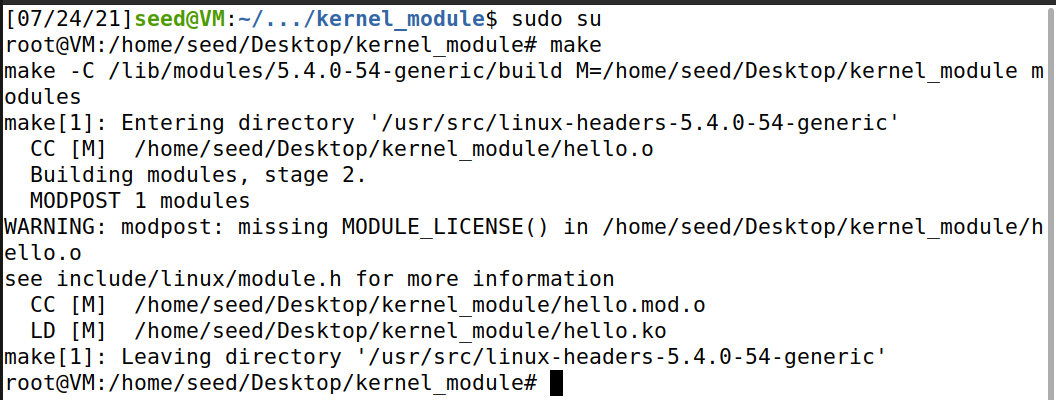
**TASK 1**

**1.A**

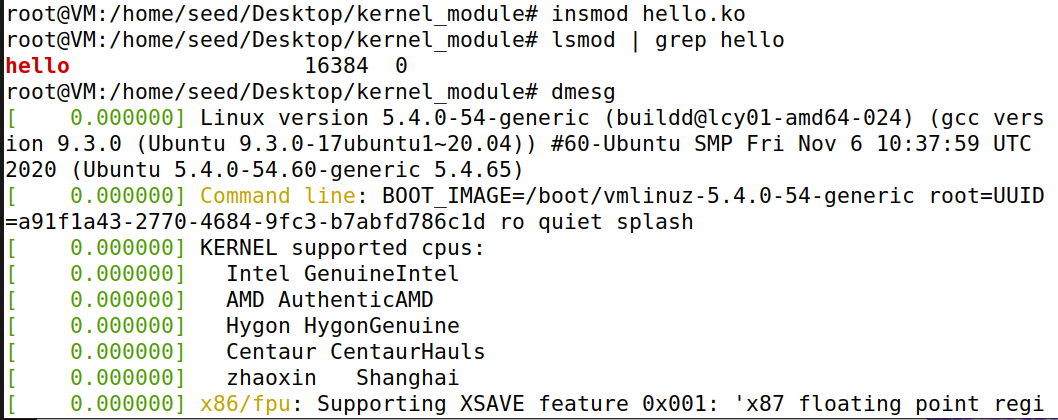
输入make编译

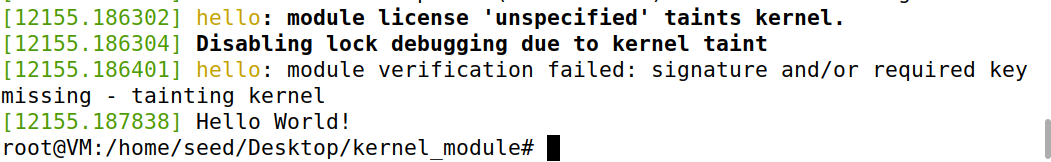
结果如下



加载该内核模块

然后列出该模块的信息，打印日志



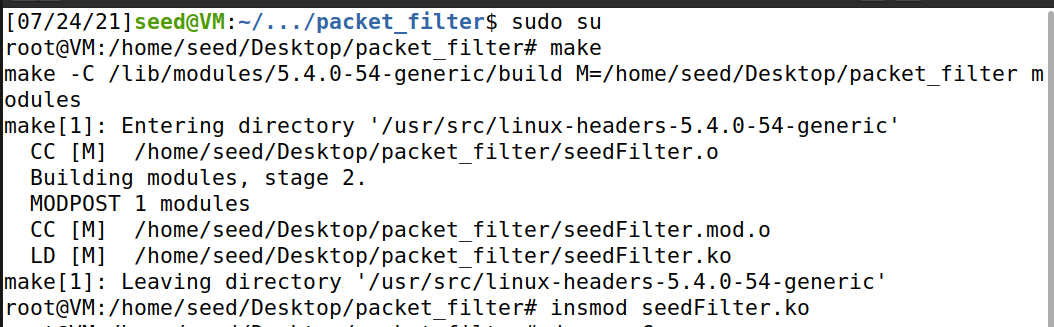


成功打印出Hello World! 查看提示的报错信息是因为内核模块未进行签名，但打印仍是正常的

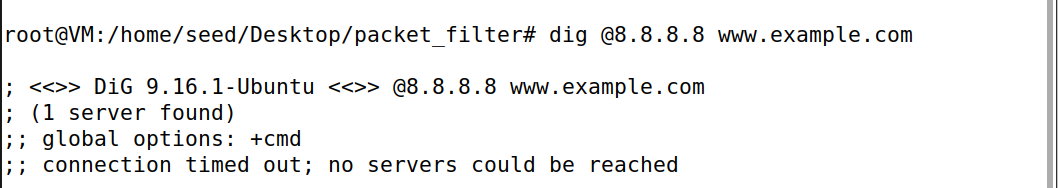
**1.B**

**task 1**

将packet\_filter文件夹复制到桌面，进入文件夹运行make命令编译内核模块，然后进行加载，过程如下



输入命令测试，无法连接到域名服务器



task 2

将printInfo函数挂接到所有的netfilter hook上

所给代码修改部分如下

static struct nf\_hook\_ops hook1, hook2, hook3, hook4, hook5;

int registerFilter(void) {

printk(KERN\_INFO "Registering filters.\n");

hook1.hook = printInfo;

hook1.hooknum = NF\_INET\_PRE\_ROUTING;

hook1.pf = PF\_INET;

hook1.priority = NF\_IP\_PRI\_FIRST;

nf\_register\_net\_hook(&init\_net, &hook1);

hook2.hook = printInfo;

hook2.hooknum = NF\_INET\_LOCAL\_IN;

hook2.pf = PF\_INET;

hook2.priority = NF\_IP\_PRI\_FIRST;

nf\_register\_net\_hook(&init\_net, &hook2);

hook3.hook = printInfo;

hook3.hooknum = NF\_INET\_FORWARD;

hook3.pf = PF\_INET;

hook3.priority = NF\_IP\_PRI\_FIRST;

nf\_register\_net\_hook(&init\_net, &hook3);

hook4.hook = printInfo;

hook4.hooknum = NF\_INET\_LOCAL\_OUT;

hook4.pf = PF\_INET;

hook4.priority = NF\_IP\_PRI\_FIRST;

nf\_register\_net\_hook(&init\_net, &hook4);

hook5.hook = printInfo;

hook5.hooknum = NF\_INET\_POST\_ROUTING;

hook5.pf = PF\_INET;

hook5.priority = NF\_IP\_PRI\_FIRST;

nf\_register\_net\_hook(&init\_net, &hook5);

return 0;

}

void removeFilter(void) {

printk(KERN\_INFO "The filters are being removed.\n");

nf\_unregister\_net\_hook(&init\_net, &hook1);

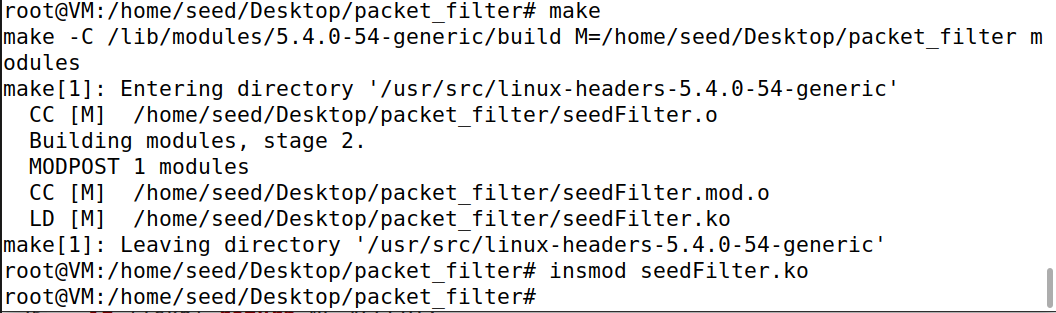
nf\_unregister\_net\_hook(&init\_net, &hook2);

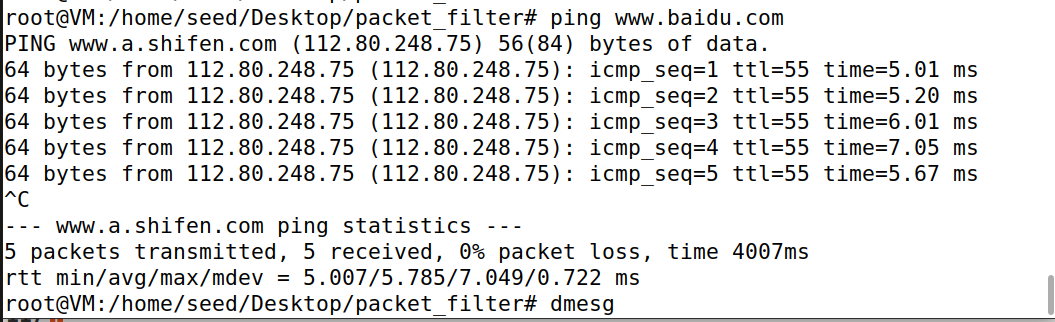
nf\_unregister\_net\_hook(&init\_net, &hook3);

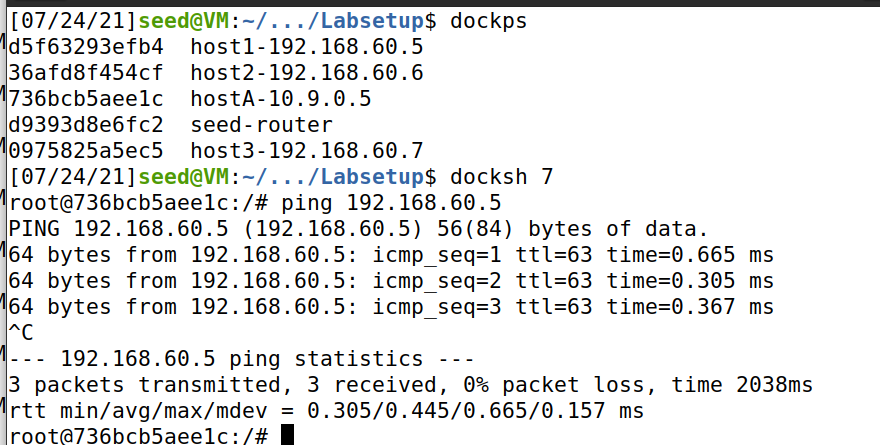
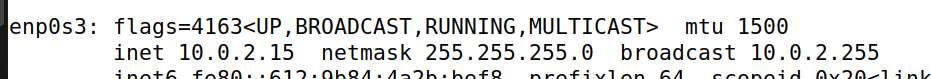
nf\_unregister\_net\_hook(&init\_net, &hook4);

nf\_unregister\_net\_hook(&init\_net, &hook5);

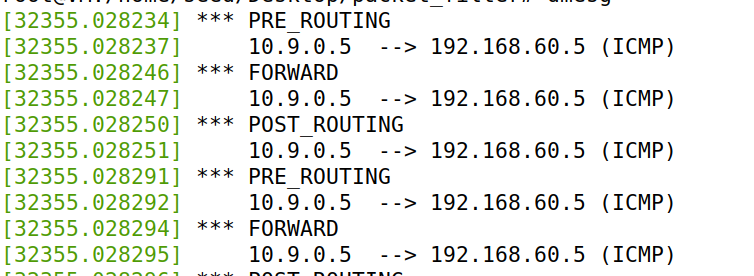
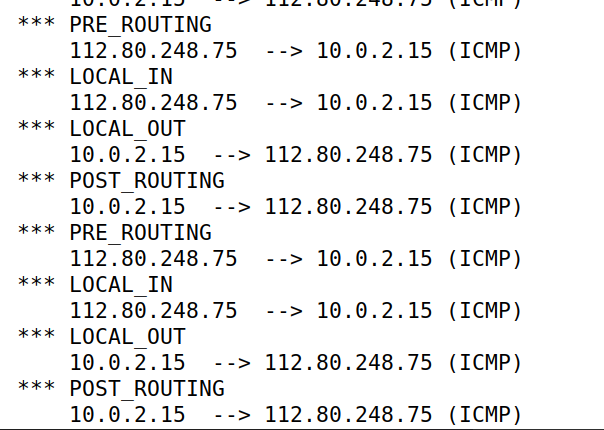
}

编译内核模块并加载

分别在虚拟主机和hostA上进行测试



得到的结果如下



根据实验结果可知，NF\_IP\_PRE\_ROUTING在数据包刚进入主机进行处理的时候调用；NF\_IP\_LOCAL\_IN在确认数据包的目的地址为本机的时候调用；NF\_IP\_FORWARD在要数据包通过主机进行转发的时候调用；NF\_IP\_LOCAL\_OUT在确认数据包的源地址为本机的时候调用；NF\_IP\_POST\_ROUTING在数据包将离开主机进行处理的时候调用。

task 3

所给代码的修改部分如下

unsigned int ICMPFilter(void \*priv, struct sk\_buff \*skb, const struct nf\_hook\_state \*state)

{

struct iphdr \*iph;

iph = ip\_hdr(skb);

if (iph->protocol == IPPROTO\_ICMP) {

printk(KERN\_INFO "Dropping ICMP packet:%pI4\n", &(iph->saddr));

return NF\_DROP;

}

return NF\_ACCEPT;

}

unsigned int telnetFilter(void \*priv, struct sk\_buff \*skb, const struct nf\_hook\_state \*state)

{

struct iphdr \*iph;

struct tcphdr \*tcph;

iph = ip\_hdr(skb);

tcph = (void \*)iph + iph->ihl \* 4;

if (iph->protocol == IPPROTO\_TCP && tcph->dest == htons(23)) {

printk(KERN\_INFO "Dropping telnet packet:%pI4\n", &(iph->saddr));

return NF\_DROP;

}

return NF\_ACCEPT;

}

int registerFilter(void) {

printk(KERN\_INFO "Registering filters.\n");

hook1.hook = ICMPFilter;

hook1.hooknum = NF\_INET\_LOCAL\_IN;

hook1.pf = PF\_INET;

hook1.priority = NF\_IP\_PRI\_FIRST;

nf\_register\_net\_hook(&init\_net, &hook1);

hook2.hook = telnetFilter;

hook2.hooknum = NF\_INET\_LOCAL\_IN;

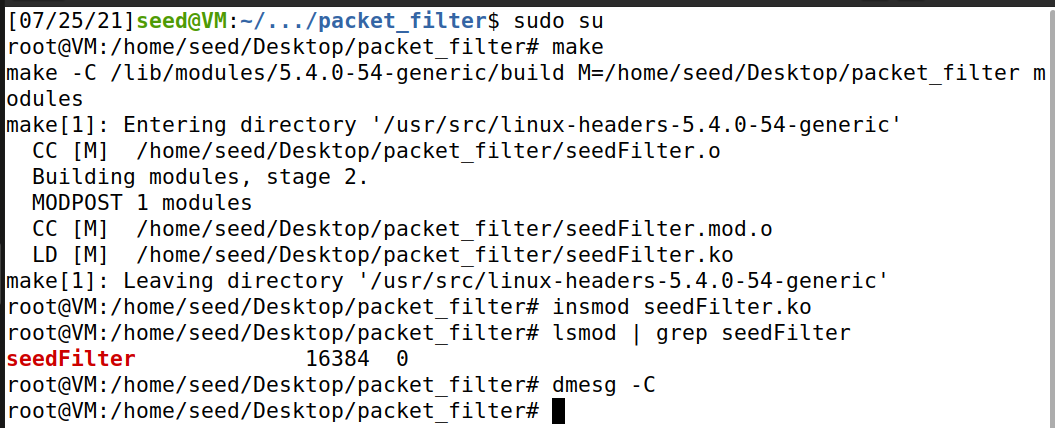
hook2.pf = PF\_INET;

hook2.priority = NF\_IP\_PRI\_FIRST;

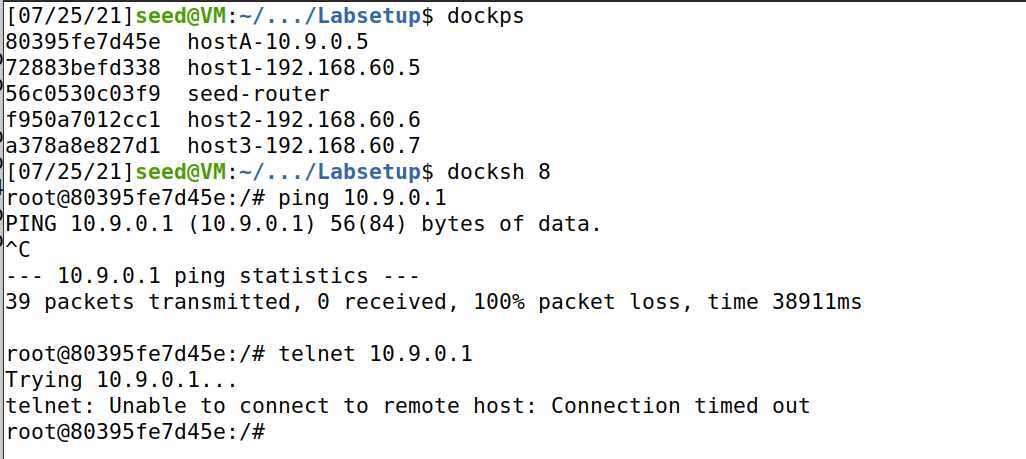
nf\_register\_net\_hook(&init\_net, &hook2);

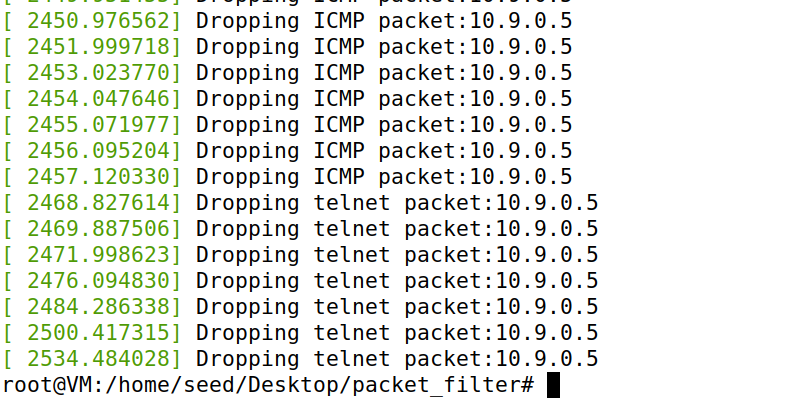
return 0;

}

编译加载内核模块，展示相关信息，清除日志

登录用户主机，对VM主机进行ping和telnet连接尝试，结果如下

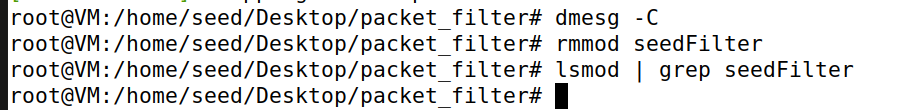


查看日志发现报文均被丢弃

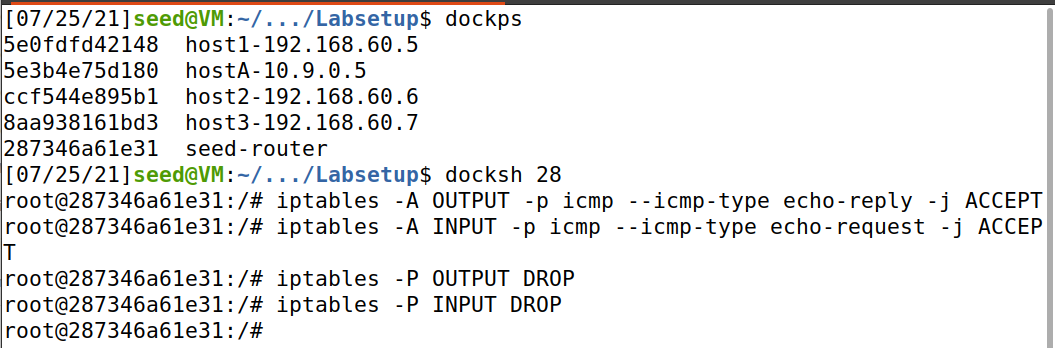
**TASK 2**

**2.A**

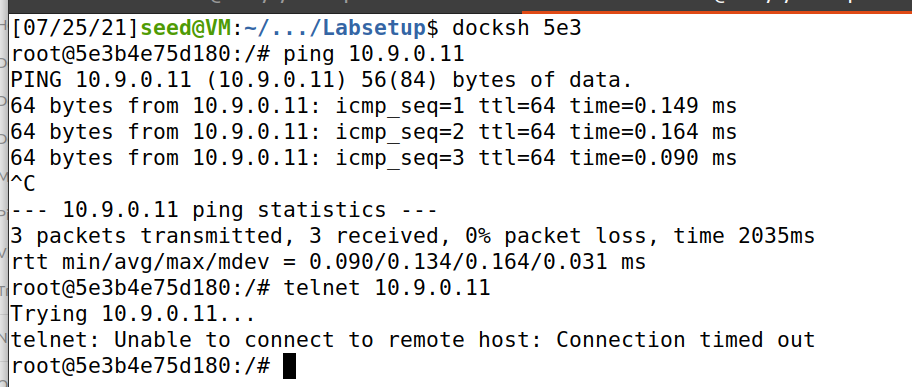
首先清除日志，关闭任务1中加载的内核模块防止干扰



登录路由对其进行配置，路由ip为10.9.0.11，需要注意的是pdf上所给的命令存在问题，检查后发现INPUT和OUTPUT对应的echo-request和reply关系错误，会导致ping命令也不通

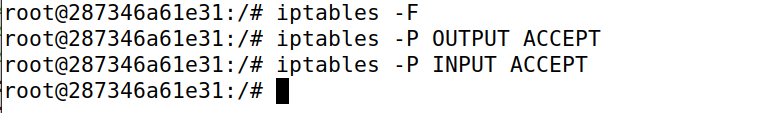


登录用户主机ping和telnet 路由器，结果如下

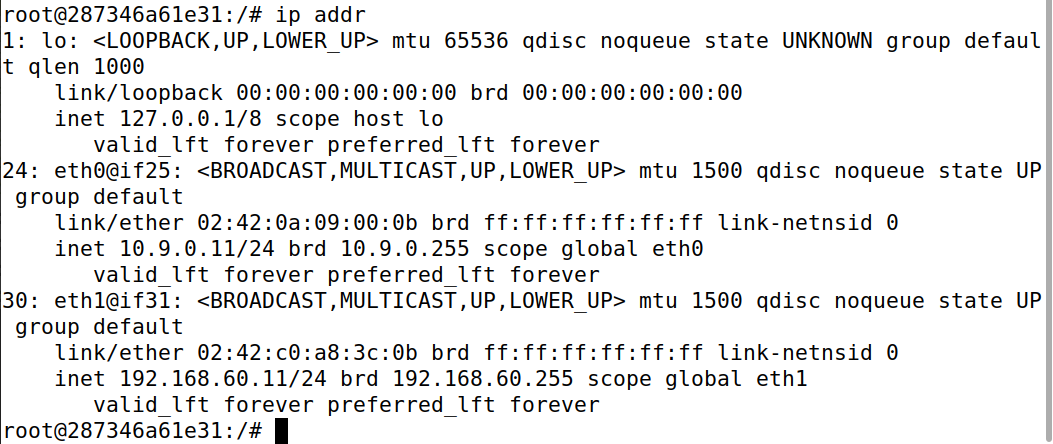


**2.B**

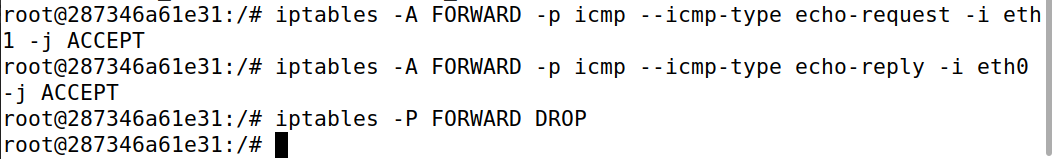
清除并恢复默认过滤表



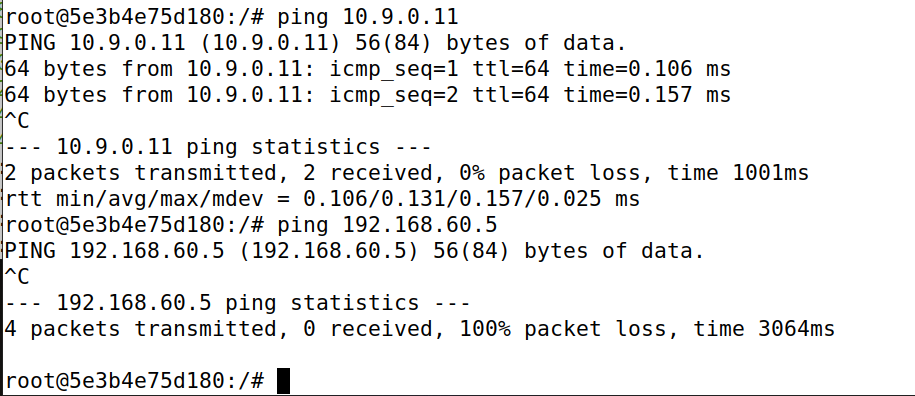
查看路由器的接口，可以看到eth0对应的是10.9.0.0/24网络，eth1对应的则是192.168.60.0/24



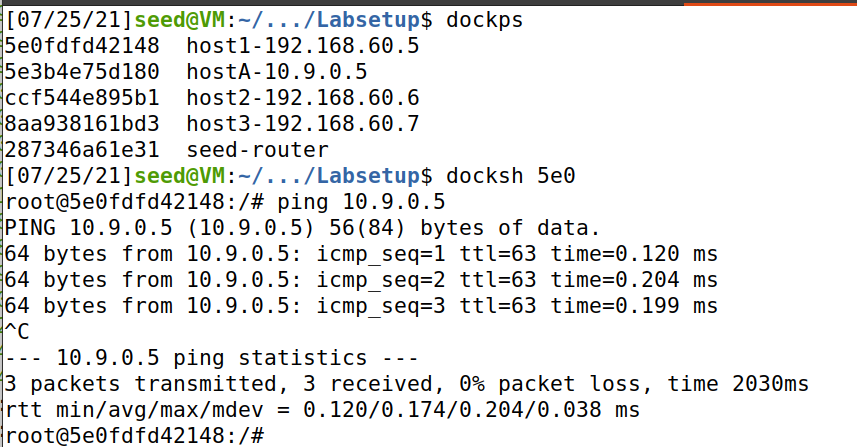
在路由器上配置如下规则，只允许内部主机向外部主机的ping的请求以及外部主机响应的转发，其他转发禁止



首先在用户主机上测试，可以ping通路由器但无法ping通内部主机

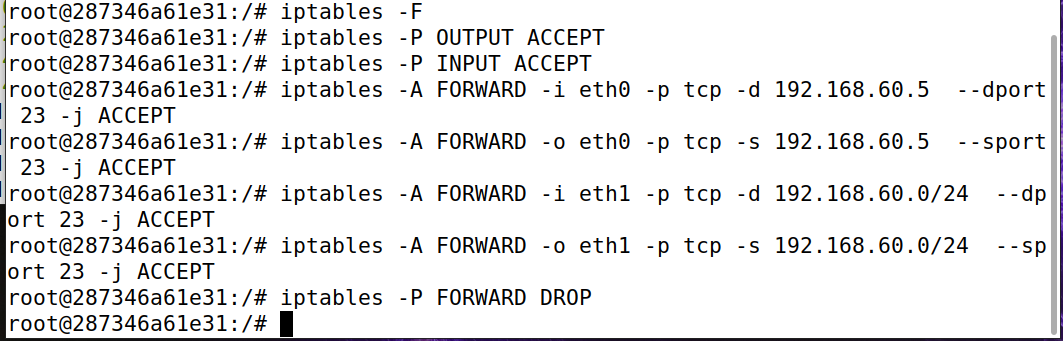


在内部主机上ping外部主机，结果成功

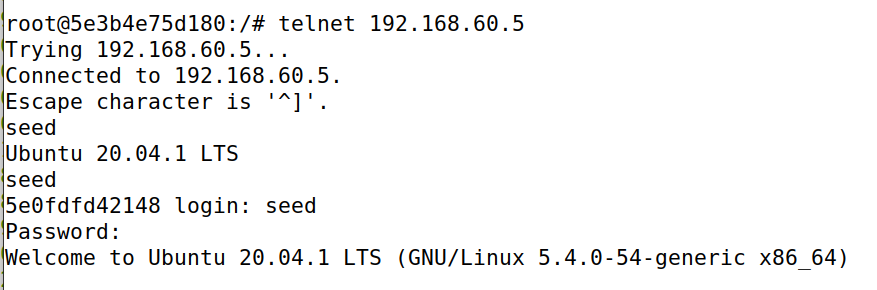
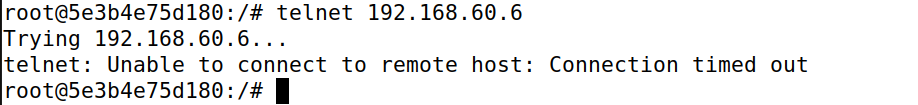


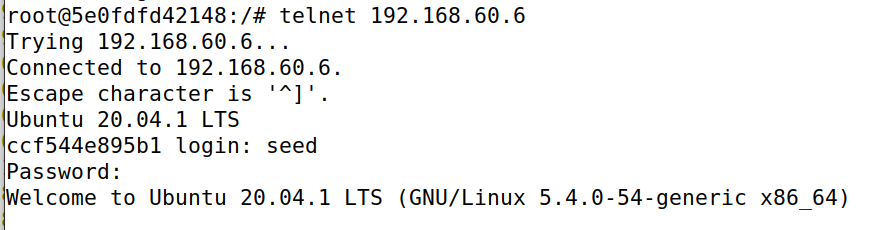
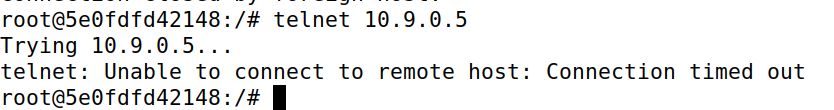
**2.C**

首先清除并恢复路由器上的默认配置，然后设置新的过滤规则



在用户主机10.9.0.5上进行测试



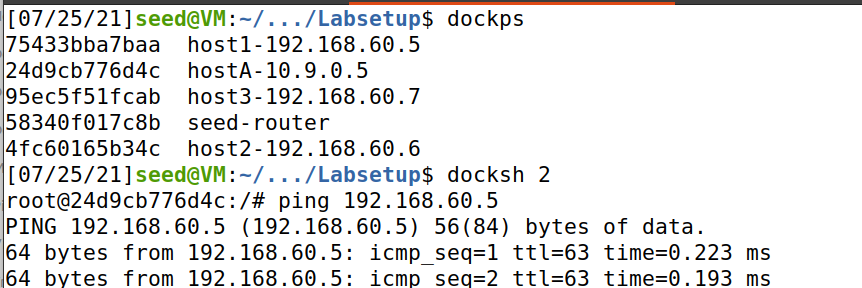
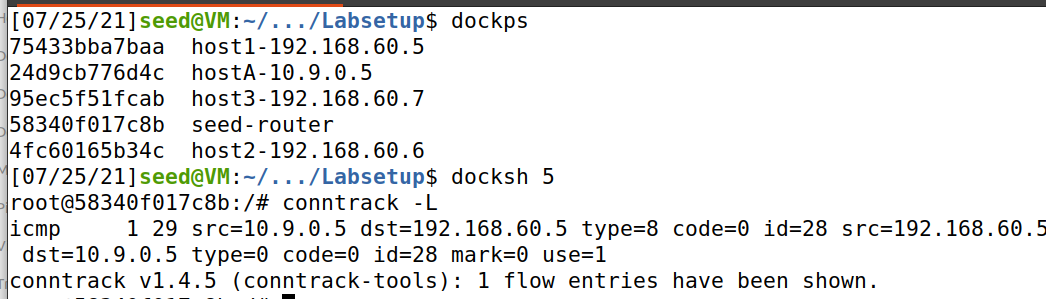
内部主机192.168.60.5上测试，结果如下

**TASK 3**

**3.A**

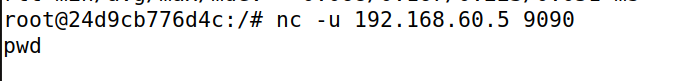
首先清除并恢复路由器默认配置

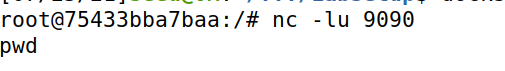
**ICMP experiment**

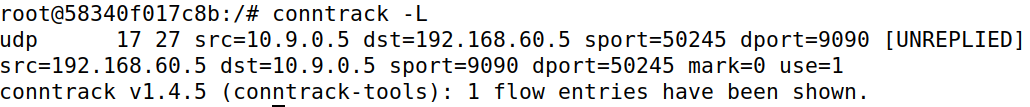


在路由器上输入命令进行连接跟踪，得到结果如图，ICMP保留连接时间约为30s。

**UDP experiment**



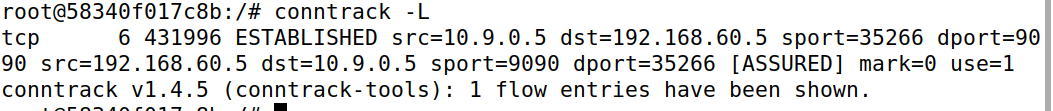




UDP保留连接时间约为30s。

**TCP experiment**

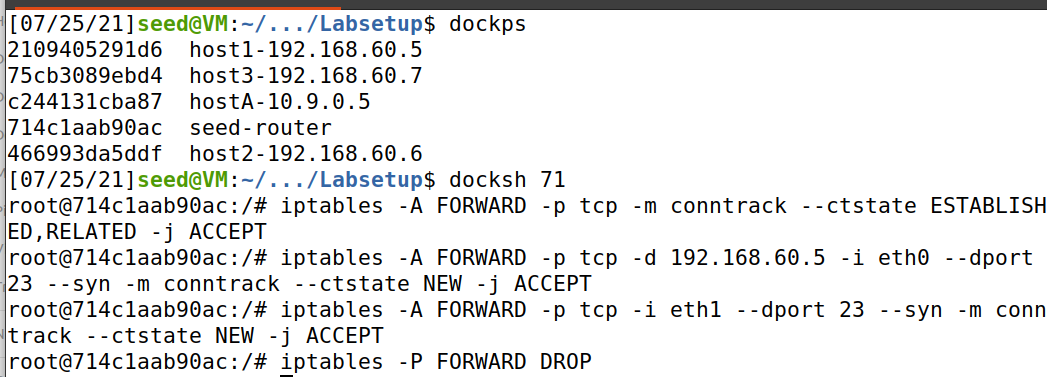




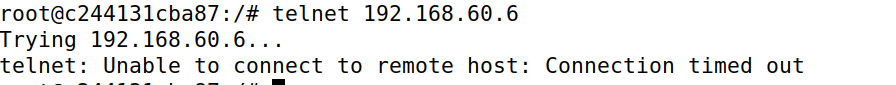
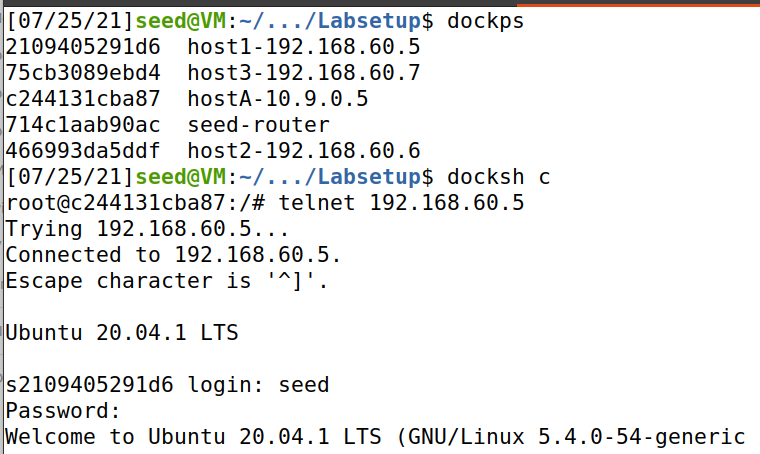
TCP保留连接时间约为432000s，即5天

**3.B**

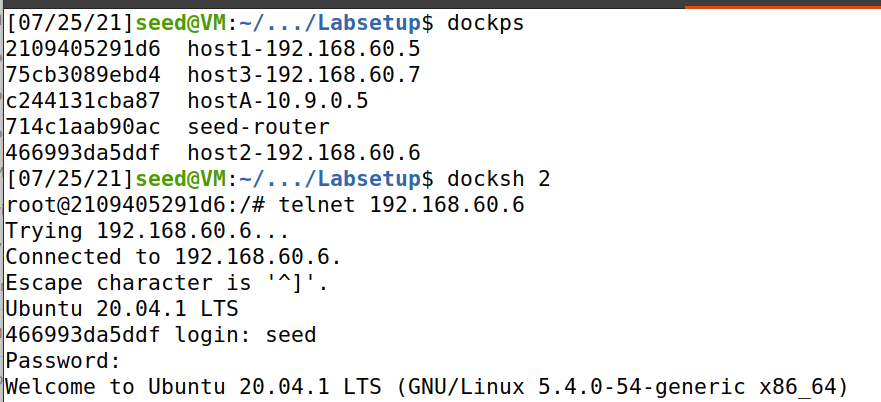
登录路由器配置如下规则

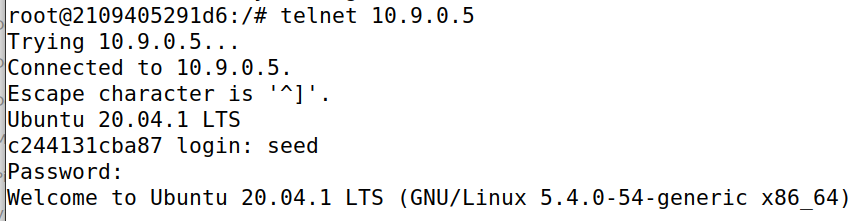


用户主机分别telnet 192.168.60.5和.6，结果如下，只有.5连接成功



内部主机分别telnet 192.168.60.6 和10.9.0.5，结果如下，均连接成功





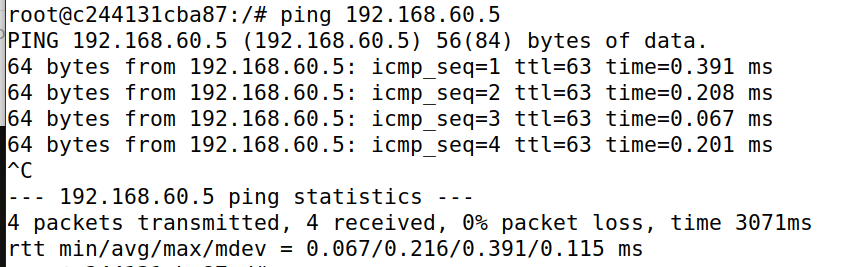
实验发现，不利用连接跟踪机制的过滤规则处理速度快，但无法定义精细的规则、不适合复杂的访问控制；而利用连接跟踪机制的过滤规则对数据包的状态也进行检查，能够定义更加严格的规则、应用范围更广、安全性更高，但是无法对数据包的内容进行识别，且对性能要求更高。

**TASK 4**

清除并恢复路由器默认配置后，首先添加下述规则



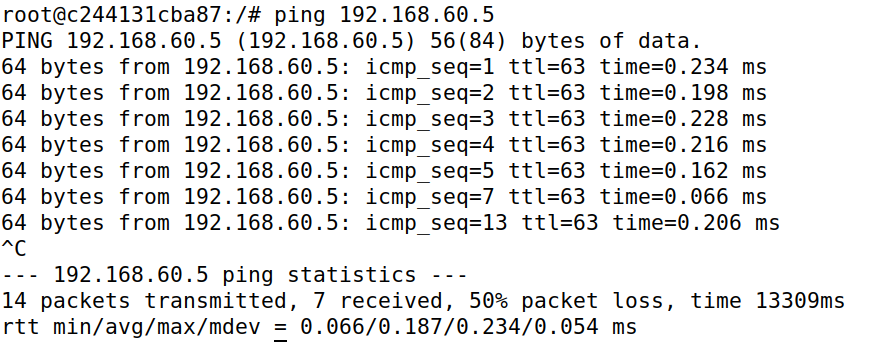
此时用户主机ping内部主机发现，并未限制报文流量



增加下述规则



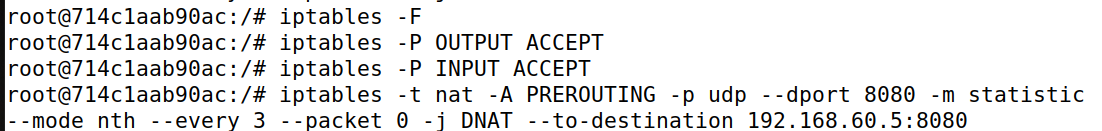
再次ping命令观察，流量受到了限制



该现象的原因是路由器的转发链的默认规则为ACCEPT，即使超过流量限制，报文根据默认规则也可以进行传输，因此需要第二条规则。

**TASK 5**

清除并恢复路由器默认配置，增加如下规则

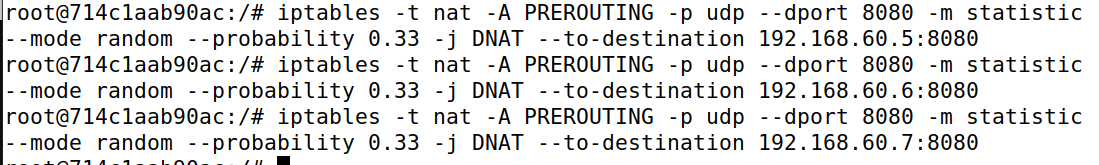


在 nth mode下

然后在用户主机上进行测试，输入echo hello | nc -u 10.9.0.11 8080

内部主机上输入nc -luk 8080 进行监听，理论上更改目的地址的报文数量应为输入命令条数的三分之一，实际进行大量测量发现数据并不理想接近二分之一故不放出

清除并重新配置路由器规则，在random模式下，输入以下规则命令



重新在用户主机上测试，分别在三个内部主机上监听，结果发现，大量数据测试下192.168.60.5修改的报文数量会明显地比其他两主机多很多，其他两主机反而是大致相等的情况

