防火墙实验

1 概述

本实验的目标有两个：学习防火墙的工作原理，为网络设置简单的防火墙。学生将首先实现一个简单的无状态包过滤防火墙，该防火墙会检查数据包，并根据防火墙规则决定是丢弃还是转发数据包。通过这一实现任务，学生可以了解防火墙工作的基本原理。

Linux 已经有一个基于 `netfilter` 的防火墙，称为 `iptables` 。在本实验中，我们要求学生利用 `iptables` 设置防火墙规则以保护一个简单的网络。学生们还将接触到 `iptables` 的其他一些有趣的用法。本实验涵盖以下主题：

- 防火墙
- Netfilter
- 可加载内核模块 (LKM)
- 使用 `iptables` 设置防火墙规则
- `iptables` 的各种应用

阅读材料和视频。 关于防火墙的详细内容可以在以下章节中找到：

- SEED 教科书，*Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net`.
- SEED 视频，*Internet Security: A Hands-on Approach*, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net/video.html`.

实验环境。 本实验在 SEED Ubuntu 20.04 VM 中测试可行。您可以从 SEED 网站上下载我们预先构建好的镜像并在您自己的电脑上运行 SEED VM。然而，大多数 SEED 实验可以在云端进行，您可以按照我们的说明在云端创建 SEED VM。

2 使用容器设置实验环境

在本实验中，我们需要使用多个机器。其设置如图 1 所示。我们将使用容器来设置此实验环境。

2.1 容器的设置和命令

请从实验的网站下载 `Labsetup.zip` 文件到你的 VM 中，解压它，进入 `Labsetup` 文件夹，然后用 `docker-compose.yml` 文件安装实验环境。对这个文件及其包含的所有 `Dockerfile` 文件中的内容的详

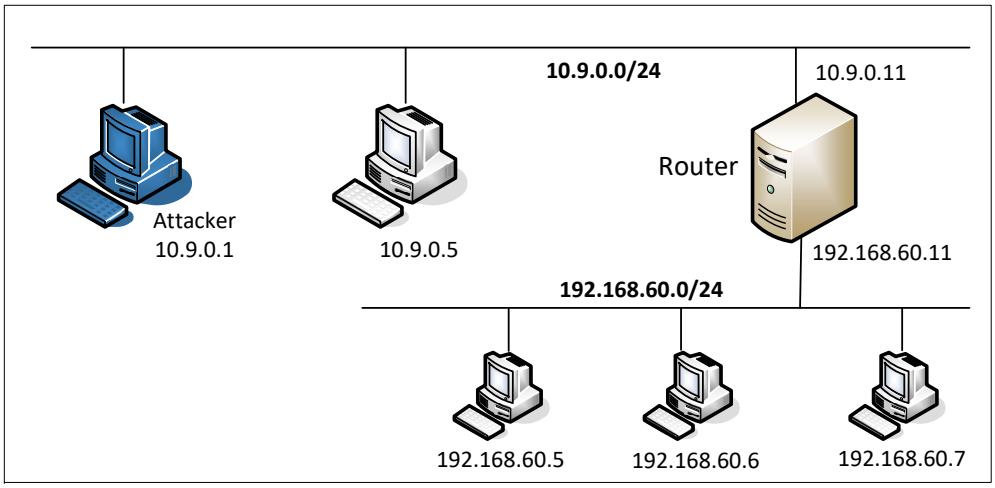

图 1: 实验环境设置

细解释都可以在链接到本实验网站的用户手册 [1] 中找到。如果这是您第一次使用容器设置 SEED 实验环境，那么阅读用户手册非常重要。

在下面，我们列出了一些与 Docker 和 Compose 相关的常用命令。由于我们将非常频繁地使用这些命令，因此我们在 `.bashrc` 文件（在我们提供的 SEED Ubuntu 20.04 虚拟机中）中为它们创建了别名。

```
$ docker-compose build  # 建立容器镜像
$ docker-compose up     # 启动容器
$ docker-compose down   # 关闭容器

// 上述 Compose 命令的别名
$ dcbuild       # docker-compose build 的别名
$ dcup          # docker-compose up 的别名
$ dcdown        # docker-compose down 的别名
```

所有容器都在后台运行。要在容器上运行命令，我们通常需要获得容器里的 Shell 。首先需要使用 `docker ps` 命令找出容器的 ID ，然后使用 `docker exec` 在该容器上启动 Shell 。我们已经在 `.bashrc` 文件中为这两个命令创建了别名。

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

// 下面的例子展示了如何在主机 C 内部得到 Shell
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7
```

[1] 如果你在部署容器的过程中发现从官方源下载容器镜像非常慢，可以参考手册中的说明使用当地的镜像服务器

```
$ docksh 96
root@9652715c8e0a:/#

// 注: 如果一条 docker 命令需要容器 ID, 你不需要
//     输入整个 ID 字符串。只要它们在所有容器当中
//     是独一无二的, 那只输入前几个字符就足够了。
```

如果你在设置实验环境时遇到问题，可以尝试从手册的 “Miscellaneous Problems” 部分中寻找解决方案。

3 任务 1：实现一个简单的防火墙

在本任务中，我们将实现一种简单的包过滤类型防火墙，它会检查每个传入和传出的数据包，并执行管理员设置的防火墙策略。由于数据包的处理是在内核中完成的，因此过滤过程也必须在内核中完成。因此，要实现这样的防火墙，似乎我们需要修改 Linux 内核。在过去，这的确需要通过修改和重新编译内核来完成。然而，现代的 Linux 操作系统提供了几种机制，使得无需重新编译内核就可以实现防火墙。这些机制包括 可加载内核模块（*Loadable Kernel Module, LKM*）和 Netfilter。

关于容器的注意事项 由于所有的容器共享相同的内核，内核模块是全局的。因此，如果从一个容器设置了内核模块，它将影响所有容器及主机。出于这个原因，在设置内核模块的位置无关紧要。在本实验中，我们将直接在主机虚拟机上设置内核模块。

另一个需要注意的问题是，容器的 IP 地址是虚拟的。发送到这些虚拟 IP 地址的数据包可能不会按照 Netfilter 文档中描述的路径进行传递。因此，在本任务中，为了避免混淆，我们尽量避免使用这些虚拟地址。我们的大多数任务将在主机虚拟机上完成，容器主要用于其他任务。

3.1 任务 1.A：实现一个简单的内核模块

LKM（可加载内核模块）允许我们在系统运行时向内核添加一个新模块。通过这个新模块，我们可以扩展内核的功能，而无需重新编译内核甚至重启计算机。防火墙的包过滤部分可以通过 LKM 实现。在本任务中，我们将熟悉 LKM 的基本用法。

以下是一个简单的可加载内核模块示例。当该模块被加载时，它会打印出 "Hello World!"。当模块从内核中移除时，它会打印出 "Bye-bye World!"。这些信息不会显示在屏幕上，而是记录在 /var/log/syslog 文件中。你可以使用 "dmesg" 命令查看这些信息。

Listing 1: hello.c（在实验设置文件中已包含）

```
#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);
```

接下来我们需要创建一个 Makefile 文件，其内容如下（文件已包含在实验设置文件中）。只需运行 make，上述程序就会被编译成一个可加载的内核模块（如果将以下内容粘贴到 Makefile 中，请确保将 make 命令前的空格替换为 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
```

生成的内核模块文件名为 hello.ko。你可以使用以下命令加载模块、列出所有模块以及移除模块。此外，可以使用 "modinfo hello.ko" 查看有关该内核模块的信息。

```
$ sudo insmod hello.ko      (插入模块)
$ lsmod | grep hello        (列出模块)
$ sudo rmmod hello          (移除模块)
$ dmesg                     (查看信息)
$ sudo dmesg                (在 Ubuntu 22.04 中，此命令需要超级用户权限)
```

任务： 请在你的虚拟机上编译此简单内核模块，并在虚拟机上运行它。在本任务中，我们不使用容器。请在实验报告中展示你的运行结果。

注意： 如果你在 Apple Silicon 机器上（通过 Ubuntu 22.04 虚拟机）编译内核模块，可能会看到一条错误消息，提示 gcc-12 未安装。可以使用以下命令安装它：

```
$ sudo apt install gcc-12
```

3.2 任务 1.B：使用 Netfilter 实现一个简单的防火墙

在本任务中，我们将通过 LKM 编写一个数据包过滤程序，并将其插入到内核中的数据包处理路径中。在 netfilter 被引入 Linux 之前，这种操作很难实现。

Netfilter 的设计目的是帮助授权用户操作数据包。它通过在 Linux 内核中实现多个钩子（hooks）

来实现这一目标。这些钩子分布在不同的位置，包括数据包的输入和输出路径。如果我们想操作输入的数据包，只需将我们的程序（通过 LKM 实现）连接到相应的钩子上。一旦有输入数据包到达，我们的程序就会被调用。通过我们的程序，可以决定是否允许数据包经过，此外，还可以修改数据包内容。

在本任务中，你需要使用 LKM 和 Netfilter 实现一个数据包过滤模块。该模块将从一个数据结构中获取防火墙策略，并使用这些策略决定是否阻止数据包。为了让学生专注于过滤部分（即防火墙的核心功能），允许学生直接把防火墙策略写在程序中（在真正的防火墙软件中，策略和代码是分开的）。关于如何使用 Netfilter ，可以参考 SEED 教科书。我们在此也提供一些指导。

挂载到 Netfilter。 使用 netfilter 非常简单。我们只需将自定义函数（位于内核模块中）挂载到相应的 netfilter 钩子上。以下示例展示了如何实现（代码位于 Labsetup/packet_filter 文件夹中，但可能与此示例略有不同）。

代码的结构与前面实现的内核模块类似。当内核模块被添加到内核中时，代码中的 registerFilter() 函数会被调用。在此函数内，我们向 netfilter 注册了两个钩子。

为了注册一个钩子，你需要准备一个钩子数据结构，并设置所有必要的参数，其中最重要的是函数名（第 ❶ 行）和钩子编号（第 ❷ 行）。钩子编号是 netfilter 中的 5 个钩子之一，指定的函数将在数据包到达该钩子时被调用。在本例中，当数据包到达 LOCAL_IN 钩子时，函数 printInfo() 会被调用（该函数稍后会给出）。当钩子数据结构准备好后，我们在第 ❸ 行将其注册到 netfilter 上。

Listing 2: 向 netfilter 注册钩子函数

```
static struct nf_hook_ops hook1, hook2;

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

   // 钩子 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

   // 钩子 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);
```

Ubuntu 20.04 虚拟机注意事项。 SEED 教科书第 2 版中的代码是在 Ubuntu 16.04 上开发的。在 Ubuntu 20.04 上运行时，需要对钩子注册和取消注册的 API 进行一些修改。以下是不同版本的差异：

```
// 钩子注册：
  nf_register_hook(&nfho);                  // 对于 Ubuntu 16.04
  nf_register_net_hook(&init_net, &nfho);   // 对于 Ubuntu 20.04

// 钩子取消注册：
  nf_unregister_hook(&nfho);                // 对于 Ubuntu 16.04
  nf_unregister_net_hook(&init_net, &nfho); // 对于 Ubuntu 20.04
```

钩子函数。 下面是一个钩子函数的示例。它仅打印出数据包的信息。当 `netfilter` 调用钩子函数时，它会将三个参数传递给该函数，其中包括一个指向实际数据包的指针（`skb`）。在下面的代码中，第 ❶ 行展示了如何从 `state` 参数中检索钩子编号。第 ❷ 行使用 `ip_hdr()` 函数获取 IP 头的指针，第 ❸ 行通过 `%pI4` 格式打印出源和目标 IP 地址。

Listing 3: 钩子函数示例

```
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;
     .. (代码省略) ...
   }

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

如果需要获取其他协议的头部信息，可以使用以下定义在头文件中的函数。这些头部的结构定义可以在 `/lib/modules/5.4.0-54-generic/build/include/uapi/linux` 文件夹中找到，其中路径中

的版本号由命令 "uname -r" 的结果决定，因此如果内核版本不同，路径也可能不同。

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

阻止数据包传输。 我们还提供了一个钩子函数的示例，展示了阻止数据包。以下示例阻止了目的 IP 为 8.8.8.8 且目的端口为 53 的 UDP 数据包，这意味着阻止了对域名服务器 8.8.8.8 的 DNS 查询。

Listing 4: 代码示例：阻止 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";

   // 将 IPv4 地址从点分十进制格式（即，字符串，如 1.2.3.4）转换为一个 32 位二进制数
   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
}
```

在上面的代码中，第 ❶ 行展示了在内核中如何将点分十进制格式的 IP 地址（即一个字符串，如 1.2.3.4）转换为 32 位的二进制（0x01020304），以便与存储在数据包中的二进制数进行比较。第 ❷ 行将目标 IP 地址和端口号与我们指定规则中的值进行比较。如果匹配，NF_DROP 将返回给 netfilter，从而丢弃数据包。否则，NF_ACCEPT 将被返回，netfilter 将允许数据包继续传输（NF_ACCEPT 仅表示数据包被当前钩子函数接受，它仍可能被其他钩子函数阻止）。

任务。 完整的示例代码名为 seedFilter.c，包含在实验设置文件的 Files/packet_filter 文件夹中。请完成以下任务（分别完成每个任务）：

1. 使用提供的 Makefile 编译代码。将结果加载到内核中，并演示防火墙是否按预期工作。你可以

使用以下命令生成到 8.8.8.8 的 UDP 数据包，即 Google 的 DNS 服务器。如果你的防火墙工作正常，请求将被阻止；否则，你将收到响应。

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

2. 将 printInfo 函数挂接到所有 netfilter 钩子上。以下是钩子编号的宏定义。通过实验结果解释每个钩子函数在何种条件下会被调用。

```
NF_INET_PRE_ROUTING
NF_INET_LOCAL_IN
NF_INET_FORWARD
NF_INET_LOCAL_OUT
NF_INET_POST_ROUTING
```

3. 实现另外两个钩子以实现以下功能：(1) 阻止其他计算机 ping 虚拟机，(2) 阻止其他计算机 telnet 到虚拟机。请实现两个不同的钩子函数，但将它们注册到相同的 netfilter 钩子上。你需要决定使用哪个钩子。Telnet 的默认端口为 TCP 端口 23。为了测试，你可以启动容器，进入 10.9.0.5，运行以下命令（10.9.0.1 是分配给虚拟机的 IP 地址。为了简化，可以将此 IP 地址写在到防火墙规则中）：

```
ping 10.9.0.1
telnet 10.9.0.1
```

重要提示。 由于我们对内核进行了修改，系统可能会崩溃。请务必经常备份文件，以免丢失数据。系统崩溃的常见原因之一是忘记取消注册的钩子。当模块被移除时，这些钩子仍然会被触发，但模块已经不存在于内核中，从而导致系统崩溃。为避免这种情况，请确保在 removeFilter 中添加相应的代码，以便在移除模块时删除所有注册的钩子函数。

4 任务 2：防火墙规则的实验

在上一任务中，我们构建了一个简单的基于 netfilter 的防火墙。实际上，Linux 已经有一个内置的防火墙，也基于 netfilter 。这个防火墙叫做 iptables 。其实该防火墙的内核部分叫做 Xtables，而 iptables 是用于配置防火墙的用户程序。然而，iptables 通常被用来同时指代内核部分和用户空间程序。

4.1 iptables 的背景

在本任务中，我们将使用 iptables 设置防火墙。除了过滤数据包外，iptables 防火墙还可以对数据包进行修改。为了管理用于不同目的的防火墙规则，iptables 使用了一个分层结构：表（table）、链（chain）和规则（rules）。表 1 列出了不同表的用途。例如，数据包过滤的规则应该放在 filter 表中，而对数据包进行修改的规则应该放在 nat 或 mangle 表中。

每个表包含若干链，每条链对应一个 netfilter 钩子。每条链表示其规则被执行的位置。例如，FORWARD 链上的规则在 NF_INET_FORWARD 钩子处被执行，INPUT 链上的规则在 NF_INET_LOCAL_IN 钩子处被执行。

每条链包含一组将被执行的防火墙规则。当我们设置防火墙时，需要将规则添加到这些链中。例如，如果我们希望拦截所有进入的 telnet 流量，可以在 filter 表的 INPUT 链中添加规则。如果我们希望将所有进入的 telnet 流量重定向到不同主机的不同端口（即端口转发），可以在 mangle 表的 INPUT 链中添加规则，因为我们需要对数据包进行修改。

表 1: iptables 表和链

表	**链**	**功能**
filter	INPUT FORWARD OUTPUT	数据包过滤
nat	PREROUTING INPUT OUTPUT POSTROUTING	修改源或目的网络地址
mangle	PREROUTING INPUT FORWARD OUTPUT POSTROUTING	数据包修改

4.2 使用 iptables

要向每个表中的链添加规则，可以使用 iptables 命令，这是一条非常强大的命令。学生可以通过输入 "man iptables" 查看 iptables 的手册，也可以在网上找到许多教程。使 iptables 显得复杂的原因在于需要提供许多命令行参数。然而，如果我们理解了这些命令行参数的结构，就会发现这个命令其实并不复杂。

在通常的 iptables 命令中，我们是向某个表中的某个链添加规则或删除规则，因此需要指定表名（默认是 filter）、链名以及对链执行的操作。之后，我们需要指定规则，用于匹配通过的数据包。如果匹配成功，将对该数据包执行一个操作。命令的一般结构如下所示：

```
iptables -t <table> -<operation> <chain>  <rule>   -j <target>
         ---------- --------------------  -------  -----------
            表              链               规则       动作
```

规则是 iptables 命令中最复杂的部分。当我们在后面使用特定的规则时会提供更详细信息。在这里我们列出了一些常用的命令：

```
// 列出某表格中的所有规则（不显示行号）
```

```
iptables -t nat -L -n

// 列出某表格中的所有规则（带有行号）
iptables -t filter -L -n --line-numbers



// 从过滤表的 INPUT 链中删除第2条规则
iptables -t filter -D INPUT 2

// 阻止所有满足<规则>的传入包
iptables -t filter -A INPUT <rule>  -j DROP
```

注意事项。 Docker 依赖 `iptables` 来管理它创建的网络，因此会向 `nat` 表中添加许多规则。当我们删除 `iptables` 规则时，小心不要删除 Docker 的规则。例如，运行 `"iptables -t nat -F"` 命令将会非常危险，因为它将删除 `nat` 表中的所有规则，包括许多 Docker 的规则。这会给 Docker 容器带来麻烦。对于 `filter` 表来说，则没有问题，因为 Docker 并不会用这个表。

4.3 任务 2.A：保护路由器

在本任务中，我们将设置规则来防止外部机器访问路由器，ping 除外。请在路由器容器上执行以下 `iptables` 命令，然后尝试从 `10.9.0.5` 访问路由器。(1) 你能 ping 通路由器吗？(2) 你能 telnet 进入路由器吗（所有容器上都运行了 telnet 服务器，并且创建了一个名为 `seed` 的账户，密码为 `dees`）。请报告你的观察结果，并解释每条规则的目的。

```
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     ← 设置 OUTPUT 的默认规则
iptables -P INPUT  DROP     ← 设置 INPUT 的默认规则
```

清理。 在继续下一个任务前，请通过运行以下命令将 `filter` 表恢复到初始状态：

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

另一种恢复所有表状态的方法是重启容器。你可以使用以下命令（首先需要找到容器的 ID）：

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

4.4 任务 2.B：保护内部网络

在本任务中，我们将在路由器上设置防火墙规则，以保护内部网络 `192.168.60.0/24`。我们需要使用 FORWARD 链来完成这一任务。

INPUT 和 OUTPUT 链中的数据包方向是明确的：数据包要么进入（INPUT），要么出去（OUTPUT）。但是，FORWARD 链不同，它是双向的：进入内部网络或从外部网络出去的所有数据包都会经过此链。为了指定方向，我们可以使用 `"-i xyz"` 选项（来自 `xyz` 接口）或 `"-o xyz"` 选项（从 `xyz` 接口出去）。你可以通过运行 `"ip addr"` 命令来得到接口名称。

在本任务中，我们希望实现一个防火墙，来保护内部网络。具体来说，我们需要对 ICMP 流量实施以下限制：

1. 外部主机不能 ping 内部主机
2. 外部主机可以 ping 路由器
3. 内部主机可以 ping 外部主机
4. 内外部网络之间的其他数据包应当被阻止

你需要使用 `"-p icmp"` 选项来设置与 ICMP 协议相关的规则。你可以运行 `"iptables -p icmp -h"` 来查看 ICMP 的规则说明。下面的例子会阻止 ICMP 回显请求。

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

在你的实验报告中，请包含你的规则和截图，以证明防火墙的配置按预期工作。完成此任务后，请记得清理规则表，或在继续下一个任务之前重启容器。

4.5 任务 2.C：保护内部服务器

在本任务中，我们将保护内部网络（`192.168.60.0/24`）中的 TCP 服务器。具体来说，我们希望实现以下目标：

1. 所有内部主机都运行了一个 telnet 服务器（监听端口 `23`）。外部主机只能访问 `192.168.60.5` 上的 telnet 服务器，不能访问其他内部主机
2. 外部主机不能访问其他内部服务器
3. 内部主机可以访问所有内部服务器
4. 内部主机不能访问外部服务器
5. 在本任务中，不允许使用连接追踪机制，它将在后续任务中使用

你可以使用 `"-p tcp"` 选项来设置与 TCP 协议相关的规则。你可以运行 `"iptables -p tcp -h"` 来查看 TCP 的规则说明。下面的例子允许来自接口 `eth0` 的源端口为 `5000` 的 TCP 数据包通过。

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

完成此任务后，请记得清理规则表，或在继续下一个任务之前重启容器。

5 任务 3：连接追踪与有状态防火墙

在上一个任务中，我们只设置了无状态防火墙，逐个检查每个数据包。然而，数据包通常不是独立的；它们可能是某个 TCP 连接的一部分，或者它们可能是由其他数据包触发的 ICMP 数据包。将数据包独立处理并不会考虑其上下文，因此可能导致不准确、不安全或复杂的防火墙规则。例如，如果我

们希望仅允许在建立连接之后才能进入网络的 TCP 数据包，使用无状态数据包过滤器是无法轻松实现的，因为当防火墙检查每个单独的 TCP 数据包时，它无法知道该数据包是否属于一个已建立的连接，除非防火墙为每个连接维护一些状态信息。如果防火墙这样做，它就变成了有状态防火墙。

5.1 任务 3.A：连接追踪实验

为了支持有状态防火墙，我们需要能够追踪连接。这是通过内核中的 conntrack 模块来实现的。在本任务中，我们将进行一些与该模块相关的实验，并熟悉连接追踪机制。在我们的实验中，我们将检查路由器容器上的连接追踪信息。可以使用以下命令进行操作：

```
# conntrack -L
```

本任务的目标是通过一系列实验帮助学生理解此追踪机制中的连接概念，特别是对于 ICMP 和 UDP 协议，因为与 TCP 协议不同，它们没有连接。请进行以下实验。每个实验后，请描述你的观察结果并解释。

- ICMP 实验：运行以下命令并检查路由器上的连接追踪信息。描述你的观察结果。ICMP 连接状态保持多久？

```
// 在 10.9.0.5 上，发送 ICMP 数据包
# ping 192.168.60.5
```

- UDP 实验：运行以下命令并检查路由器上的连接追踪信息。描述你的观察结果。UDP 连接状态保持多久？

```
// 在 192.168.60.5 上，启动一个 netcat UDP 服务器
# nc -lu 9090

// 在 10.9.0.5 上，发送 UDP 数据包
# nc -u 192.168.60.5 9090
<输入一些内容，然后回车>
```

- TCP 实验：运行以下命令并检查路由器上的连接追踪信息。描述你的观察结果。TCP 连接状态保持多久？

```
// 在 192.168.60.5 上，启动一个 netcat TCP 服务器
# nc -l 9090

// 在 10.9.0.5 上，发送 TCP 数据包
# nc 192.168.60.5 9090
<输入一些内容，然后回车>
```

5.2 任务 3.B：设置有状态防火墙

现在，我们准备根据连接设置防火墙规则。在以下示例中，"-m conntrack" 选项表示我们正在使用 conntrack 模块。这是 iptables 中非常重要的模块，它用于追踪连接，iptables 依赖于追踪信息来构建有状态防火墙。--ctstate ESTABLISHED,RELATED 表示数据包是否属于 ESTABLISHED 或 RELATED 连接。此规则允许属于现有连接的 TCP 数据包通过。

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

上面的规则不包括 SYN 数据包，因为这些数据包不属于任何已建立的连接。我们需要让创建连接的包通过。因此，我们需要添加一条规则来接受传入的 SYN 数据包：

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

最后，我们将 FORWARD 链的默认策略设置为丢弃所有数据包。这样，如果数据包未被上述两条规则接受，它们将被丢弃。

```
iptables -P FORWARD DROP
```

请重写任务 2.C 中的防火墙规则，但这次 **我们将添加一条允许内部主机访问任何外部服务器的规则**（这是任务 2.C 中未允许的）。在你使用连接追踪机制写出规则后，请思考如果不使用连接追踪机制该如何做（你不需要实际实现这些规则）。基于这两套规则，请比较这两种不同的方式，并解释每种方法的优缺点。完成此任务后，请记得清除所有规则。

6 任务 4：限制网络流量

除了阻止数据包之外，我们还可以限制通过防火墙的数据包数量。这可以使用 iptables 的 limit 模块来实现。在本任务中，我们将使用该模块来限制来自 10.9.0.5 的数据包进入内部网络的数量。你可以使用 "iptables -m limit -h" 来查看手册。

```
$ 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
```

请在路由器上运行以下命令，然后从 10.9.0.5 ping 192.168.60.5。描述你的观察结果。请分别在有和没有第二条规则的情况下进行实验，并解释是否需要第二条规则，及其原因。

```
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 任务 5：负载均衡

iptables 功能非常强大。除了作为防火墙外，它还有许多其他应用。我们无法在这个实验中覆盖所有应用，但我们将实验其中之一–负载均衡。在本任务中，我们将使用它来对运行在内部网络中的三个 UDP 服务器进行负载均衡。首先，在每个主机上启动服务器：192.168.60.5、192.168.60.6 和 192.168.60.7（-k 选项表示该服务器可以接收来自多个主机的 UDP 数据报）：

```
nc -luk 8080
```

我们可以使用 statistic 模块来实现负载均衡。你可以输入以下命令查看其手册。可以看到有两种模式：random 和 nth。我们将分别使用这两种模式进行实验。

```
$ 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)
```

使用 nth 模式（轮询）。 在路由器容器上，我们设置以下规则，适用于所有进入端口 8080 的 UDP 数据包。statistic 模块的 nth 模式被使用，它实现了轮询负载均衡策略：每三个数据包中，选择第一个数据包（即第 0 个），将其目标 IP 地址和端口号分别更改为 192.168.60.5 和 8080。修改后的数据包将继续其传输。

```
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
```

需要注意的是，未匹配此规则的数据包也将继续传输，它们不会被修改或阻止。设置此规则后，如果你发送一个 UDP 数据包到路由器的 8080 端口，你会看到每三分之一的数据包到达 192.168.60.5。

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

请在路由器容器中添加更多规则，使所有三个内部主机接收到相同数量的数据包。请为这些规则提供一些解释。

使用 random 模式。 让我们使用不同的模式来实现负载均衡。以下规则将按概率 P 选择匹配的数据包。你需要将 P 替换为一个概率值。

```
iptables -t nat -A PREROUTING -p udp --dport 8080   \
         -m statistic --mode random --probability P \
```

```
        -j DNAT --to-destination 192.168.60.5:8080
```

请使用此模式来实现你的负载均衡规则，使每个内部服务器都能获得大致相同的流量（虽然不一定完全相同，但在总包数足够大时应非常接近）。请提供一些关于这些规则的解释。

8 提交和演示

你需要提交一份带有截图的详细实验报告来描述你所做的工作和你观察到的现象。你还需要对一些有趣或令人惊讶的观察结果进行解释。请同时列出重要的代码段并附上解释。只是简单地附上代码不加以解释不会获得学分。