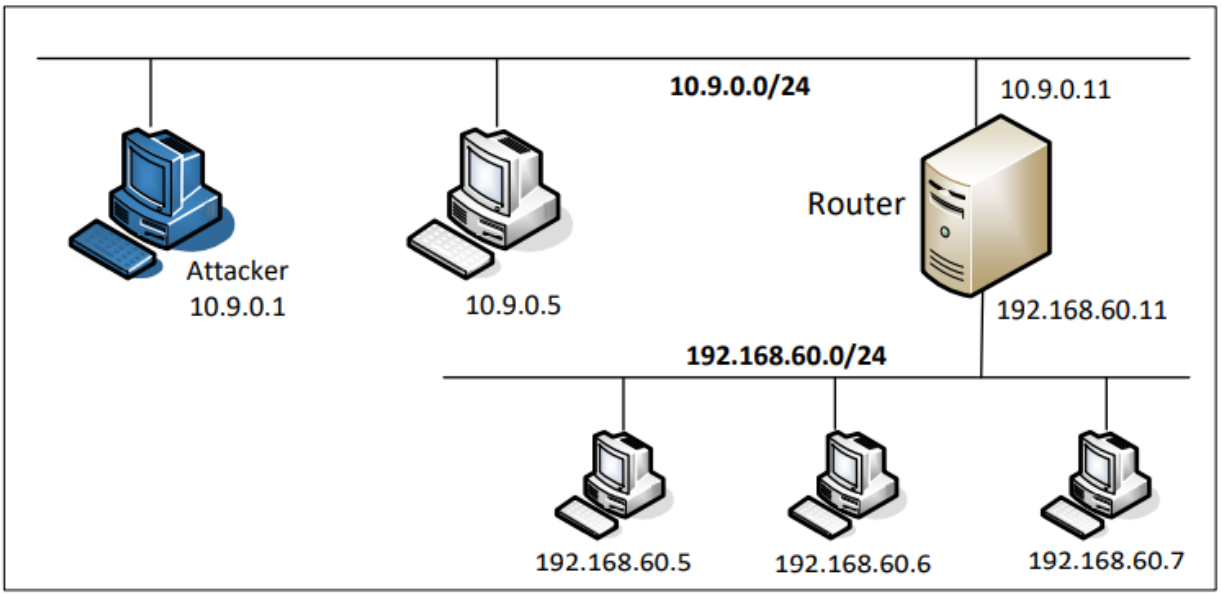
CSE - 406

Online on Firewall

A **SYN flood** is a form of denial-of-service attack (**DOS**) in which an attacker sends a succession of **SYN** requests to a target's system. This is a well known type of attack and works if a server allocates resources after receiving a **SYN**, but before it has received the **ACK**. This attack can be prevented by limiting the number of connections from a specific IP.

Now, we want to be extra cautious to prevent our internal network from such attacks. First of all, we want the hosts in the external network to communicate with the internal network servers through **telnet** connection only. Secondly, each IP address in the external network can make at most 3 connection requests and further request attempts will be dropped.

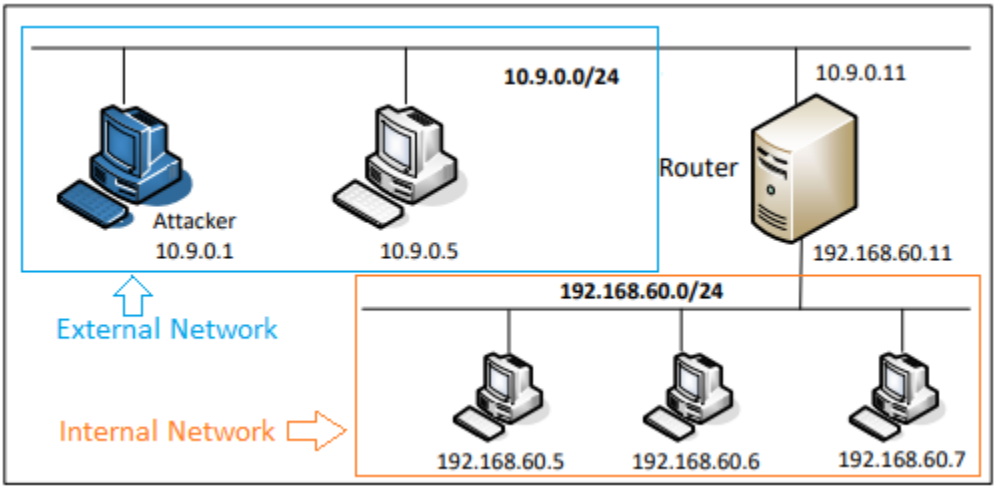
Now, you have to write down necessary firewall rules using ***netfilter*** to achieve this.



Hint: A **TCP** connection starts with a **SYN** packet i.e. a packet where the **SYN** flag is set in the **TCP** header.

CSE 406
Security Online - Firewall

Look carefully at the given topology below(the same topology as given in seed labs):



You need write the following firewall rules:

- External Hosts cannot connect via SSH to the internal host.
- No Host can connect via SSH to the router.
- Internal Hosts can connect via SSH with each other.
- Machine with IP address 10.9.0.5 can connect to 192.168.60.6 via TELNET
- No other machine can connect to any other machine via TELNET.
- Machine with IP address 10.9.0.1 can't ping any other machine
- Machine with IP address 10.9.0.5 can ping all other machines (except 10.9.0.1)
- No other machine can ping any other machine.