Lab 4
VPN Lab: The Container Version

Author：57119108 吴桐

Date：2022.8.15

Lab Task

Task 1: Network Setup

在本次实验中，我们将在计算机（客户端）和网关之间创建一个 VPN 隧道，允许计算机通过网关安全地访问专用网络。我们至少需要三台机器：VPN 客户端（主机 U）、VPN 服务器（路由器/网关）和专用网络中的主机（主机 V）。网络拓扑如图 1 所示。

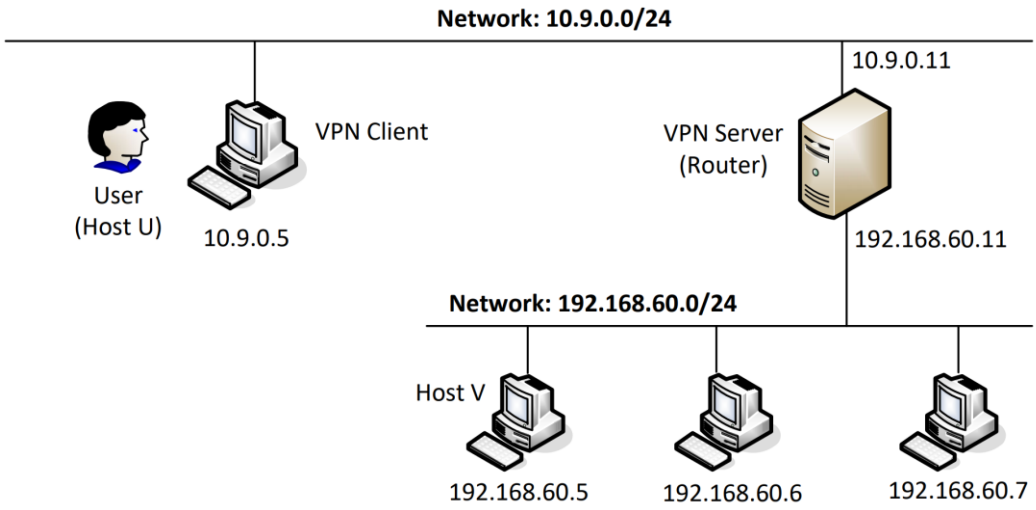


图 1

真实情况下，VPN 客户端和 VPN 服务器需要通过互联网连接。在实验环境中，我们简单得将这两台机器直接连接到同一局域网。主机 V 是专用网络中的一台计算机，被连接到了 VPN 服务器（网关）上。主机 V 不能直接从互联网访问，也不能直接从主机 U 访问。主机 U（专用网络之外）上的用户希望通过 VPN 隧道与主机 V 通信。

使用 dcup 命令创建本实验所需的容器。查看创建的各个容器，结果如图 2 所示。

```
[08/18/22]seed@VM:~/.../Labsetup$ dockps
2075e74b852b  server-router
3a19bcff8d37  client-10.9.0.5
a345ca878da8  host-192.168.60.5
196e7ee1d408  host-192.168.60.6
```

图 2

在实验过程中，我们需要嗅探数据包，以便分析程序的运行状况。我们可以在容器上运行 tcpdump 命令，用于嗅探特定接口上的数据包。需要注意的是，在容器内部进行嗅探时，我们只能得到进出该容器的数据包；在主机上使用 tcpdump 命令进行嗅探，则可以得到容器之间传输的所有数据包。主机上的网络接口名称与容器内的不同：容器内的接口名称通常以 eth 开头；而主机上由 Docker 创建的网络接口名称通常以 br 开头。当然，我们也可以使用主机上的 WireShark 来嗅探数据包。

（1）主机 U 可以与 VPN 服务器通信。

在主机 U 上 Ping VPN 服务器，结果如图 3 所示。

```
root@3a19bcff8d37:/# 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=0.155 ms
64 bytes from 10.9.0.11: icmp_seq=2 ttl=64 time=0.075 ms
64 bytes from 10.9.0.11: icmp_seq=3 ttl=64 time=0.149 ms
```

图 3

（2）VPN 服务器可以与主机 V 通信。

在 VPN 服务器上 Ping 主机 V，结果如图 4 所示。

```
root@2075e74b852b:/# 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=64 time=0.168 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.072 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=64 time=0.137 ms
```

图 4

（3）主机 U 不能直接与主机 V 通信。

在主机 U 上 Ping 主机 V，结果如图 5 所示。

```
root@3a19bcff8d37:/# 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 3079ms
```

图 5

（4）在网关（VPN 服务器）上运行 tcpdump 命令，嗅探网络上的数据包。

首先在 VPN 服务器上查看端口的信息，结果如图 6 所示。可以看到 eth0 连接主机 U 所在的网络，eth1 连接主机 V 所在的网络。

```
root@2075e74b852b:/# 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 98  bytes 10646 (10.6 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 85  bytes 7994 (7.9 KB)
        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 93  bytes 10344 (10.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 16  bytes 1344 (1.3 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        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
```

图 6

在 VPN 服务器上监听网络接口 eth0，从主机 U Ping VPN 服务器，结果如图 7 所示。

```
root@2075e74b852b:/# tcpdump -i eth0 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
06:29:08.405545 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 17, seq 1, length 64
06:29:08.405577 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 17, seq 1, length 64
06:29:09.409326 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 17, seq 2, length 64
06:29:09.409351 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 17, seq 2, length 64
06:29:10.433126 IP 10.9.0.5 > 10.9.0.11: ICMP echo request, id 17, seq 3, length 64
06:29:10.433187 IP 10.9.0.11 > 10.9.0.5: ICMP echo reply, id 17, seq 3, length 64
```

图 7

在 VPN 服务器上监听网络接口 eth1，从主机 V Ping 同网络主机（192.168.60.6），可以嗅探到一个 ARP 报文，从主机 V Ping VPN 服务器，可以得到 ICMP Echo 请求报文及回复报文。

```
root@2075e74b852b:/# tcpdump -i eth1 -n
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
06:32:14.677028 ARP, Request who-has 192.168.60.6 tell 192.168.60.5, length 28
06:32:50.007057 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 30, seq 1, length 64
06:32:50.007083 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 30, seq 1, length 64
06:32:51.009895 IP 192.168.60.5 > 192.168.60.11: ICMP echo request, id 30, seq 2, length 64
06:32:51.009996 IP 192.168.60.11 > 192.168.60.5: ICMP echo reply, id 30, seq 2, length 64
```

图 8

Task 2: Create and Configure TUN Interface

在本任务中，我们将使用 TUN 技术实现 VPN 隧道。我们需要创建一个 TUN 虚拟网络接口。一方面，内核通过该接口发送的数据包会被传递给用户程序；另一方面，用户程序写

入该接口的数据会被当做网络数据包进入操作系统内核。

我们将使用以下 Python 程序作为实验的基础，并在后续的实验过程中不断完善此代码。

```python
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'tun%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

while True:
   time.sleep(10)
```

图 9

Task 2.a: Name of the Interface

在主机 U 上运行图 9 所示代码（tun.py），结果如图 10 所示。该程序创建了一个 TUN 接口后被阻塞，否则程序结束 TUN 接口会被立即撤销。

```
root@3a19bcff8d37:/volumes# chmod a+x tun.py
root@3a19bcff8d37:/volumes# tun.py
Interface Name: tun0
```

图 10

查看主机 U 上的所有接口，结果如图 11 所示。可见，此时程序为我们创建了一个 TUN 虚拟网络接口，接口名称为 tun0。

```
root@3a19bcff8d37:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
8: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

图 11

如图 12 所示，修改 tun.py 创建自命名的 TUN 端口：cocot0。

```
16 ifr = struct.pack('16sH', b'cocot%d', IFF_TUN | IFF_NO_PI)
```

图 12

在主机 U 上重新运行 tun.py，结果如图 13 所示。可见，此时程序为我们创建了一个 TUN 虚拟网络接口，接口名称为 cocot0。

```
root@3a19bcff8d37:/volumes# tun.py
Interface Name: cocot0
```

图 13

此时，查看主机 U 上的所有接口，结果如图 14 所示。

```
root@3a19bcff8d37:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
3: cocot0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 500
    link/none
8: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

图 14

Task 2.b: Set up the TUN Interface

在此任务中，我们将配置并启动 TUN 接口。首先，我们需要为其分配一个 IP 地址；除此之外，我们还需要手动打开接口。在 tun.py 中添加红色方框所示的代码实现接口配置。

```
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'cocot%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

while True:
    time.sleep(10)
```

图 15

在主机 U 上重新运行 tun.py 后，查看主机 U 上的所有接口，结果如图 16 所示。可见，此时我们为创建的 TUN 接口分配了一个 IP 地址（192.168.53.99），并将接口状态设置为开启。这里我们还可以注意到，接口对应的编号增加了，在后续实验中我们会发现接口编号的上限是非常高的。

```
root@3a19bcff8d37:/# ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
4: cocot0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UNKNOWN group
 default qlen 500
    link/none
    inet 192.168.53.99/24 scope global cocot0
       valid_lft forever preferred_lft forever
8: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

图 16

Task 2.c: Read from the TUN Interface

在此任务中，我们将实现从 TUN 接口中读取数据。从 TUN 接口接收到的是一个 IP 数据包，我们需要将其转换成一个 Scapy IP 对象，便于读取 IP 数据包的每个字段。使用以下 while 循环替换 tun.py 中的循环：

```
while True:
        # Get a packet from the tun interface
        packet = os.read(tun, 2048)
        if packet:
                ip = IP(packet)
                print(ip.summary())
```

图 17

在主机 U 上运行 tun.py 后，从主机 U Ping 192.168.53.0/24 网段的任一主机（以 192.168.53.2 为例），结果如图 18~19 所示。可以看到，tun.py 产生了响应的输出，这就说明系统会将发往 192.168.53.0/24 网段的数据包转发到 TUN 接口，且程序能够正常地从 TUN 接口中读取数据包。

```
root@3a19bcff8d37:/# ping 192.168.53.2
PING 192.168.53.2 (192.168.53.2) 56(84) bytes of data.
^C
--- 192.168.53.2 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4095ms
```

图 18

```
root@3a19bcff8d37:/volumes# tun.py
Interface Name: cocot0
IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.2 echo-request 0 / Raw
```

图 19

从主机 U Ping 主机 192.168.60.6，结果如图 20 所示。此时，我们没有在 TUN 接口上接收到任何数据，这是因为系统中并没有配置相应的路由，将发往 192.168.60.0/24 网段的数据包转发到 TUN 接口。

```
root@3a19bcff8d37:/# ping 192.168.60.6
PING 192.168.60.6 (192.168.60.6) 56(84) bytes of data.
```

```
root@3a19bcff8d37:/volumes# tun.py
Interface Name: cocot0
```

图 20

这里需要注意，如果我们尝试从主机 U 连接 192.168.60.1，会发现 Ping 操作成功，但是 tun.py 无输出。这是因为在创建容器时，系统自动配置了容器之间的网桥的默认网关。

```
[08/18/22]seed@VM:~/.../Labsetup$ route -nN
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.2.1        0.0.0.0         UG    100    0        0 enp0s3
10.0.2.0        0.0.0.0         255.255.255.0   U     100    0        0 enp0s3
10.9.0.0        0.0.0.0         255.255.255.0   U     0      0        0 br-e92ab0737b5f
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 enp0s3
172.17.0.0      0.0.0.0         255.255.0.0     U     0      0        0 docker0
192.168.60.0    0.0.0.0         255.255.255.0   U     0      0        0 br-4739946ebfe7
[08/18/22]seed@VM:~/.../Labsetup$ ifconfig br-4739946ebfe7
br-4739946ebfe7: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.60.1  netmask 255.255.255.0  broadcast 192.168.60.255
        inet6 fe80::42:a0ff:fe74:6310  prefixlen 64  scopeid 0x20<link>
        ether 02:42:a0:74:63:10  txqueuelen 0  (Ethernet)
        RX packets 2  bytes 56 (56.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 63  bytes 7134 (7.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

图 21

Task 2.d: Write to the TUN Interface

在此任务中，我们将实现向 TUN 接口写数据。因为 TUN 建立的是虚拟网络接口，所以从接口写入的数据都会被内核当作是接收到的 IP 数据包。程序 tun.py 的内容如图 22 所示。当 TUN 接收到一个数据包时，若该数据包为 ICMP Echo 请求报文，程序就会伪造一个相应的 ICMP 回复报文，并将伪造报文写入 TUN 接口。

需要注意的是，在实验过程中，我们可能需要终止正在运行的程序，此时只需要按下 Ctrl+C 键即可。如果按下了 Ctrl+Z 键，我们会发现程序同样停止运行了，但此时程序只是被阻塞，并没有终止，再次运行程序时就会发现 TUN 接口的编号会递增（如 cocot1，cocot2），此时我们需要输入 exit 指令退出所有进程。

```python
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'cocot%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

while True:
        # Get a packet from the tun interface
        packet = os.read(tun, 2048)
        if packet:
                ip = IP(packet)
                print(ip.summary())
                if ICMP in ip:
                        # os.write(tun, bytes("Hello,world!", encoding='utf-8'))

                        # Send out a packet using the tun interface
                        newip = IP(src=ip[IP].dst, dst=ip[IP].src, ihl=ip[IP].ihl)
                        newip.ttl = 99
                        newicmp = ICMP(type=0, id=ip[ICMP].id, seq=ip[ICMP].seq)
                        if ip.haslayer(Raw):
                                data = ip[Raw].load
                                newpkt = newip/newicmp/data
                        else:
                                newpkt = newip/newicmp
                        os.write(tun, bytes(newpkt))
                        print('Send out:')
                        print(IP(bytes(newpkt)).summary())
```

图 22

在主机 U 上运行 tun.py 后，从主机 U Ping 192.168.53.0/24 网段的任一主机（以 192.168.53.1 为例），结果如图 23~24 所示。可以看到，这次我们的 Ping 操作成功了，且 tun.py 输出了相应的回复报文提示信息，这就说明程序成功伪造了正确的 ICMP Echo Reply 报文并将其写入了 TUN 接口中。

```
root@3a19bcff8d37:/# ping 192.168.53.1
PING 192.168.53.1 (192.168.53.1) 56(84) bytes of data.
64 bytes from 192.168.53.1: icmp_seq=1 ttl=99 time=2.40 ms
64 bytes from 192.168.53.1: icmp_seq=2 ttl=99 time=1.26 ms
64 bytes from 192.168.53.1: icmp_seq=3 ttl=99 time=1.22 ms
```

图 23

```
root@3a19bcff8d37:/volumes# tun.py
Interface Name: cocot0
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
Send out:
IP / ICMP 192.168.53.1 > 192.168.53.99 echo-reply 0 / Raw
```

图 24

修改程序 tun.py 的内容如图 25 所示，程序会向 TUN 接口写入我们自定义的一串字符。在主机 U 上运行 tun.py 后，从主机 U Ping 主机 192.168.53.1，结果如图 26~27 所示。可以看到，这次我们的 Ping 操作失败，TUN 接口接收到了 ICMP Echo 请求报文但是并没有返回正确的回复报文。

```
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'cocot%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

while True:
        # Get a packet from the tun interface
        packet = os.read(tun, 2048)
        if packet:
                ip = IP(packet)
                print(ip.summary())
                if ICMP in ip:
                        os.write(tun, bytes("Hello,world!", encoding='utf-8'))
```

图 25

```
root@3a19bcff8d37:/# ping 192.168.53.1
PING 192.168.53.1 (192.168.53.1) 56(84) bytes of data.
^C
--- 192.168.53.1 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3085ms
```

图 26

```
root@3a19bcff8d37:/volumes# tun.py
Interface Name: cocot0
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
IP / ICMP 192.168.53.99 > 192.168.53.1 echo-request 0 / Raw
```

图 27

Task 3: Send the IP Packet to VPN Server Through a Tunnel

在此任务中，我们将把从 TUN 接口接收的 IP 数据包放入新 IP 数据包的 UDP 有效负载中，并将其发送到另一台计算机。这实际上就是 IP 隧道的实现原理。程序 tun_server.py 的内容如图 28 所示。程序 tun_client.py 的内容如图 29 所示，其中红色方框标识的代码为新加入的内容。可以看到，我们在客户端与服务器上创建了 UDP Socket，当客户端从 TUN 接口上接收到报文后，就会将报文通过 UDP Socket 转发到服务器（10.9.0.11）的相应接口。

```
#!/usr/bin/env python3

from scapy.all import *

IP_A = "0.0.0.0"
PORT = 9090

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_A, PORT))

while True:
    data, (ip, port) = sock.recvfrom(2048)
    print("{}:{} --> {}:{}".format(ip, port, IP_A, PORT))
    pkt = IP(data)
    print(" Inside: {} --> {}".format(pkt.src, pkt.dst))
```

图 28

```
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

# Create UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
SERVER_IP, SERVER_PORT = '10.9.0.11', 9090

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'cocot%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

os.system("ip route add 192.168.60.0/24 dev {}".format(ifname))

while True:
        # Get a packet from the tun interface
        packet = os.read(tun, 2048)
        if packet:
                # Send the packet via the tunnel
                sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

图 29

我们希望客户端发出的前往 192.168.60.0/24 网段的数据包能够转发到 TUN 接口。因此，在程序 tun_client.py 的第 30 行，我们加入了一条路由信息，将相应的数据包转发到 TUN 接口。

首先，我们来测试一下程序是否正常工作。在主机 U 上运行 tun_client.py，在 VPN 服务器上运行 tun_server.py。在主机 U 上 Ping 主机 192.168.53.2（从 Task 2.c 可知，系统会将发往 192.168.53.0/24 网段的数据包转发到 TUN 接口），结果如图 30 所示。同时，服务器上产生了相应的输出（图 31）。可以看到，此时服务器收到了来自 10.9.0.5 的 ICMP Echo 请求报文，其内部 UDP 报文的源地址为 192.168.53.99，即为我们在 Task 2.b 中为 TUN 端口配置的 IP 地址，然而客户端并没有接收到正确的回复。

```
root@3a19bcff8d37:/# ping 192.168.53.2
PING 192.168.53.2 (192.168.53.2) 56(84) bytes of data.
^C
--- 192.168.53.2 ping statistics ---
8 packets transmitted, 0 received, 100% packet loss, time 7207ms
```

图 30

```
root@2075e74b852b:/volumes# tun_server.py
10.9.0.5:51993 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.2
10.9.0.5:51993 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.2
```

图 31

在主机 U 上查看路由信息，可以看到，发往 192.168.53.0/24 网段的数据包都会被转发到我们创建的 TUN 接口（cocot0）上。

```
root@3a19bcff8d37:/# ip route
default via 10.9.0.1 dev eth0
10.9.0.0/24 dev eth0 proto kernel scope link src 10.9.0.5
192.168.53.0/24 dev cocot0 proto kernel scope link src 192.168.53.99
192.168.60.0/24 dev cocot0 scope link
```

图 32

在主机 U 上运行 tun_client.py，在 VPN 服务器上运行 tun_server.py。在主机 U 上 Ping 主机 V（192.168.60.5），结果如图 33~34 所示。可以看到，此时服务器收到了来自 10.9.0.5 的 ICMP Echo 请求报文，其内部 UDP 报文的源地址为 192.168.53.99（TUN 端口 IP 地址），宿地址为 192.168.60.5，然而客户端并没有接收到正确的回复。

```
root@3a19bcff8d37:/# 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 3062ms
```

图 33

```
root@2075e74b852b:/volumes# tun_server.py
10.9.0.5:48270 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:48270 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
```

图 34

Task 4: Set Up the VPN Server

当 tun_server.py 接收到隧道传来的数据包后，它需要将其递交给内核，内核会将数据包转发到其最终的目的地。这就需要我们在服务器上建立一个 TUN 虚拟网络接口，该接口会接收客户端应用程序（tun_client.py）发送的 IP 隧道数据包，将其递交给操作系统内核。程序 tun_server.py 的内容如图 35 所示，这里我们为 TUN 接口配置了 IP 地址 192.268.53.11，与客户端的 TUN 接口在同一网段下。

```
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'cocot%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")

os.system("ip addr add 192.168.53.11/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

IP_A = "0.0.0.0"
PORT = 9090

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_A, PORT))

while True:
        data, (ip, port) = sock.recvfrom(2048)
        print("{}:{} --> {}:{}".format(ip, port, IP_A, PORT))
        pkt = IP(data)
        print(" Inside: {} --> {}".format(pkt.src, pkt.dst))

        os.write(tun, bytes(pkt))
```

图 35

在主机 U 上运行 tun_client.py，在 VPN 服务器上运行 tun_server.py。在主机 U 上 Ping 主机 V（192.168.60.5），结果如图 36~37 所示。我们可以使用 WireShark 监听主机 V 的网络

端口，结果如图 38 所示。可以看到，主机 V 在收到了来自主机 U 的 ICMP Echo 请求后回复了相应的 ICMP Echo Reply 报文，但是服务器此时还无法将数据包反向传回客户端。

```
root@3a19bcff8d37:/# 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 ---
5 packets transmitted, 0 received, 100% packet loss, time 4091ms
```

图 36

```
root@2075e74b852b:/volumes# tun_server.py
10.9.0.5:45591 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
10.9.0.5:45591 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.60.5
```

图 37

No.	Time	Source	Destination	Protocol	Length	Info
1	2022-08-18 05:33:55.773116786	192.168.53.99	192.168.60.5	ICMP	98	Echo (ping) request id=0x0159, seq=1/256, ttl=63 (reply in 2)
2	2022-08-18 05:33:55.773158691	192.168.60.5	192.168.53.99	ICMP	98	Echo (ping) reply id=0x0159, seq=1/256, ttl=64 (request in 1)
3	2022-08-18 05:33:56.804531921	192.168.53.99	192.168.60.5	ICMP	98	Echo (ping) request id=0x0159, seq=2/512, ttl=63 (reply in 4)
4	2022-08-18 05:33:56.804707705	192.168.60.5	192.168.53.99	ICMP	98	Echo (ping) reply id=0x0159, seq=2/512, ttl=64 (request in 3)
5	2022-08-18 05:33:57.827124301	192.168.53.99	192.168.60.5	ICMP	98	Echo (ping) request id=0x0159, seq=3/768, ttl=63 (reply in 6)
6	2022-08-18 05:33:57.827191908	192.168.60.5	192.168.53.99	ICMP	98	Echo (ping) reply id=0x0159, seq=3/768, ttl=64 (request in 5)

图 38

Task 5: Handling Traffic in Both Directions

在此任务中，我们将解决 Task 4 遗留的问题：使用 TUN 接口传输双向流量。为了实现这一点，我们的 TUN 客户端和服务器程序需要从两个接口读取数据，即 TUN 接口和 Socket 接口，这些接口由文件描述符表示，在 Linux 系统上我们可以使用 select 系统调用同时监视多个文件描述符。

程序 tun_client.py 的内容如图 39 所示。如果程序接收到 Socket 接口传来的数据包，即该数据包从服务器端通过隧道传输到客户端，程序会将该数据包的有效负载提取出来，作为 IP 数据包写入 TUN 接口中，递交给操作系统内核，操作系统会将其转发给最终的宿地址；如果程序接收到 TUN 接口传来的数据包，程序会将添加新的 IP 头，将其封装成新的 IP 数据包，并通过 Socket 接口发送。

同理，程序 tun_server.py 和内容如图 40 所示，while 循环内的代码与 tun_client.py 完全一致。需要说明的是，当专用网络中的主机 V 向主机 U 发送信息时，主机 V 的默认路由会将其导向 VPN 服务器，从图 38 中我们可以看到，该 ICMP 回复报文的目的地址 192.168.53.99，VPN 服务器会自动将其转发到 TUN 接口上（Task 2.c 的结论）。当服务器程序接收到 TUN 接口传来的数据包时，需要将其通过 Socket 接口发送到客户端（10.9.0.5）。

之所以用了很大的篇幅去讲解程序的原理，是因为我们在这里犯了一个美丽的错误，这

个错误需要在 Task 8 中纠正过来。

```python
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

# Create UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
SERVER_IP, SERVER_PORT = '10.9.0.11', 9090

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'cocot%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

os.system("ip addr add 192.168.53.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

os.system("ip route add 192.168.60.0/24 dev {}".format(ifname))

while True:
    # this will block until at least one interface is ready
    ready, _, _ = select.select([sock, tun], [], [])

    for fd in ready:
        if fd is sock:
            data, (ip, port) = sock.recvfrom(2048)
            pkt = IP(data)
            print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
            os.write(tun, bytes(pkt))
        if fd is tun:
            packet = os.read(tun, 2048)
            pkt = IP(packet)
            print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
            sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

图 39

```python
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'cocot%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")

os.system("ip addr add 192.168.53.11/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

IP_A = "0.0.0.0"
PORT = 9090

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_A, PORT))
SERVER_IP, SERVER_PORT = '10.9.0.5', 9090

while True:
    # this will block until at least one interface is ready
    ready, _, _ = select.select([sock, tun], [], [])

    for fd in ready:
        if fd is sock:
            data, (ip, port) = sock.recvfrom(2048)
            pkt = IP(data)
            print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
            os.write(tun, bytes(pkt))
        if fd is tun:
            packet = os.read(tun, 2048)
            pkt = IP(packet)
            print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
            sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

图 40

在主机 U 上运行 tun_client.py，在 VPN 服务器上运行 tun_server.py。在主机 U 上 Ping 主机 V（192.168.60.5），结果如图 41~43 所示。我们可以使用 WireShark 嗅探网络上的数据包，结果如图 44 所示。可以看到，Ping 大致可以分为四个过程：主机 U（客户端）向服务器发送请求→服务器将请求转发主机 V→主机 V 向服务器发送回复→服务器将回复返回给主机 U。

```
root@3a19bcff8d37:/# 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=4.45 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=63 time=7.52 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=63 time=9.17 ms
```

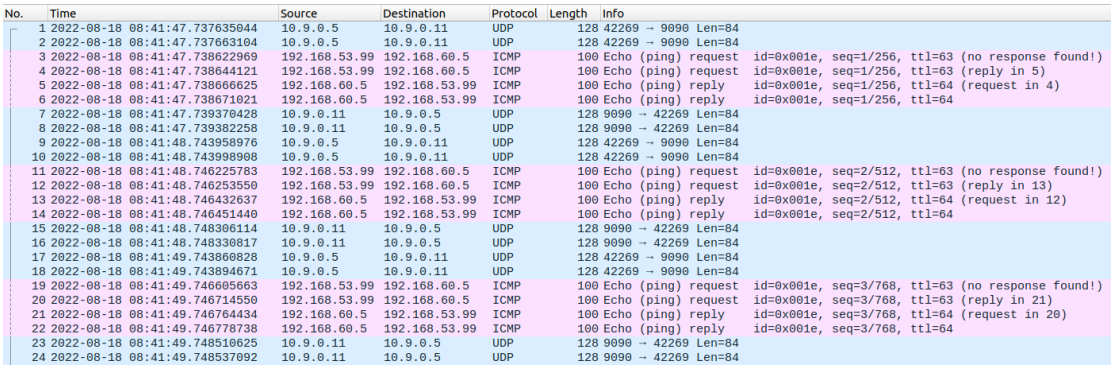
图 41

```
root@3a19bcff8d37:/volumes# tun_client.py
Interface Name: cocot0
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
From tun ==>: 192.168.53.99 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.53.99
```

图 42

```
root@2075e74b852b:/volumes# tun_server.py
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
From socket <==: 192.168.53.99 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.53.99
```

图 43



No.	Time	Source	Destination	Protocol	Length	Info
1	2022-08-18 08:41:47.737635044	10.9.0.5	10.9.0.11	UDP	128	42269 → 9090 Len=84
2	2022-08-18 08:41:47.737663104	10.9.0.5	10.9.0.11	UDP	128	42269 → 9090 Len=84
3	2022-08-18 08:41:47.738622969	192.168.53.99	192.168.60.5	ICMP	100	Echo (ping) request id=0x001e, seq=1/256, ttl=63 (no response found!)
4	2022-08-18 08:41:47.738644121	192.168.53.99	192.168.60.5	ICMP	100	Echo (ping) request id=0x001e, seq=1/256, ttl=63 (reply in 5)
5	2022-08-18 08:41:47.738666625	192.168.60.5	192.168.53.99	ICMP	100	Echo (ping) reply id=0x001e, seq=1/256, ttl=64 (request in 4)
6	2022-08-18 08:41:47.738671021	192.168.60.5	192.168.53.99	ICMP	100	Echo (ping) reply id=0x001e, seq=1/256, ttl=64
7	2022-08-18 08:41:47.739370428	10.9.0.11	10.9.0.5	UDP	128	9090 → 42269 Len=84
8	2022-08-18 08:41:47.739382258	10.9.0.11	10.9.0.5	UDP	128	9090 → 42269 Len=84
9	2022-08-18 08:41:48.743958976	10.9.0.5	10.9.0.11	UDP	128	42269 → 9090 Len=84
10	2022-08-18 08:41:48.743998908	10.9.0.5	10.9.0.11	UDP	128	42269 → 9090 Len=84
11	2022-08-18 08:41:48.746225783	192.168.53.99	192.168.60.5	ICMP	100	Echo (ping) request id=0x001e, seq=2/512, ttl=63 (no response found!)
12	2022-08-18 08:41:48.746253550	192.168.53.99	192.168.60.5	ICMP	100	Echo (ping) request id=0x001e, seq=2/512, ttl=63 (reply in 13)
13	2022-08-18 08:41:48.746432637	192.168.60.5	192.168.53.99	ICMP	100	Echo (ping) reply id=0x001e, seq=2/512, ttl=64 (request in 12)
14	2022-08-18 08:41:48.746451440	192.168.60.5	192.168.53.99	ICMP	100	Echo (ping) reply id=0x001e, seq=2/512, ttl=64
15	2022-08-18 08:41:48.748306114	10.9.0.11	10.9.0.5	UDP	128	9090 → 42269 Len=84
16	2022-08-18 08:41:48.748330817	10.9.0.11	10.9.0.5	UDP	128	9090 → 42269 Len=84
17	2022-08-18 08:41:49.743860828	10.9.0.5	10.9.0.11	UDP	128	42269 → 9090 Len=84
18	2022-08-18 08:41:49.743894671	10.9.0.5	10.9.0.11	UDP	128	42269 → 9090 Len=84
19	2022-08-18 08:41:49.746605663	192.168.53.99	192.168.60.5	ICMP	100	Echo (ping) request id=0x001e, seq=3/768, ttl=63 (no response found!)
20	2022-08-18 08:41:49.746714550	192.168.53.99	192.168.60.5	ICMP	100	Echo (ping) request id=0x001e, seq=3/768, ttl=63 (reply in 21)
21	2022-08-18 08:41:49.746764434	192.168.60.5	192.168.53.99	ICMP	100	Echo (ping) reply id=0x001e, seq=3/768, ttl=64 (request in 20)
22	2022-08-18 08:41:49.746778738	192.168.60.5	192.168.53.99	ICMP	100	Echo (ping) reply id=0x001e, seq=3/768, ttl=64
23	2022-08-18 08:41:49.748510625	10.9.0.11	10.9.0.5	UDP	128	9090 → 42269 Len=84
24	2022-08-18 08:41:49.748537092	10.9.0.11	10.9.0.5	UDP	128	9090 → 42269 Len=84

图 44

我们可以在主机 U 上使用 telnet 命令连接主机 V，结果如图 45 所示。

```
root@8b50fe6f74bf:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
2f1979050909 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.
```

图 45

Task 6: Tunnel-Breaking Experiment

我们可以在主机 U 上使用 telnet 命令连接主机 V，成功后我们就得到了一个主机 V 的 shell，结果如图 46 所示。

```
root@3a19bcff8d37:/# telnet 192.168.60.5
Trying 192.168.60.5...
Connected to 192.168.60.5.
Escape character is '^]'.
Ubuntu 20.04.1 LTS
a345ca878da8 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: Thu Aug 18 09:51:54 UTC 2022 on pts/2
seed@a345ca878da8:~$ ls
seed@a345ca878da8:~$ cd ..
seed@a345ca878da8:/home$ ls
seed
```

图 46

此时，我们终止主机 U 上的 tun_client.py，可以发现主机 V 的 shell 并没有退出，但是不管我们输入什么，shell 都不会显示。

这时，我们在主机 U 上重新运行 tun_client.py，可以看到主机 V 的 shell 显示了之前我们输入的指令（图 47）。这是因为 VPN 的建立对于上层应用来说是透明的，断开连接后，应用不会立刻终止，输入的内容会保存在缓冲区内，当连接重新建立后，缓冲区内的内容会被立即发送。

```
seed@a345ca878da8:/home$ cd ..
seed@a345ca878da8:/$ ls
bin   dev  home  lib32  libx32  mnt  proc  run   srv  tmp  var
boot  etc  lib   lib64  media   opt  root  sbin  sys  usr
seed@a345ca878da8:/$
```

图 47

Task 7: Routing Experiment on Host V

在 Task 5 中，我们提到了主机 V 的默认路由会将非本地网络的流量全部导向 VPN 服务器（192.168.60.11）。如图 48 所示，我们删除默认路由，并指定前往 192.168.53.0/24 网段的数据包转发到 VPN 服务器上。重复 Task 5，可以发现主机 U 与主机 V 仍能够正常通信（图 49）。

```
root@a345ca878da8:/# ip route
default via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@a345ca878da8:/# ip route del default
root@a345ca878da8:/# ip route
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
root@a345ca878da8:/# ip route add 192.168.53.0/24 via 192.168.60.11
root@a345ca878da8:/# ip route
192.168.53.0/24 via 192.168.60.11 dev eth0
192.168.60.0/24 dev eth0 proto kernel scope link src 192.168.60.5
```

图 48

```
seed@a345ca878da8:/$ 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=64 time=0.109 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=64 time=0.107 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=64 time=0.096 ms
```

图 49

Task 8: VPN Between Private Networks

在此任务中，我们将在两个专用网络之间建立 VPN。网络拓扑如图 50 所示。

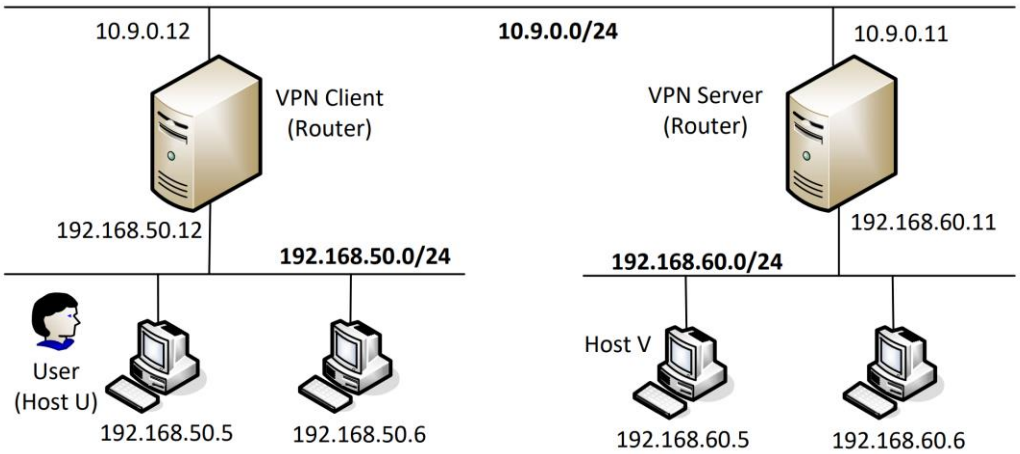


图 50

如图 51 所示，启动相应的容器。

```
[08/18/22]seed@VM:~/.../Labsetup$ docker-compose -f docker-compose2.yml up
```

图 51

查看创建的各个容器，结果如图 52 所示。

```
[08/18/22]seed@VM:~/.../Labsetup$ dockps
9b30f735171d  host-192.168.60.6
4ef9330d351b  server-router
6c8b5ce1ed5f  host-192.168.60.5
1e45a5169442  client-10.9.0.5
59e46e6ff526  host-192.168.50.5
0ce42a3ed386  host-192.168.50.6
```

图 52

从主机 U Ping 主机 V，结果如图 53 所示。从主机 V Ping 主机 U 结果如图 53 所示。可见，此时主机 U 与主机 V 之间不能互相通信。

```
root@59e46e6ff526:/# 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 3113ms
```

图 53

```
root@6c8b5ce1ed5f:/# ping 192.168.50.5
PING 192.168.50.5 (192.168.50.5) 56(84) bytes of data.
^C
--- 192.168.50.5 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4093ms
```

图 54

修改程序 tun_server.py 的内容如图 55 所示。

```
#!/usr/bin/env python3

import fcntl
import struct
import os
import time
from scapy.all import *

TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002
IFF_NO_PI = 0x1000

# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'cocot%d', IFF_TUN | IFF_NO_PI)
ifname_bytes  = fcntl.ioctl(tun, TUNSETIFF, ifr)

# Get the interface name
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")

os.system("ip addr add 192.168.53.11/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

os.system("ip route add 192.168.50.0/24 dev {}".format(ifname))

IP_A = "0.0.0.0"
PORT = 9090

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_A, PORT))
SERVER_IP, SERVER_PORT = '10.9.0.12', 9090

while True:
    # this will block until at least one interface is ready
    ready, _, _ = select.select([sock, tun], [], [])

    for fd in ready:
        if fd is sock:
            data, (ip, port) = sock.recvfrom(2048)
            pkt = IP(data)
            print("From socket <==: {} --> {}".format(pkt.src, pkt.dst))
            os.write(tun, bytes(pkt))
        if fd is tun:
            packet = os.read(tun, 2048)
            pkt = IP(packet)
            print("From tun ==>: {} --> {}".format(pkt.src, pkt.dst))
            sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

图 55

由网络拓扑图可知，服务器在接收到 TUN 接口传来的数据包后，需要将其通过 Socket 接口发送到客户端（10.9.0.12）。

在 Task 5 中，我们的主机 U 与 VPN 客户端二合为一，从主机 U 向主机 V 发出 Ping 请求时，前往 192.168.60.0/24 网段的数据包会被自动转发到 TUN 接口，tun_client.py 程序在请求的源地址字段填充了 TUN 接口的 IP 地址，并通过 Socket 接口发送给服务器；当服务器得到主机 V 的回复报文后，报文的目的地址为 192.168.53.99（客户端 TUN 接口的 IP 地址），服务器顺理成章地将回复转发到 TUN 接口（Task 2.c 的结论），之后通过 Socket 接口发送给 VPN 客户端（主机 U），主机 U 收到回复，Ping 操作成功。

上述的通信过程中存在着一个致命的漏洞：主机 U 与服务器位于同一网段，因此服务器可以将回复直接转发给主机。在 Task 8 中，主机 U 与客户端分离，从主机 U 发出的 Ping 请求（源地址为 192.168.50.5）会根据默认路由转发到客户端上，根据图 39 所示的代码 tun_client.py，客户端会将请求转发到服务器，服务器再将其转发到直接相连的主机 V 上；主机 V 的回复报文根据默认路由转发到服务器上，报文的目的地址为 192.168.50.5，由于没有相应的路由，操作系统就不会将报文转发到 TUN 接口，从而导致回复没法返回客户端。

因此我们需要在程序 tun_server.py 中添加一条路由信息，前往 192.168.50.0/24 网段的数据包转发到 TUN 接口。

在 VPN 客户端上运行 tun_client.py，在 VPN 服务器上运行 tun_server.py。在主机 U 上 Ping 主机 V，结果如图 56~58 所示。

```
root@59e46e6ff526:/# 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=62 time=5.76 ms
64 bytes from 192.168.60.5: icmp_seq=2 ttl=62 time=7.75 ms
64 bytes from 192.168.60.5: icmp_seq=3 ttl=62 time=8.39 ms
```

图 56

```
root@1e45a5169442:/volumes# tun_client.py
Interface Name: cocot0
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
From tun ==>: 192.168.50.5 --> 192.168.60.5
From socket <==: 192.168.60.5 --> 192.168.50.5
```

图 57

```
root@4ef9330d351b:/volumes# tun_server.py
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
From socket <==: 192.168.50.5 --> 192.168.60.5
From tun ==>: 192.168.60.5 --> 192.168.50.5
```

图 58

Summary

通过本次实验，我对 VPN 的原理和技术有了一个全面的认知。在实验过程中，一定要考虑清楚 TUN 接口和 Socket 接口的作用，在每台机器上配置正确的路由。