



山东大学
SHANDONG UNIVERSITY

Lab6: Firewall Exploration Lab

实验报告

课程名称： 网络安全

姓名： 刘舒畅（学号： 202122460175）

李 昕（学号： 202100460065）

林宗茂（学号： 202100460128）

指导老师： 郭山清

专业： 密码科学与技术

2023 年 9 月 25 日

目录

1	实验分工	1
2	实验目标	1
3	实验原理	1
3.1	iptables	1
3.1.1	基础概念	1
3.2	可加载内核模块 (LKM)	1
3.2.1	特性	2
3.2.2	LKM 的结构	2
3.3	Netfilter	2
3.3.1	钩子 (hook)	3
3.3.2	Netfilter 架构	3
3.3.3	数据包处理	3
4	实验器材	4
5	实验步骤及运行结果	4
5.1	环境搭建	4
5.2	Lab Task Set 1: 实现一个简单防火墙	5
5.2.1	Task 1A: 实现一个简单的内核模块	5
5.2.2	Task 1B: 使用 Netfilter 实现一个简单的防火墙	6
5.2.3	任务一收获和总结	15
5.3	Lab Task Set 2: 试验无状态防火墙规则	15
5.3.1	Task2.A 保护路由器	15
5.3.2	Task2.B 保护内部网络	16
5.3.3	Task2.C 保护内部服务器	18
5.3.4	任务二收获和总结	19
5.4	Lab Task Set 3: 连接跟踪和有状态防火墙	19

5.4.1	Task3.A 测试连接跟踪	20
5.4.2	Task3.B 建立有状态防火墙	22
5.4.3	任务三收获和总结	23
5.5	Lab Task Set 4: 限制网络流量	23
5.5.1	任务四收获和总结	25
5.6	Lab Task Set 5: 负载均衡	25
5.6.1	任务五收获和总结	27
A	附录	28
A.1	Task1B1 源代码	28
A.2	Task1B2 源代码	31

1 实验分工

刘舒畅：实验代码编写，SEEDLAB 实验环境操作，报告校对

李昕：SEEDLAB 实验环境操作，报告编写

林宗茂：实验原理分析，报告编写与校对

2 实验目标

Firewall Exploration Lab 的学习目标有两个：了解防火墙如何工作，并为网络建立一个简单的防火墙。

学生将实现一个简单的无状态包过滤防火墙，它可以检查数据包，并根据防火墙规则决定是丢弃还是转发数据包。通过这个实现任务，学生可以了解防火墙的基本工作原理。

3 实验原理

3.1 iptables

3.1.1 基础概念

表 (Tables) iptables 有不同的表，每个表有特定的功能。最常用的表包括 filter（用于过滤数据包）、nat（用于网络地址转换）、和 mangle（用于特殊的包处理）。

链 (Chains) 表中的链定义了规则的应用点。filter 表中有 INPUT、FORWARD 和 OUTPUT 链，nat 表中有 PREROUTING、INPUT、OUTPUT 和 POSTROUTING 链，mangle 表中有 PREROUTING、INPUT、FORWARD、OUTPUT 和 POSTROUTING 链。

规则 (Rules) 链中的规则定义了单个的过滤条件和动作。

一些常用的 iptables 操作实验手册中已经给出。

3.2 可加载内核模块 (LKM)

LKM 是 Linux 内核的一部分，允许在运行的内核中动态加载和卸载功能代码。内核模块可以提供硬件驱动、文件系统、网络协议栈或其他扩展功能，而无需重新编译内核或重启系统。这种模块化使得 Linux 内核非常灵活且可扩展。

3.2.1 特性

动态加载和卸载 LKM 可以在系统运行时动态地添加到或从内核中删除。insmod, rmmod, modprobe 是用来加载和卸载模块的常用工具。

符号解析 模块可以访问内核提供的接口和函数。并非所有内核符号（变量和函数）对模块都可见，且模块只能使用内核明确导出的符号。

参数传递 在加载时可以向模块传递参数。参数可以控制模块的行为或配置模块的某些属性。

特权级别 LKM 运行在内核空间，具有执行关键操作的权限，因此在编写模块时需要谨慎。

3.2.2 LKM 的结构

一个典型的内核模块包含几个基本部分：

初始化和退出函数

每个模块都有一个初始化函数和一个退出函数。

初始化函数在模块加载时调用，用于设置模块以及注册任何必要的处理程序或服务。

退出函数在模块卸载时被调用，用于清理和注销模块所使用的资源。

模块元数据

包含版本信息、作者、许可证等，用于描述模块的属性。

模块功能实现

实际的代码实现，包括所需的所有函数和数据结构。其中可以包括对其他内核函数的调用，如设备驱动接口或网络功能钩子。

3.3 Netfilter

Netfilter 是 Linux 内核中的一个模块框架，广泛应用于多种网络相关任务，包括但不限于包过滤、网络地址转换（NAT）、负载均衡、端口转发、日志记录和防火墙功能。Netfilter 运作在内核空间，它通过定义在不同网络栈层之间的一系列“钩子”（hooks），提供在这些层见插入处理函数（称为“处理程序”）的机制。

3.3.1 钩子 (hook)

Netfilter 提供了一系列的钩子点，允许内核模块在网络协议栈的关键位置注册回调函数。这些钩子点大致对应网络数据包的处理流程：

NF_INET_PRE_ROUTING：预路由处理，适用于路由决策前的所有数据包。

NF_INET_LOCAL_IN：本地输入，适用于目的是本机的数据包。

NF_INET_FORWARD：转发，适用于路由通过本机的数据包。

NF_INET_LOCAL_OUT：本地输出，适用于由本机产生的数据包。

NF_INET_POST_ROUTING：后路由处理，适用于即将被发送出去的所有数据包。

钩子类似于中断点，允许一个或多个处理程序进行数据包的检查和修改。

3.3.2 Netfilter 架构

Netfilter 的架构可以分为以下几个组成部分：

钩子 (Hooks)： 如上所述，这些是内核中用于监控网络流量的预定位置。

处理程序： 可以理解为附着到各钩子点的函数。每个处理程序可以根据业务逻辑检查或操作经过的数据包。

数据包选择 (Packet Selection)： 又称为“规则集”或“策略”，通过一套规则定义了针对特定类型、地址或内容的数据包应执行的操作。例如，它可以是允许、丢弃或修改数据包。

表 (Tables)： 这些表是规则集的集合，内核模块可以通过这些表定义对数据包的处理。

链 (Chains)： 链是某表中的一组规则。如 filter 表有三条默认链：INPUT、FORWARD 和 OUTPUT。

3.3.3 数据包处理

当数据包流经网络栈时，它会遇到上述的 Netfilter 钩子。每个钩子处的处理程序可以对数据包执行以下操作之一：

接受 (ACCEPT)： 允许数据包继续它的旅行。

丢弃 (DROP)： 无声地摆脱数据包。

拒绝 (REJECT)： 不接受数据包，并且可以选择发送一个错误响应给发送者。

修改 (MODIFY)： 更改数据包的某些部分，然后让它继续它的路径。

4 实验器材

名称	版本
系统	Ubuntu20.04
捕包工具	Wireshark
编程语言	python

5 实验步骤及运行结果

5.1 环境搭建

按照实验要求，打开 SEEDLAB 虚拟机，使用准备好的 docker-compose.yml 去配置虚拟机环境，输入命令启动 docker：

```
1 $ dcbuild
2 $ dcup
```

本实验的 LAN 结构如下：

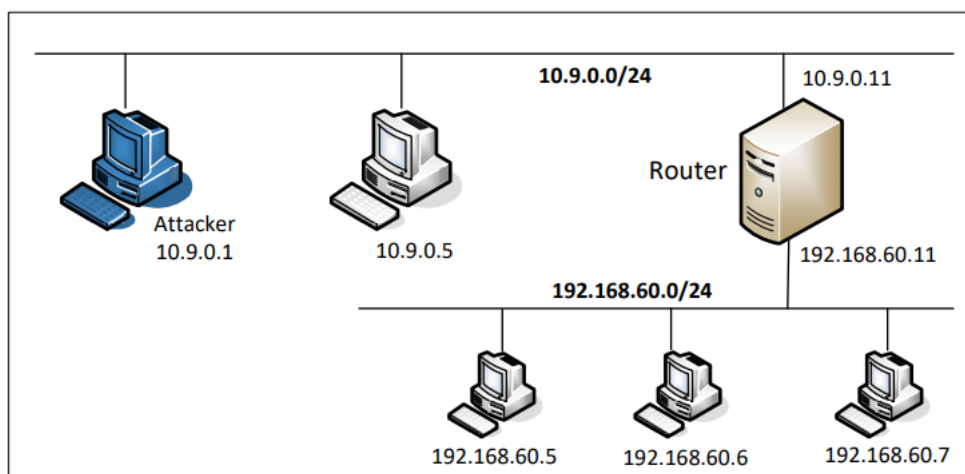


Figure 1: dockps 命令查看当前容器

使用命令 `docksh <id>` 即可进入对应容器内的主机的命令行，环境配置完毕

5.2 Lab Task Set 1: 实现一个简单防火墙

在本任务中，我们的目标是实现一个简单的包过滤防火墙。这种防火墙会检查进出的数据包，并根据管理员设定的策略执行相应的过滤操作。传统上，要在 Linux 内核中进行这样的操作，可能需要修改并重新编译内核。但现代 Linux 系统提供了一些机制，允许我们在不重建内核映像的情况下对数据包进行处理，这两种机制分别是可加载内核模块（LKM）和 Netfilter。

可加载内核模块（LKM）允许我们向内核动态添加功能，而无需重新编译整个内核。这种模块可以在运行时加载或卸载，为内核添加或移除特定的功能。由于 Linux 容器共享宿主机的内核，任何在一个容器中加载的内核模块都会影响到所有容器以及宿主机。因此，无论我们在哪个容器中设置内核模块，它都会全局生效。在本实验中，我们将选择在宿主虚拟机（VM）上设置内核模块。

Netfilter 是 Linux 内核的一个部分，它提供了一系列钩子（hooks）于网络栈的不同点，允许内核模块注册回调函数，这些函数可以在数据包通过网络栈的过程中被调用，从而实现包过滤、网络地址转换（NAT）等功能。Netfilter 是 iptables 工具背后的技术基础。

需要注意的是，容器使用的 IP 地址是虚拟化的，也就是说，发送到这些虚拟 IP 地址的数据包可能不会经过 Netfilter 文档中描述的标准路径。为了避免混淆，在本任务中，我们将尽量不使用这些虚拟地址，并且大多数操作将在宿主 VM 上执行。容器在这里主要用于辅助其他任务。

5.2.1 Task 1A: 实现一个简单的内核模块

LKM 允许我们在运行时向内核添加一个新模块。这个新模块使我们能够扩展内核，而不需要重新构建内核，甚至不需要重新启动计算机。防火墙的包过滤部分可以实现为 LKM。

进入目录 Labsetup/Files/kernel_module/，实验给出了程序 hello.c，它是一个简单的可加载内核模块，按照函数的定义与调用，当模块加载进入内核时，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!.”。信息不会打印在屏幕上，它们实际上被打印到 /var/log/syslog 文件中，我们可以使用命令 dmesg 查看消息。

在终端输入命令 make，使用 Makefile 对 hello.c 进行编译，在终端输入命令 sudo insmod hello.ko 即可将我们刚才的模块加入到内核中，继续使用命令 lsmod | grep hello 可以进一步查看 hello 是否加入了内核，成功返回则说明 hello 已经加入到了内核中，同

理，在终端输入命令 `sudo rmmod hello` 即可将我们刚才加入到内核的模块从内核中移除：

```
[12/07/23]seed@VM:~/.../kernel_module$ make
make -C /lib/modules/5.4.0-54-generic/build M=/home/seed/Desktop/lab8/Labsetup/Files/kernel_module modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-54-generic'
  CC [M]  /home/seed/Desktop/lab8/Labsetup/Files/kernel_module/hello.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/seed/Desktop/lab8/Labsetup/Files/kernel_module/hello.mod.o
  LD [M]  /home/seed/Desktop/lab8/Labsetup/Files/kernel_module/hello.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-54-generic'
[12/07/23]seed@VM:~/.../kernel_module$ sudo insmod hello.ko
[12/07/23]seed@VM:~/.../kernel_module$ lsmod | grep hello
hello                16384  0
[12/07/23]seed@VM:~/.../kernel_module$ sudo rmmod hello
[12/07/23]seed@VM:~/.../kernel_module$ dmesg
[ 0.000000] Linux version 5.4.0-54-generic (buildd@lcy01-amd64-024) (gcc
version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)) #60-Ubuntu SMP Fri Nov 6 10:37
:59 UTC 2020 (Ubuntu 5.4.0-54.60-generic 5.4.65)
[ 0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-5.4.0-54-generic root=
UUID=a91f1a43-2770-4684-9fc3-b7abfd786c1d ro quiet splash
[ 0.000000] KERNEL supported cpus:
[ 0.000000]   Intel GenuineIntel
[ 0.000000]   AMD AuthenticAMD
```

使用命令 `dmesg` 查看消息，可以看到成功输出了“Hello World!” 和 “Bye-bye World!.”：

```
[ 8776.250614] Hello World!
[ 8796.842840] Bye-bye World!.
[12/07/23]seed@VM:~/.../kernel_module$
```

5.2.2 Task 1B: 使用 Netfilter 实现一个简单的防火墙

在这个任务中，我们将包过滤程序写成一个 LKM，然后插入到内核的包处理路径中，

Tasks.1B1：使用提供的 Makefile 编译样例代码，将其加载到内核中，并演示防火墙按预期工作。

和 Task1.A 一样，Task1.B 同样自带了 Makefile 文件，其中包含了将程序 `seedFilter.c` 进行指定编译的指令，直接在终端输入命令 `make` 即可使用 Makefile 对 `seedFilter.c` 进行编译，使用 `sudo insmod` 将模块插入内核，检测到内核以及被插入：

```
[12/07/23]seed@VM:~/.../packet_filter$ make
make -C /lib/modules/5.4.0-54-generic/build M=/home/seed/Desktop/lab8/Labsetup/Files/packet_filter modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-54-generic'
  CC [M] /home/seed/Desktop/lab8/Labsetup/Files/packet_filter/seedFilter.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M] /home/seed/Desktop/lab8/Labsetup/Files/packet_filter/seedFilter.mod.o
  LD [M] /home/seed/Desktop/lab8/Labsetup/Files/packet_filter/seedFilter.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-54-generic'
[12/07/23]seed@VM:~/.../packet_filter$ sudo insmod seedFilter.ko
[12/07/23]seed@VM:~/.../packet_filter$ lsmod | grep seedFilter
seedFilter                16384  0
[12/07/23]seed@VM:~/.../packet_filter$ █
```

尝试 DNS 请求，发现连接超时没有数据包到达，说明 UDP 数据包已经被拦截：

```
[12/07/23]seed@VM:~$ dig @8.8.8.8 www.example.com
; <<>> DiG 9.16.1-Ubuntu <<>> @8.8.8.8 www.example.com
; (1 server found)
;; global options: +cmd
;; connection timed out; no servers could be reached
```

运行 `rmmod` 算法移除任务中插入内核的模块：

```
[12/07/23]seed@VM:~/.../packet_filter$ sudo rmmod seedFilter
```

再次请求 UDP 数据包：

```
[12/07/23]seed@VM:~$ dig @8.8.8.8 www.example.com

; <<>> DiG 9.16.1-Ubuntu <<>> @8.8.8.8 www.example.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 13631
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONAL: 5

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.example.com.                IN      A

;; ANSWER SECTION:
www.example.com.                49167   IN      A      93.184.216.34

;; AUTHORITY SECTION:
example.com.                    103487  IN      NS      b.iana-servers.net.
example.com.                    103487  IN      NS      a.iana-servers.net.

;; ADDITIONAL SECTION:
a.iana-servers.net.            187     IN      A      199.43.135.53
a.iana-servers.net.            1490    IN      AAAA    2001:500:8f::53
b.iana-servers.net.            99      IN      A      199.43.133.53
b.iana-servers.net.            1490    IN      AAAA    2001:500:8d::53

;; Query time: 263 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Thu Dec 07 10:33:29 EST 2023
;; MSG SIZE rcvd: 196
```

可以看到得到了回应。

Tasks.1B2：将 `printInfo` 函数挂接到所有 netfilter hook 上, 使用实验结果来帮助解释在什么条件下将调用每个 hook 函数

修改 `seedFilter.c` 中的代码，首先我们创建 5 个 hook 对象，程序原本有 2 个，因此我们还需要添加 3 个：

```
1 static struct nf_hook_ops hook1, hook2, hook3, hook4, hook5;
```

然后修改 `registerFilter()` 部分，将 `printInfo()` 函数挂载到每个 hook 上，并且设置好 `hooknum` 的值：

```
1 int registerFilter(void) {
2     printk(KERN_INFO "Registering filters.\n");
3
4     hook1.hook = printInfo;
```

```

5  hook1.hooknum = NF_INET_LOCAL_OUT;
6  hook1.pf = PF_INET;
7  hook1.priority = NF_IP_PRI_FIRST;
8  nf_register_net_hook(&init_net, &hook1);
9
10 hook2.hook = printInfo;
11 hook2.hooknum = NF_INET_LOCAL_IN;
12 hook2.pf = PF_INET;
13 hook2.priority = NF_IP_PRI_FIRST;
14 nf_register_net_hook(&init_net, &hook2);
15
16 hook3.hook = printInfo;
17 hook3.hooknum = NF_INET_PRE_ROUTING;
18 hook3.pf = PF_INET;
19 hook3.priority = NF_IP_PRI_FIRST;
20 nf_register_net_hook(&init_net, &hook3);
21
22 hook4.hook = printInfo;
23 hook4.hooknum = NF_INET_POST_ROUTING;
24 hook4.pf = PF_INET;
25 hook4.priority = NF_IP_PRI_FIRST;
26 nf_register_net_hook(&init_net, &hook4);
27
28 hook5.hook = printInfo;
29 hook5.hooknum = NF_INET_FORWARD;
30 hook5.pf = PF_INET;
31 hook5.priority = NF_IP_PRI_FIRST;
32 nf_register_net_hook(&init_net, &hook5);
33
34 return 0;
35 }

```

同理对 removeFilter() 部分进行修改，从网络堆栈中移除之前注册的网络过滤钩子：

```

1 void removeFilter(void) {
2     printk(KERN_INFO "The filters are being removed.\n");
3     nf_unregister_net_hook(&init_net, &hook1);
4     nf_unregister_net_hook(&init_net, &hook2);

```

```
5  nf_unregister_net_hook(&init_net, &hook3);
6  nf_unregister_net_hook(&init_net, &hook4);
7  nf_unregister_net_hook(&init_net, &hook5);
8 }
```

编译文件，将模块插入内核：

```
[12/07/23]seed@VM:~/.../task1B$ make
make -C /lib/modules/5.4.0-54-generic/build M=/home/seed/Desktop/lab8/Labsetup/Files/task1B modules
make[1]: Entering directory '/usr/src/linux-headers-5.4.0-54-generic'
  CC [M]  /home/seed/Desktop/lab8/Labsetup/Files/task1B/seedFilter.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC [M]  /home/seed/Desktop/lab8/Labsetup/Files/task1B/seedFilter.mod.o
  LD [M]  /home/seed/Desktop/lab8/Labsetup/Files/task1B/seedFilter.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-54-generic'
[12/07/23]seed@VM:~/.../task1B$ sudo insmod seedFilter.ko
```

请求 UDP 数据包：

```

[12/07/23]seed@VM:~/.../task1B$ dig @8.8.8.8 www.example.com

; <<>> DiG 9.16.1-Ubuntu <<>> @8.8.8.8 www.example.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 42488
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 13, ADDITIONAL: 3

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.example.com.                IN      A

;; ANSWER SECTION:
www.example.com.                 3235    IN      A      93.184.216.34

;; AUTHORITY SECTION:
com.                             151132  IN      NS      j.gtld-servers.net.
com.                             151132  IN      NS      h.gtld-servers.net.
com.                             151132  IN      NS      b.gtld-servers.net.
com.                             151132  IN      NS      g.gtld-servers.net.
com.                             151132  IN      NS      a.gtld-servers.net.
com.                             151132  IN      NS      f.gtld-servers.net.
com.                             151132  IN      NS      m.gtld-servers.net.
com.                             151132  IN      NS      k.gtld-servers.net.
com.                             151132  IN      NS      e.gtld-servers.net.
com.                             151132  IN      NS      d.gtld-servers.net.
com.                             151132  IN      NS      i.gtld-servers.net.
com.                             151132  IN      NS      c.gtld-servers.net.
com.                             151132  IN      NS      l.gtld-servers.net.

;; ADDITIONAL SECTION:
k.gtld-servers.net.             151068  IN      A      192.52.178.30
k.gtld-servers.net.             150856  IN      AAAA    2001:503:d2d::30

;; Query time: 28 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Thu Dec 07 19:34:07 EST 2023
;; MSG SIZE rcvd: 328

```

可以看到得到了回应，然后键入命令 `dmesg` 查看输出情况，可以看到激活了不同的 hook:

```
[62597.519792] Registering filters.
[62609.828171] *** LOCAL_OUT
[62609.828173] 127.0.0.1 --> 127.0.0.1 (UDP)
[62609.828183] *** POST_ROUTING
[62609.828183] 127.0.0.1 --> 127.0.0.1 (UDP)
[62609.828208] *** PRE_ROUTING
[62609.828209] 127.0.0.1 --> 127.0.0.1 (UDP)
[62609.828210] *** LOCAL_IN
[62609.828211] 127.0.0.1 --> 127.0.0.1 (UDP)
[62609.828511] *** LOCAL_OUT
[62609.828511] 10.0.2.5 --> 8.8.8.8 (UDP)
[62609.828516] *** POST_ROUTING
[62609.828516] 10.0.2.5 --> 8.8.8.8 (UDP)
[62614.826591] *** LOCAL_OUT
[62614.826593] 10.0.2.5 --> 8.8.8.8 (UDP)
[62614.826600] *** POST_ROUTING
[62614.826601] 10.0.2.5 --> 8.8.8.8 (UDP)
[62614.854831] *** PRE_ROUTING
[62614.854833] 8.8.8.8 --> 10.0.2.5 (UDP)
[62614.854841] *** LOCAL_IN
[62614.854841] 8.8.8.8 --> 10.0.2.5 (UDP)
[62624.973194] *** LOCAL_OUT
[62624.973196] 10.0.2.5 --> 224.0.0.251 (UDP)
[62624.973206] *** POST_ROUTING
[62624.973206] 10.0.2.5 --> 224.0.0.251 (UDP)
[62624.973246] *** PRE_ROUTING
[62624.973246] 10.0.2.5 --> 224.0.0.251 (UDP)
[62624.973247] *** LOCAL_IN
[62624.973248] 10.0.2.5 --> 224.0.0.251 (UDP)
[62624.973253] *** POST_ROUTING
[62624.973253] 10.0.2.5 --> 224.0.0.251 (UDP)
[62630.328999] *** LOCAL_OUT
[62630.329002] 10.0.2.5 --> 192.168.254.245 (UDP)
[62630.329011] *** POST_ROUTING
[62630.329012] 10.0.2.5 --> 192.168.254.245 (UDP)
[62630.330482] *** PRE_ROUTING
[62630.330483] 192.168.254.245 --> 10.0.2.5 (UDP)
[62630.330491] *** LOCAL_IN
[62630.330492] 192.168.254.245 --> 10.0.2.5 (UDP)
```

Tasks.1B3 : 再实现两个 hook, 以实现以下目的:(1) 防止其他计算机 ping 该虚拟机
(2) 防止其他计算机 telnet 到该虚拟机

任务三要做做的就是禁止 ping 和 telnet, 要像之前阻塞 UDP 数据包一样阻塞指定的 ICMP 数据包, 因此要设计一个 BlockICMP() 和 BlockTCP() 函数:

```

1 unsigned int BlockICMP()(void *priv, struct sk_buff *skb,
2                          const struct nf_hook_state *state)
3 {
4     struct iphdr *iph; //取出ip头
5     struct tcphdr *tcph; //取出tcp头
6     struct icmphdr *icmph;
7     iph = ip_hdr(skb);
8     tcph = tcp_hdr(skb);
9     icmph = icmp_hdr(skb);
10    unsigned char* saddr = (unsigned char*)&iph->saddr; //源IP地址
11    if (iph->protocol == IPPROTO_ICMP && icmph->type==ICMP_ECHO &&
12        (int)saddr[0] == 10 && saddr[1] == 0 && saddr[2] == 2 && saddr[3] == 6)
13    {
14        printk(KERN_INFO "丢弃ping包\n");
15        return NF_DROP;
16    }
17    return NF_ACCEPT;
18 }
19 unsigned int BlockTCP()(void *priv, struct sk_buff *skb,
20                        const struct nf_hook_state *state)
21 {
22     struct iphdr *iph; //取出ip头
23     struct tcphdr *tcph; //取出tcp头
24     iph = ip_hdr(skb);
25     tcph = tcp_hdr(skb);
26     unsigned char* saddr = (unsigned char*)&iph->saddr; //源IP地址
27     if (iph->protocol == IPPROTO_TCP && tcph->dest == htons(23) &&
28        (int)saddr[0] == 10 && saddr[1] == 0 && saddr[2] == 2 && saddr[3] == 6)
29    {
30        printk(KERN_INFO "丢弃telnet包\n");
31        return NF_DROP;
32    }
33    return NF_ACCEPT;
34 }

```

先移除上一个任务中插入内核的模块，重新编译 seedFilter.c 程序，将模块插入内核，使用另一台虚拟机 ping 本机，得不到回应：


```
root@21be5b4b393b:/# ping 10.9.0.1
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
64 bytes from 10.9.0.1: icmp_seq=1 ttl=64 time=0.063 ms
64 bytes from 10.9.0.1: icmp_seq=2 ttl=64 time=0.058 ms
^C
--- 10.9.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1011ms
rtt min/avg/max/mdev = 0.058/0.060/0.063/0.002 ms
root@21be5b4b393b:/# ping 10.9.0.1
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
^C
--- 10.9.0.1 ping statistics ---
18 packets transmitted, 0 received, 100% packet loss, time 17457ms

root@21be5b4b393b:/# telnet 10.9.0.1
Trying 10.9.0.1...
^C
```

使用命令 `dmesg` 查看输出情况，可以看到成功对 ICMP ECHO 数据包进行了拦截：

```
[ 4200.830549] Registering filters.
[ 4203.520388] *** Dropping 10.9.0.1 (ICMP)
[ 4204.529710] *** Dropping 10.9.0.1 (ICMP)
[ 4205.553943] *** Dropping 10.9.0.1 (ICMP)
[ 4206.581158] *** Dropping 10.9.0.1 (ICMP)
[ 4207.602648] *** Dropping 10.9.0.1 (ICMP)
[ 4208.625919] *** Dropping 10.9.0.1 (ICMP)
[ 4209.649800] *** Dropping 10.9.0.1 (ICMP)
[ 4210.700833] *** Dropping 10.9.0.1 (ICMP)
[ 4211.731367] *** Dropping 10.9.0.1 (ICMP)
[ 4212.754010] *** Dropping 10.9.0.1 (ICMP)
[ 4213.777734] *** Dropping 10.9.0.1 (ICMP)
[ 4214.801558] *** Dropping 10.9.0.1 (ICMP)
[ 4215.829511] *** Dropping 10.9.0.1 (ICMP)
[ 4216.879490] *** Dropping 10.9.0.1 (ICMP)
[ 4217.914344] *** Dropping 10.9.0.1 (ICMP)
[ 4218.929911] *** Dropping 10.9.0.1 (ICMP)
[ 4219.953904] *** Dropping 10.9.0.1 (ICMP)
[ 4220.977838] *** Dropping 10.9.0.1 (ICMP)
[ 4226.574306] *** Dropping 10.9.0.1 (TCP), port 23
[ 4227.602062] *** Dropping 10.9.0.1 (TCP), port 23
[ 4229.628919] *** Dropping 10.9.0.1 (TCP), port 23
[ 4233.783291] *** Dropping 10.9.0.1 (TCP), port 23
```

继续使用 TELNET 试图建立连接，连接超时，其情况与 PING 命令相同，不再赘述。

5.2.3 任务一收获和总结

在该部分实验中我们分别利用 Linux 系统中的可加载内核模块以及 netfilter 实现了简单防火墙的功能，学会了如何编写以及编译内核模块并将其加入到内核中，以及如何将自己编写的程序链接到钩子 (hook) 上，实现自定义过滤数据包规则如防止被 ping 以及防止被 telnet 链接。

5.3 Lab Task Set 2: 试验无状态防火墙规则

此任务中，我们将使用 iptables 来设置防火墙。

5.3.1 Task2.A 保护路由器

使用命令 docksh 26 进入容器 seed-router 的终端，依次键入如下命令：

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

第一条命令和第二条命令设置了对所有进入路由器的 echo-request 和 echo-reply 数据包进行放行，这相当于设置了一个白名单；第三条命令和第四条命令设置了对进出路由器的其他全部报文予以丢弃，但是不会丢弃白名单内的数据包，由此设置好防火墙：

```
root@86cd13164c7d:/# iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
root@86cd13164c7d:/# iptables -A OUTPUT -p icmp --icmp-type echo-reply -j ACCEPT
root@86cd13164c7d:/# iptables OUTPUT DROP
Bad argument 'OUTPUT'
Try 'iptables -h' or 'iptables --help' for more information.
root@86cd13164c7d:/# iptables -P OUTPUT DROP
root@86cd13164c7d:/# iptables -P INPUT DROP
root@86cd13164c7d:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.9.0.11 netmask 255.255.255.0 broadcast 10.9.0.255
    ether 02:42:0a:09:00:0b txqueuelen 0 (Ethernet)
    RX packets 78 bytes 8709 (8.7 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

尝试 ping 路由器成功，但是使用 Telnet 尝试远程连接超时：

```

root@21be5b4b393b:/# ping 10.9.0.11
PING 10.9.0.11 (10.9.0.11) 56(84) bytes of data.
64 bytes from 10.9.0.11: icmp_seq=1 ttl=64 time=0.214 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.069 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=64 time=0.062 ms
64 bytes from 10.9.0.11: icmp_seq=4 ttl=64 time=0.056 ms
^C
--- 10.9.0.11 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3049ms
rtt min/avg/max/mdev = 0.056/0.100/0.214/0.065 ms
root@21be5b4b393b:/# telnet 10.9.0.11
Trying 10.9.0.11...
^C

```

这说明我们使用 iptables 设置的防火墙取得了理想的效果。

5.3.2 Task2.B 保护内部网络

这个任务的要求是：

外网机器不能 ping 内网

外网机器能 ping 路由器

内网机器能 ping 外网机器

内网与外网之间的其他数据包需要被丢弃

首先我们使进入容器 seed-router 的终端，键入命令 ifconfig 查看网口和 ip，防止伪造包，同时用网口和 ip 判断放行：

```

root@4864766000eb:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.9.0.11 netmask 255.255.255.0 broadcast 10.9.0.255
    ether 02:42:0a:09:00:0b txqueuelen 0 (Ethernet)
    RX packets 84 bytes 9024 (9.0 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.60.11 netmask 255.255.255.0 broadcast 192.168.60.255
    ether 02:42:c0:a8:3c:0b txqueuelen 0 (Ethernet)
    RX packets 77 bytes 8121 (8.1 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

接下来我们需要使用下面三条命令，前两条命令可以实现外部主机可以 ping 通路由器和内部主机可以 ping 外部主机的功能，最后一条命令可以阻止内部网络和外部网络之间的所有其他数据包（白名单除外）：

```
root@4864766000eb:/# iptables -A FORWARD -i eth1 -p icmp --icmp-type echo-request -d 10.9.0.0/24 -j ACCEPT
root@4864766000eb:/# iptables -A FORWARD -i eth0 -p icmp --icmp-type echo-reply -s 10.9.0.0/24 -j ACCEPT
root@4864766000eb:/# iptables -P FORWARD DROP
root@4864766000eb:/#
```

测试如下，可以看到内部主机可以 ping 外部主机，外网机器能 ping 路由器，内部主机和外部主机无法通信（测试使用 telnet 包）：

```
root@c39feb9085f1:/# ping 10.9.0.5
PING 10.9.0.5 (10.9.0.5) 56(84) bytes of data.
64 bytes from 10.9.0.5: icmp_seq=1 ttl=63 time=0.080 ms
64 bytes from 10.9.0.5: icmp_seq=2 ttl=63 time=0.073 ms
64 bytes from 10.9.0.5: icmp_seq=3 ttl=63 time=0.070 ms
^C
--- 10.9.0.5 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2051ms
rtt min/avg/max/mdev = 0.070/0.074/0.080/0.004 ms
root@c39feb9085f1:/# ping 10.9.0.1
PING 10.9.0.1 (10.9.0.1) 56(84) bytes of data.
64 bytes from 10.9.0.1: icmp_seq=1 ttl=64 time=0.345 ms
64 bytes from 10.9.0.1: icmp_seq=2 ttl=64 time=0.069 ms
64 bytes from 10.9.0.1: icmp_seq=3 ttl=64 time=0.149 ms
64 bytes from 10.9.0.1: icmp_seq=4 ttl=64 time=0.086 ms
64 bytes from 10.9.0.1: icmp_seq=5 ttl=64 time=0.071 ms
^C
--- 10.9.0.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4094ms
rtt min/avg/max/mdev = 0.069/0.144/0.345/0.104 ms
root@c39feb9085f1:/# telnet 10.9.0.5
Trying 10.9.0.5...
```

同时，外网机器不能 ping 内网，内部主机和外部主机无法通信（测试使用 telnet 包）：

```
root@c89f2dcf722e:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
^C
--- 192.168.60.5 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3064ms

root@c89f2dcf722e:/# telnet 192.168.60.5
Trying 192.168.60.5...
```

证明我们设置的防火墙成功取得了预期的效果。

5.3.3 Task2.C 保护内部服务器

任务要求：

外网只能远程登陆 192.168.60.5，无法访问其他内部服务器

内部主机可以访问所有内部服务器

内部主机无法访问外部服务器

不能使用连接跟踪机制，因为它将在后面的任务中使用

使用以下三条指令，第一条命令和第二条命令设置了白名单，对来自外网的、目的地址为 192.168.60.5 且端口号为 23 的数据包和从内网发出、源地址为 192.168.60.5 且端口号为 23 的数据包进行放包操作，即实现外网只能 Telnet 到内网中的 192.168.60.5 的功能，最后对于其他数据包全部默认丢弃。：

```
root@4864766000eb:/# iptables -A FORWARD -i eth0 -p tcp --dport 23 -d 192.168.60.5 -j ACCEPT
root@4864766000eb:/# iptables -A FORWARD -i eth1 -p tcp --sport 23 -s 192.168.60.0/24 -j ACCEPT
root@4864766000eb:/# iptables -P FORWARD DROP
root@4864766000eb:/#
```

外部主机可以连接上 192.168.60.5，但是连接不上其他内部服务器：

```
root@c89f2dcf722e:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^J'.
Ubuntu 20.04.1 LTS
08f2f1f8fcf2 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.
Last login: Fri Dec  8 16:37:20 UTC 2023 on pts/1
seed@08f2f1f8fcf2:~$ exit
logout
Connection closed by foreign host.
root@c89f2dcf722e:/# telnet 192.168.60.6
Trying 192.168.60.6...
^C
root@c89f2dcf722e:/# telnet 192.168.60.7
Trying 192.168.60.7...
```

内部主机可以连接其他内部服务器，但是无法连接外部服务器：

```
root@c39feb9085f1:/# telnet 10.9.0.5
Trying 10.9.0.5...
^C
root@c39feb9085f1:/# telnet 192.168.60.7
Trying 192.168.60.7...
Connected to 192.168.60.7.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
05ebec906dc8 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.
```

即实现了保护内部服务器的防火墙。

5.3.4 任务二收获和总结

在本部分实验中，我们通过设置 iptables 实现了各种无状态防火墙的功能。通过实验我们学习到 iptables 的规则由“表 + 链 + 规则 + 行为”四个部分构成，通过对不同部分的设置达到拦截/放行各种数据包的目的，同时进一步了解了防火墙实现禁止访问、访问控制等功能的原理。

5.4 Lab Task Set 3: 连接跟踪和有状态防火墙

在前面的任务中，我们只设置了无状态防火墙，它独立地检查每个包。无论如何，数据包通常不是独立的；它们可能是 TCP 连接的一部分，也可能是由其他数据包触发的 ICMP 数据包。单独处理它们没有考虑数据包的上下文文本，因此可能导致不准确、不安全或复杂的防火墙规则。例如，如果我们希望只有在建立连接后才允许 TCP 包进入我们的网络，那么使用无状态包过滤器就不能轻松实现这一点，因为当防火墙检查每个 TCP 包时，它不知道该包是否属于现有的连接，除非防火墙为每个连接维护一些状态信息。如果它这样做，它就变成了一个有状态防火墙。

5.4.1 Task3.A 测试连接跟踪

我们先对 ICMP 报文测试连接跟踪机制，进入 hostA-10.9.0.5 容器，尝试 ping 内网主机 host1-192.168.60.5：

```
root@c89f2dcf722e:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=0.134 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.076 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.089 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=0.067 ms
^C
--- 192.168.60.5 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3051ms
rtt min/avg/max/mdev = 0.067/0.091/0.134/0.025 ms
```

进入 seed-router 容器，键入命令 conntrack -L 查看连接跟踪结果：

```
root@4864766000eb:/# conntrack -L
conntrack v1.4.5 (conntrack-tools): 0 flow entries have been shown.
root@4864766000eb:/# conntrack -L
icmp      1 29 src=10.9.0.5 dst=192.168.60.5 type=8 code=0 id=59 src=192.168.
60.5 dst=10.9.0.5 type=0 code=0 id=59 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@4864766000eb:/# conntrack -L
conntrack v1.4.5 (conntrack-tools): 0 flow entries have been shown.
```

可以看到，协议名是 icmp；右侧的 1 表示其对应应在 IP 协议中的协议号；再右方的数字表示的就是剩余时间，以秒为单位，初始为 30，随着时间不断减小直至 0，并消失在记录中。因此可知，其保存的时间为 30 秒左右。

接下来对 UDP 报文测试连接跟踪机制，我们首先在 host1-192.168.60.5 的 9090 端口建立起 UDP 的 netcat 监听，在 hostA-10.9.0.5 向 host1-192.168.60.5 发送 UDP 数据包：

```
root@c89f2dcf722e:/# nc -u 192.168.60.5 9090
hello
└─┘
seed@VM: ~
root@08f2f1f8fcf2:/# nc -lu 9090
hello
█
```

再次进入 seed-router 容器，键入命令 conntrack -L 查看连接跟踪结果：

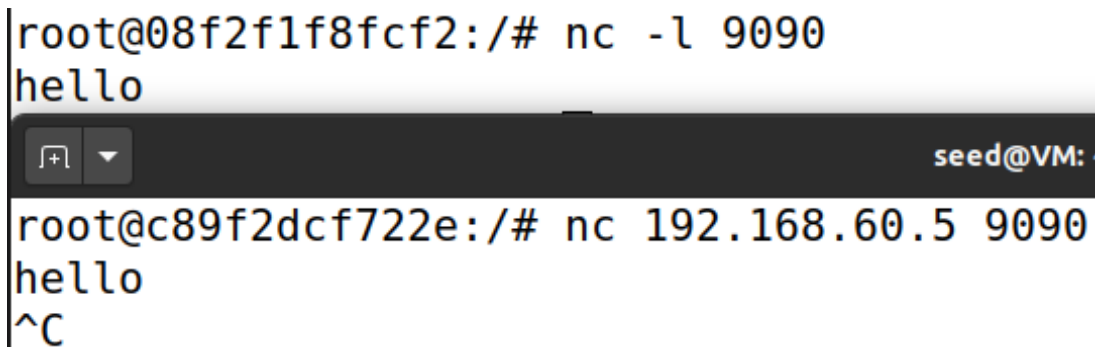
```

root@4864766000eb:/# conntrack -L
conntrack v1.4.5 (conntrack-tools): 0 flow entries have been shown.
root@4864766000eb:/# conntrack -L
conntrack v1.4.5 (conntrack-tools): 0 flow entries have been shown.
root@4864766000eb:/# conntrack -L
udp      17 29 src=10.9.0.5 dst=192.168.60.5 sport=51852 dport=9090 [UNREPLI
ED] src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=51852 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@4864766000eb:/# conntrack -L
udp      17 22 src=10.9.0.5 dst=192.168.60.5 sport=51852 dport=9090 [UNREPLI
ED] src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=51852 mark=0 use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@4864766000eb:/# conntrack -L
conntrack v1.4.5 (conntrack-tools): 0 flow entries have been shown.

```

可以看到和刚才的 ICMP 报文类似，协议名是 udp；右侧的 17 表示其对应 IP 协议中的协议号；

最后我们对 TCP 报文测试连接跟踪机制，我们首先在 host1-192.168.60.5 的 9090 端口建立起 tcp 的 netcat 监听，在 hostA-10.9.0.5 向 host1-192.168.60.5 发送 tcp 数据包：



```

root@08f2f1f8fcf2:/# nc -l 9090
hello
root@c89f2dcf722e:/# nc 192.168.60.5 9090
hello
^C

```

再次进入 seed-router 容器，键入命令 conntrack -L 查看连接跟踪结果：

```

root@4864766000eb:/# conntrack -L
conntrack v1.4.5 (conntrack-tools): 0 flow entries have been shown.
root@4864766000eb:/# conntrack -L
tcp      6 431997 ESTABLISHED src=10.9.0.5 dst=192.168.60.5 sport=37456 dpor
t=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=37456 [ASSURED] mark=0
use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@4864766000eb:/# conntrack -L
tcp      6 431997 ESTABLISHED src=10.9.0.5 dst=192.168.60.5 sport=37456 dpor
t=9090 src=192.168.60.5 dst=10.9.0.5 sport=9090 dport=37456 [ASSURED] mark=0
use=1
conntrack v1.4.5 (conntrack-tools): 1 flow entries have been shown.
root@4864766000eb:/# conntrack -L
conntrack v1.4.5 (conntrack-tools): 0 flow entries have been shown.
root@4864766000eb:/# █

```

可以看到和刚才的 ICMP 报文和 UDP 报文类似，协议名是 tcp；右侧的 6 表示 TCP 协议对应 IP 协议中的协议号；再右方的数字表示剩余时间。

5.4.2 Task3.B 建立有状态防火墙

在本任务中我们将使用 Conntrack 模块，这是一个非常重要的 iptables 模块，它跟踪连接，iptables 响应跟踪信息来构建有状态防火墙。

任务要求：

外网只能远程登陆 192.168.60.5，不能登陆其他内网主机

外网主机不能连接内网服务

内网主机可以使用其他内网主机的服务

内网主机可以连接外网服务

为了完成这些功能，根据实验指导书的提示，使用以下四条 iptables 命令：

```
root@4864766000eb:/# iptables -A FORWARD -i eth0 -p tcp --dport 23 --syn -m conntrack --ctstate NEW -d 192.168.60.5 -j ACCEPT
root@4864766000eb:/# iptables -A FORWARD -i eth1 -p tcp --sport 23 --syn -m conntrack --ctstate NEW -s 192.168.60.0/24 -j ACCEPT
root@4864766000eb:/# iptables -A FORWARD -p tcp -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT
root@4864766000eb:/# iptables -P FORWARD DROP
```

第一条指令可以实现对已经建立的 TCP 会话的消息放包；第二条命令和第三条命令中的参数 NEW 代表建立新的连接，前者保证外网的主机可以通过 Telnet 与内网的 192.168.60.5 主机建立连接，后者保证内网的主机可以自由访问外网的主机；最后的一条命令依旧是设置默认其他数据包予以丢弃。

进入 hostA-10.9.0.5 容器，它是一台外网主机，分别尝试 Telnet 内网主机 host1 (192.168.60.5)、host2 (192.168.60.6)、host3 (192.168.60.7)，只有 host1 (192.168.60.5) 连接成功，其余连接均失败：

```
root@c89f2dcf722e:/# telnet 192.168.60.7
Trying 192.168.60.7...
^C
root@c89f2dcf722e:/# telnet 192.168.60.6
Trying 192.168.60.6...
^C
root@c89f2dcf722e:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
08f2f1f8fcf2 login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage
```

进入内网主机 host1 的容器，尝试 Telnet 与外网主机 (10.9.0.5) 建立连接，成功，再尝试 Telnet 与其他内网主机 (192.168.60.6) 建立连接，也成功：

```
root@08f2f1f8fcf2:/# telnet 10.9.0.5
Trying 10.9.0.5...
^C
root@08f2f1f8fcf2:/# telnet 10.9.0.5
Trying 10.9.0.5...
Connected to 10.9.0.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
c89f2dcf722e login: seed
Password:
Welcome to Ubuntu 20.04.1 LTS (GNU/Linux 5.4.0-54-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

seed@c89f2dcf722e:~$ exit
logout
Connection closed by foreign host.
root@08f2f1f8fcf2:/# telnet 192.168.60.7
Trying 192.168.60.7...
Connected to 192.168.60.7.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
```

由此我们实现了实验要求的防火墙。

5.4.3 任务三收获和总结

本实验中我们使用 iptables 中的 conntrack 模块对 ICMP、UDP、TCP 测试了连接跟踪机制，并通过跟踪的信息来设置不同的防火墙状态。通过本实验我们熟悉了有状态防火墙的原理，知道如何利用该模块追踪连接状态并利用该状态设置防火墙规则。

5.5 Lab Task Set 4: 限制网络流量

除了拦截数据包外，我们还可以限制能通过防火墙的数据包数量，也就是控制网络流量。这可以使用 iptables 的 limit 模块来完成。在这个任务中，我们将使用这个模块

来限制允许多少来自 10.9.0.5 的数据包进入内部网络。

键入实验指导书提供的两条 iptables 命令：

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

上面的第一条命令针对源 IP 为 10.9.0.5 的数据包予以限制：平均每分钟仅可通过最多 10 个数据包，并且只允许前 5 个数据包快速通过；第二条命令实质上是对第一条命令的补充，如果第一条命令的条件不满足，则直接丢弃源 IP 为 10.9.0.5 的数据包，如此一来便可以对来自 10.9.0.5 的数据包实现流量控制。执行 ping 操作，得到以下结果：

```
|root@c89f2dcf722e:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=0.129 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.070 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.060 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=0.064 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=0.061 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=63 time=0.064 ms
64 bytes from 192.168.60.5: icmp_seq=13 ttl=63 time=0.081 ms
64 bytes from 192.168.60.5: icmp_seq=19 ttl=63 time=0.097 ms
64 bytes from 192.168.60.5: icmp_seq=25 ttl=63 time=0.063 ms
64 bytes from 192.168.60.5: icmp_seq=31 ttl=63 time=0.087 ms
^C
--- 192.168.60.5 ping statistics ---
31 packets transmitted, 10 received, 67.7419% packet loss, time 30722ms
rtt min/avg/max/mdev = 0.060/0.077/0.129/0.020 ms
```

可以看到，并不是所有数据包都得到了回应，根据序列号我们可以看出端倪：一开始前 5 个数据包成功得到了回应后，第 6、8、9、10、11、12、14 等数据包出现了连续丢失，其中断断续续才出现一个数据包成功回应，这意味着流量已经被控制，根据我们的设定只允许前 5 个数据包快速通过，之后的数据包会被限速，去掉命令的第二条，再次执行 ping 操作，得到以下结果：

```
root@c89f2dcf722e:/# ping 192.168.60.5
PING 192.168.60.5 (192.168.60.5) 56(84) bytes of data.
64 bytes from 192.168.60.5: icmp_seq=1 ttl=63 time=0.208 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=0.071 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=0.072 ms
64 bytes from 192.168.60.5: icmp_seq=4 ttl=63 time=0.062 ms
64 bytes from 192.168.60.5: icmp_seq=5 ttl=63 time=0.072 ms
64 bytes from 192.168.60.5: icmp_seq=6 ttl=63 time=0.038 ms
64 bytes from 192.168.60.5: icmp_seq=7 ttl=63 time=0.070 ms
64 bytes from 192.168.60.5: icmp_seq=8 ttl=63 time=0.072 ms
64 bytes from 192.168.60.5: icmp_seq=9 ttl=63 time=0.067 ms
64 bytes from 192.168.60.5: icmp_seq=10 ttl=63 time=0.066 ms
^C
--- 192.168.60.5 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9209ms
rtt min/avg/max/mdev = 0.038/0.079/0.208/0.043 ms
```

可以看到，没有第二行命令时，没有完成我们想要的限制网络数据传输的功能，所有数据包都快速得到了回应。

5.5.1 任务四收获和总结

本部分实验通过利用 iptables 的 limit 模块实现对来自某个源 ip 的数据包的控制。我们学习并熟悉了 iptables 的 limit 模块的使用，发现想要限制流量必须丢弃不满足限速规则的数据包，否则无法达到限制流量的目的。

5.6 Lab Task Set 5: 负载均衡

iptables 功能非常强大。除了防火墙，它还有许多其他应用。在本实验室中，我们无法涵盖它的所有应用，但我们将对其中一个应用进行实验，即负载均衡。在这个任务中，我们将使用它来负载均衡运行在内部网络中的三个 UDP 服务器。

首先我们进入服务器的容器 host1-192.168.60.5、host2-192.168.60.6、host3-192.168.60.7，使用命令 nc -luk 8080 让它们在 8080 端口保持 netcat 监听

我们先尝试使用 nth 模式，也就是 round-robin 轮转调度策略。在路由器中执行如下三条命令：

```
1 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
2 iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode nth --every 3 --
   packet 1 -j DNAT --to-destination 192.168.60.6:8080
3 iptables -t nat -A PREROUTING -p udp --dport 8080 -m statistic --mode nth --every 3 --
   packet 2 -j DNAT --to-destination 192.168.60.7:8080
```

A 附录

A.1 Task1B1 源代码

```
1 from scapy.all import *
2 #include <linux/kernel.h>
3 #include <linux/module.h>
4 #include <linux/netfilter.h>
5 #include <linux/netfilter_ipv4.h>
6 #include <linux/ip.h>
7 #include <linux/tcp.h>
8 #include <linux/udp.h>
9 #include <linux/if_ether.h>
10 #include <linux/inet.h>
11
12
13 static struct nf_hook_ops hook1, hook2, hook3, hook4, hook5;
14
15
16 unsigned int blockUDP(void *priv, struct sk_buff *skb,
17                       const struct nf_hook_state *state)
18 {
19     struct iphdr *iph;
20     struct udphdr *udph;
21
22     u16 port = 53;
23     char ip[16] = "8.8.8.8";
24     u32 ip_addr;
25
26     if (!skb) return NF_ACCEPT;
27
28     iph = ip_hdr(skb);
29     // Convert the IPv4 address from dotted decimal to 32-bit binary
30     in4_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);
31
32     if (iph->protocol == IPPROTO_UDP) {
33         udph = udp_hdr(skb);
34         if (iph->daddr == ip_addr && ntohs(udph->dest) == port){
```

```

35     printk(KERN_WARNING "**** Dropping %pI4 (UDP), port %d\n", &(iph->daddr)
    , port);
36     return NF_DROP;
37 }
38 }
39 return NF_ACCEPT;
40 }
41
42 unsigned int printInfo(void *priv, struct sk_buff *skb,
43     const struct nf_hook_state *state)
44 {
45     struct iphdr *iph;
46     char *hook;
47     char *protocol;
48
49     switch (state->hook){
50         case NF_INET_LOCAL_IN:    hook = "LOCAL_IN";    break;
51         case NF_INET_LOCAL_OUT:   hook = "LOCAL_OUT";   break;
52         case NF_INET_PRE_ROUTING: hook = "PRE_ROUTING"; break;
53         case NF_INET_POST_ROUTING: hook = "POST_ROUTING"; break;
54         case NF_INET_FORWARD:     hook = "FORWARD";     break;
55         default:                  hook = "IMPOSSIBLE";   break;
56     }
57     printk(KERN_INFO "**** %s\n", hook); // Print out the hook info
58
59     iph = ip_hdr(skb);
60     switch (iph->protocol){
61         case IPPROTO_UDP: protocol = "UDP";    break;
62         case IPPROTO_TCP: protocol = "TCP";    break;
63         case IPPROTO_ICMP: protocol = "ICMP";  break;
64         default:          protocol = "OTHER";  break;
65
66     }
67     // Print out the IP addresses and protocol
68     printk(KERN_INFO "    %pI4 --> %pI4 (%s)\n",
69         &(iph->saddr), &(iph->daddr), protocol);
70
71     return NF_ACCEPT;

```



```

72 }
73
74
75 int registerFilter(void) {
76     printk(KERN_INFO "Registering filters.\n");
77
78     hook1.hook = printInfo;
79     hook1.hooknum = NF_INET_LOCAL_OUT;
80     hook1.pf = PF_INET;
81     hook1.priority = NF_IP_PRI_FIRST;
82     nf_register_net_hook(&init_net, &hook1);
83
84     hook2.hook = printInfo;
85     hook2.hooknum = NF_INET_LOCAL_IN;
86     hook2.pf = PF_INET;
87     hook2.priority = NF_IP_PRI_FIRST;
88     nf_register_net_hook(&init_net, &hook2);
89
90     hook3.hook = printInfo;
91     hook3.hooknum = NF_INET_PRE_ROUTING;
92     hook3.pf = PF_INET;
93     hook3.priority = NF_IP_PRI_FIRST;
94     nf_register_net_hook(&init_net, &hook3);
95
96     hook4.hook = printInfo;
97     hook4.hooknum = NF_INET_POST_ROUTING;
98     hook4.pf = PF_INET;
99     hook4.priority = NF_IP_PRI_FIRST;
100    nf_register_net_hook(&init_net, &hook4);
101
102    hook5.hook = printInfo;
103    hook5.hooknum = NF_INET_FORWARD;
104    hook5.pf = PF_INET;
105    hook5.priority = NF_IP_PRI_FIRST;
106    nf_register_net_hook(&init_net, &hook5);
107
108    return 0;
109 }

```

```

110
111 void removeFilter(void) {
112     printk(KERN_INFO "The filters are being removed.\n");
113     nf_unregister_net_hook(&init_net, &hook1);
114     nf_unregister_net_hook(&init_net, &hook2);
115     nf_unregister_net_hook(&init_net, &hook3);
116     nf_unregister_net_hook(&init_net, &hook4);
117     nf_unregister_net_hook(&init_net, &hook5);
118 }
119
120 module_init(registerFilter);
121 module_exit(removeFilter);
122
123 MODULE_LICENSE("GPL");

```

A.2 Task1B2 源代码

```

1 #include <linux/kernel.h>
2 #include <linux/module.h>
3 #include <linux/netfilter.h>
4 #include <linux/netfilter_ipv4.h>
5 #include <linux/ip.h>
6 #include <linux/tcp.h>
7 #include <linux/icmp.h>
8 #include <linux/if_ether.h>
9 #include <linux/inet.h>
10
11
12 static struct nf_hook_ops hook1, hook2;
13
14
15 unsigned int blockICMP(void *priv, struct sk_buff *skb,
16                        const struct nf_hook_state *state)
17 {
18     struct iphdr *iph;
19     struct icmphdr *icmph;

```

```

20
21 char ip[16] = "10.9.0.1";
22 u32 ip_addr;
23
24 if (!skb) return NF_ACCEPT;
25
26 iph = ip_hdr(skb);
27 // Convert the IPv4 address from dotted decimal to 32-bit binary
28 in4_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);
29
30 if (iph->protocol == IPPROTO_ICMP) {
31     icmph = icmp_hdr(skb);
32     if (iph->daddr == ip_addr && icmph->type==ICMP_ECHO){
33         printk(KERN_WARNING "*** Dropping %pI4 (ICMP)\n", &(iph->daddr));
34         return NF_DROP;
35     }
36 }
37 return NF_ACCEPT;
38 }
39
40 unsigned int blockTCP(void *priv, struct sk_buff *skb,
41                      const struct nf_hook_state *state)
42 {
43     struct iphdr *iph;
44     struct tcphdr *tcph;
45
46     u16 port = 23;
47     char ip[16] = "10.9.0.1";
48     u32 ip_addr;
49
50     if (!skb) return NF_ACCEPT;
51
52     iph = ip_hdr(skb);
53     // Convert the IPv4 address from dotted decimal to 32-bit binary
54     in4_pton(ip, -1, (u8 *)&ip_addr, '\0', NULL);
55
56     if (iph->protocol == IPPROTO_TCP) {
57         tcph = tcp_hdr(skb);

```

```

58     if (iph->daddr == ip_addr && ntohs(tcph->dest) == port){
59         printk(KERN_WARNING "*** Dropping %pI4 (TCP), port %d\n", &(iph->daddr)
        , port);
60         return NF_DROP;
61     }
62 }
63 return NF_ACCEPT;
64 }
65
66
67 unsigned int printInfo(void *priv, struct sk_buff *skb,
68                       const struct nf_hook_state *state)
69 {
70     struct iphdr *iph;
71     char *hook;
72     char *protocol;
73
74     switch (state->hook){
75         case NF_INET_LOCAL_IN:    hook = "LOCAL_IN";    break;
76         case NF_INET_LOCAL_OUT:   hook = "LOCAL_OUT";   break;
77         case NF_INET_PRE_ROUTING: hook = "PRE_ROUTING"; break;
78         case NF_INET_POST_ROUTING: hook = "POST_ROUTING"; break;
79         case NF_INET_FORWARD:     hook = "FORWARD";     break;
80         default:                  hook = "IMPOSSIBLE";   break;
81     }
82     printk(KERN_INFO "*** %s\n", hook); // Print out the hook info
83
84     iph = ip_hdr(skb);
85     switch (iph->protocol){
86         case IPPROTO_UDP: protocol = "UDP"; break;
87         case IPPROTO_TCP: protocol = "TCP"; break;
88         case IPPROTO_ICMP: protocol = "ICMP"; break;
89         default:          protocol = "OTHER"; break;
90
91     }
92     // Print out the IP addresses and protocol
93     printk(KERN_INFO "    %pI4 --> %pI4 (%s)\n",
94             &(iph->saddr), &(iph->daddr), protocol);

```

```

95
96  return NF_ACCEPT;
97  }
98
99
100 int registerFilter(void) {
101     printk(KERN_INFO "Registering filters.\n");
102
103     hook1.hook = blockICMP;
104     hook1.hooknum = NF_INET_PRE_ROUTING;
105     hook1.pf = PF_INET;
106     hook1.priority = NF_IP_PRI_FIRST;
107     nf_register_net_hook(&init_net, &hook1);
108
109     hook2.hook = blockTCP;
110     hook2.hooknum = NF_INET_PRE_ROUTING;
111     hook2.pf = PF_INET;
112     hook2.priority = NF_IP_PRI_FIRST;
113     nf_register_net_hook(&init_net, &hook2);
114
115     return 0;
116 }
117
118 void removeFilter(void) {
119     printk(KERN_INFO "The filters are being removed.\n");
120     nf_unregister_net_hook(&init_net, &hook1);
121     nf_unregister_net_hook(&init_net, &hook2);
122 }
123
124 module_init(registerFilter);
125 module_exit(removeFilter);
126
127 MODULE_LICENSE("GPL");

```