VPN Tunneling Lab

Task 1: Network Setup

主机网络信息如下：

```
[09/22/20]seed@VM:~$ ifconfig
ens33     Link encap:Ethernet  HWaddr 00:0c:29:99:36:68
          inet addr:192.168.220.129  Bcast:192.168.220.255  Mask:255.255.255.0
          inet6 addr: fe80::87c5:5446:9a64:9ea7/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2565 errors:0 dropped:0 overruns:0 frame:0
          TX packets:384 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:415089 (415.0 KB)  TX bytes:34450 (34.4 KB)
          Interrupt:19 Base address:0x2000
```

图 1.1 VM1（VPN Client）网络信息

```
[09/22/20]seed@VM:~$ ifconfig
ens33     Link encap:Ethernet  HWaddr 00:0c:29:14:18:9a
          inet addr:192.168.220.133  Bcast:192.168.220.255  Mask:255.255.255.0
          inet6 addr: fe80::de98:8a3a:c686:99c6/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1166 errors:0 dropped:0 overruns:0 frame:0
          TX packets:362 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:104828 (104.8 KB)  TX bytes:38122 (38.1 KB)
          Interrupt:19 Base address:0x2000

ens38     Link encap:Ethernet  HWaddr 00:0c:29:14:18:a4
          inet addr:10.42.0.100  Bcast:10.42.0.255  Mask:255.255.255.0
          inet6 addr: fe80::f0e1:ab1a:8c68:74d0/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:490 errors:0 dropped:0 overruns:0 frame:0
          TX packets:396 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:75700 (75.7 KB)  TX bytes:64331 (64.3 KB)
          Interrupt:16 Base address:0x2080
```

图 1.2 VM2（VPN Server）网络信息

```
[09/22/20]seed@VM:~$ ifconfig
ens33     Link encap:Ethernet  HWaddr 00:0c:29:73:43:db
          inet addr:10.42.0.101  Bcast:10.42.0.255  Mask:255.255.255.0
          inet6 addr: fe80::f517:6227:78a7:2efa/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:646 errors:0 dropped:0 overruns:0 frame:0
          TX packets:520 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:79142 (79.1 KB)  TX bytes:78924 (78.9 KB)
          Interrupt:19 Base address:0x2000
```

图 1.3 VM3（内网主机）网络信息

可见，在“公网”上客户端和服务器端处在 192.168.220.0/24 网段下，在“内网”中服务器和主机处在 10.42.0.0/24 网段下。下面测试网络联通状态。

```
[09/22/20]seed@VM:~$ ping 192.168.220.133
PING 192.168.220.133 (192.168.220.133) 56(84) bytes of data.
64 bytes from 192.168.220.133: icmp_seq=1 ttl=64 time=0.703 ms
64 bytes from 192.168.220.133: icmp_seq=2 ttl=64 time=0.845 ms
64 bytes from 192.168.220.133: icmp_seq=3 ttl=64 time=0.404 ms
^C
--- 192.168.220.133 ping statistics ---
```

图 1.4 主机 U 成功访问服务器

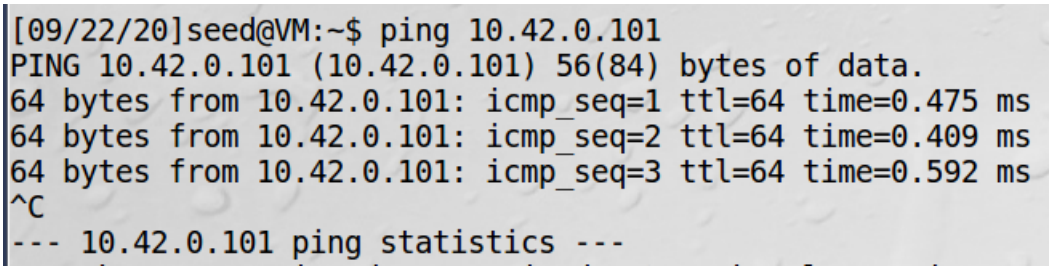

图 1.5 服务器成功访问主机 V

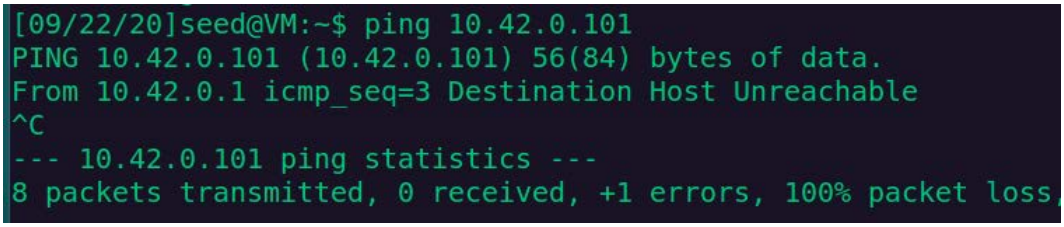

图 1.6 主机 U 不可访问主机 V

Task 2: Create and Configure TUN Interface

Task 2.a: Name of the Interface

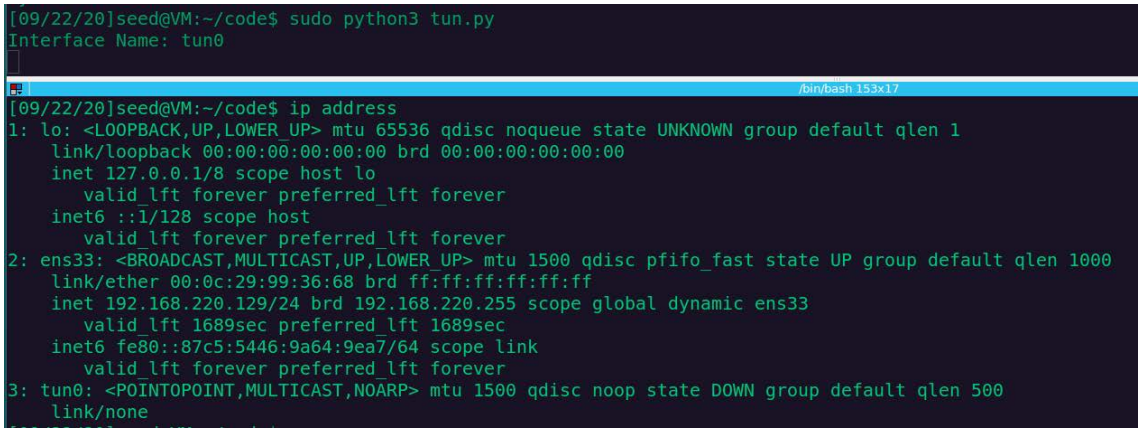

图 2.1.1 添加虚拟网卡

运行脚本之后，用 ip address 命令查看地址，发现多出了一张网卡，即 tun0。但此时网卡还没有被赋予 IP 地址。

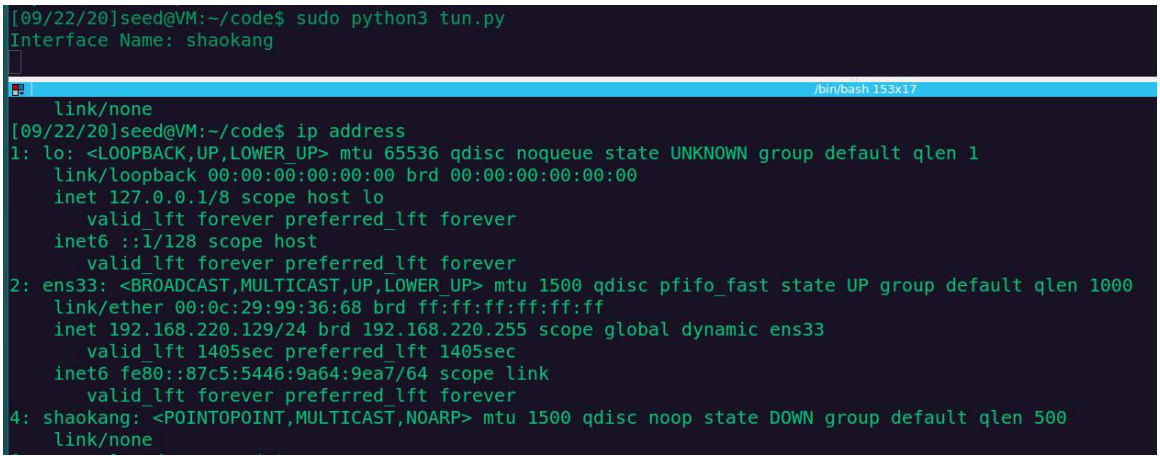

图 2.1.2 修改网卡名

简单修改 ifr 信息即可实现定制网卡名。

Task 2.b: Set up the TUN Interface

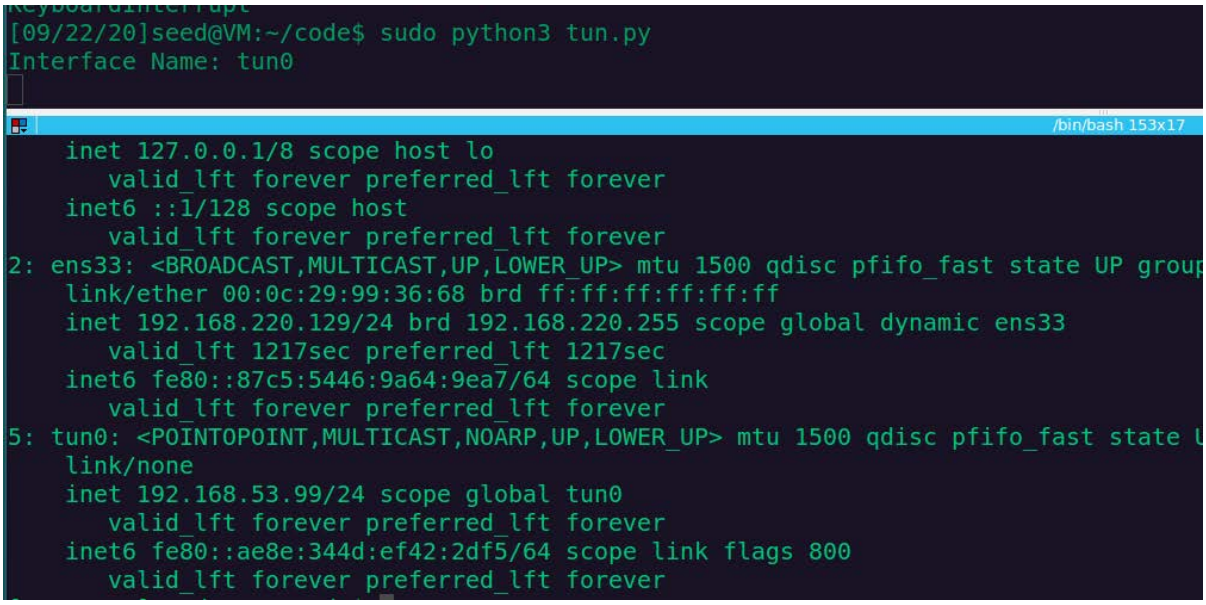

图 2.2.1 分配 IP 地址

此时再运行 ip address，发现网卡已经具有了相应的 IP 地址。

Task 2.c: Read from the TUN Interface

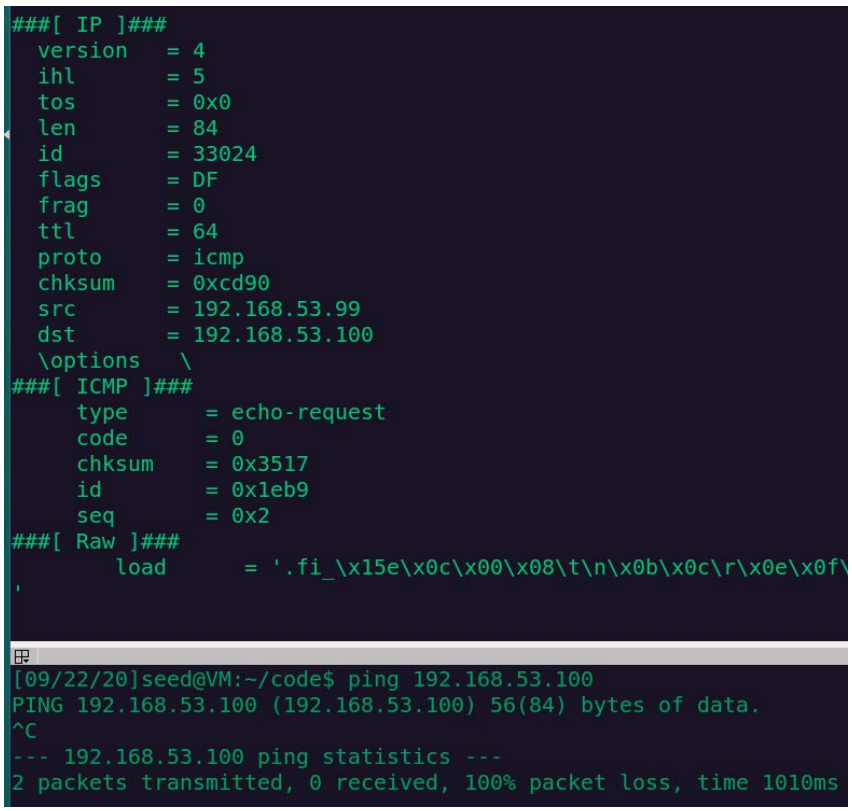

图 2.3.1 ping 192,168.53.0/24

此时运行 tun.py 的终端有输出，可以看到一条从 192.168.53.99 发往 192.168.53.100 的报文。因为 ping 主机属于 192.168.53.0/24 网段，因此报文从 tun0 网卡发出，而程序监听了 tun0 网卡上的所有报文，所以此 ICMP request 报文被打印出来。

```
^CTraceback (most recent call last):
  File "tun.py", line 23, in <module>
    packet = os.read(tun, 2048)
KeyboardInterrupt
[09/22/20]seed@VM:~/code$
[09/22/20]seed@VM:~/code$ ping 192.168.60.100
PING 192.168.60.100 (192.168.60.100) 56(84) bytes of data.
^C
--- 192.168.60.100 ping statistics ---
7 packets transmitted, 0 received, 100% packet loss, time 6111ms
```

图 2.3.2 ping 192.168.60.100

此时 tun.py 没有输出。因为对于此报文不论是源地址还是宿地址，都不属于 192.168.53.0/24 网段，也不是广播报文，报文不会经过 tun0 网卡，所以不会被程序捕捉。

Task 2.d: Write to the TUN Interface

首先修改代码发送 IP 数据包。利用 ping 程序发送请求包，当在 tun0 网卡上捕获到此请求包后，复制其中的负载以构成一个新的 IP 数据包发送出去。设置源地址是 1.2.3.4，目的地址是 192.168.53.99。用 wireshark 抓包测试。

```python
while True:
    packet = os.read(tun, 2048)
    if True:
        ip = IP(packet)
        ip.show()
        newip = IP(src='1.2.3.4', dst=ip.src)
        newpkt = newip/ip.payload
        os.write(tun, bytes(newpkt))
```

图 2.4.1 修改代码

Source	Destination	Protocol	Length	Info
192.168.53.99	192.168.53.100	ICMP	84	Echo (ping) request id=0x22f1,
1.2.3.4	192.168.53.99	ICMP	84	Echo (ping) request id=0x22f1,

图 2.4.2 wireshark 抓包

再次修改代码，不写入完整的 IP 数据报，而是写入任意字节。

```python
while True:
    packet = os.read(tun, 2048)
    if True:
        ip = IP(packet)
        ip.show()
        #newip = IP(src='1.2.3.4', dst=ip.src)
        #newpkt = newip/ip.payload
        #xxx = bytes(newpkt)
        test = b"what"
        os.write(tun,test)
```

图 2.4.3 修改代码

```
Traceback (most recent call last):
  File "tun.py", line 32, in <module>
    os.write(tun,test)
OSError: [Errno 22] Invalid argument
```

图 2.4.4 运行程序

此时无法正常发送数据，write()给出报错信息表示参数非法。应该是传递给网卡的字节串中没有正确的网络地址信息，网卡无法处理，因此抛出异常。

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

```
[09/22/20]seed@VM:~/code$ ls
tun_server.py
[09/22/20]seed@VM:~/code$ ./tun_server.py
192.168.220.129:59406-->0.0.0.0:9090
          Inside: 0.0.0.0-->232.83.169.5
192.168.220.129:59406-->0.0.0.0:9090
          Inside: 0.0.0.0-->232.83.169.5
192.168.220.129:59406-->0.0.0.0:9090
          Inside: 0.0.0.0-->232.83.169.5
192.168.220.129:59406-->0.0.0.0:9090
          Inside: 192.168.53.99-->192.168.53.200
192.168.220.129:59406-->0.0.0.0:9090
          Inside: 192.168.53.99-->192.168.53.200
192.168.220.129:59406-->0.0.0.0:9090
          Inside: 192.168.53.99-->192.168.53.200
192.168.220.129:59406-->0.0.0.0:9090
          Inside: 192.168.53.99-->192.168.53.200
```

图 3.1 ping192.168.53.0/24 主机时服务器终端回显

使用主机 U ping192.168.53.200，由于报文的原地址是 192.168.53.0/24 网段，因此从 tun0 发出，经过隧道发至服务器。服务器接到报文后首先解析出外层路由信息：这是由 192.168.220.129 发往 192.168.220.133 的。然后解析其负载，因为负载实际上是一个 IP 报文，因此又可以解析出其 IP 原宿分别是 192.168.53.99 和 192.168.53.200。

Source	Destination	Protocol	Length	Info
192.168.220.129	10.42.0.128	ICMP	98	Echo (ping) request
192.168.220.129	10.42.0.128	ICMP	98	Echo (ping) request
192.168.220.129	10.42.0.128	ICMP	98	Echo (ping) request
192.168.220.129	10.42.0.128	ICMP	98	Echo (ping) request
Vmware_c0:00:08	Broadcast	ARP	60	Who has 192.168.220

图 3.2 ping192.168.60.200 时在主机 U 上进行抓包观察

使用主机 U ping10.42.0.128，此时服务器终端并没有回显。在主机 U 上抓包，发现报文是通过 ens33 网卡，以 192.168.220.129 的原地址，直接发往 10.42.0.128 宿地址的。而没有路由信息指示这条报文如何转发到 10.42.0.0/24 网段，因此报文没有被处理。

在主机 U 上添加如下路由信息：

```
]seed@VM:~/code$ sudo route add -net 10.42.0.0/24 tun0
```

图 3.3 添加路由信息

指明前往 10.42.0.0/24 网段的报文从 tun0 网卡走出。再次 ping10.42.0.0/24 网段下的主机。

```
[09/22/20]seed@VM:~/code$ ping 10.42.0.101
PING 10.42.0.101 (10.42.0.101) 56(84) bytes of data.
^C
--- 10.42.0.101 ping statistics ---
```

图 3.4 主机 U ping10.42.0.101

```
192.168.220.129:59406-->0.0.0.0:9090
          Inside: 192.168.53.99-->10.42.0.101
192.168.220.129:59406-->0.0.0.0:9090
          Inside: 192.168.53.99-->10.42.0.101
192.168.220.129:59406-->0.0.0.0:9090
          Inside: 192.168.53.99-->10.42.0.101
192.168.220.129:59406-->0.0.0.0:9090
          Inside: 192.168.53.99-->10.42.0.101
```

图 3.5 服务器回显

可见报文成功地通过隧道发送到了服务器并被正确解析。

Task 4: Set Up the VPN Server

修改服务器代码如下：

```python
import fcntl
import struct
import os
import time
from scapy.all import *
from os import write

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

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)

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

os.system("ip addr add 192.168.53.100/24 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(data))
```

图 4.1 服务器代码

即：首先创建虚拟网卡 tun0，并赋予地址 192.168.53.100。当服务器从 socket 上收到数据时，意味着从隧道发来了数据，数据交给 tun0 去处理，因此在循环中把隧道报文的负载写入 tun0 进行处理。

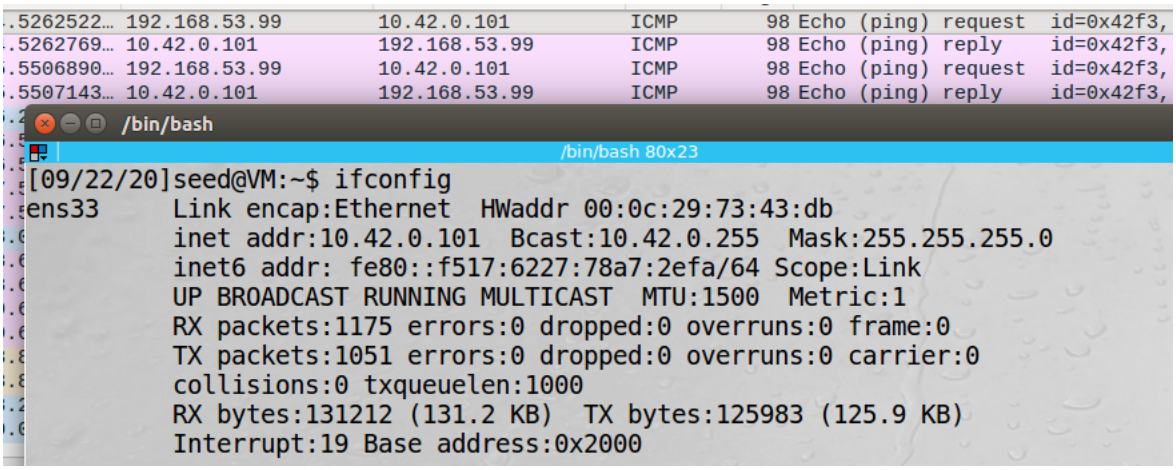

图 4.2 主机 V 抓取数据

在主机 U 上 ping 主机 V，此时可以在主机 V 上收到 ICMP 请求报文。终端的 ifconfig 运行结果证明了图中主机正是主机 V（10.42.0.101）。

Task 5: Handling Traffic in Both Directions

```python
import fcntl
import struct
import os
import time
from scapy.all import *
from os import write

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

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)

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))

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

while True:
    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(bytes(pkt), ("192.168.220.133", 9090))
```

图 5.1 代码主体

对于运行在主机 U 上的 tun_client.py 做如上改动，主要是使程序能够根据实际情况处理 tun 或 sock 上的报文。

```
From tun: 192.168.53.99-->10.42.0.101
From socket: 10.42.0.101-->192.168.53.99
From tun: 192.168.53.99-->10.42.0.101
From socket: 10.42.0.101-->192.168.53.99
From tun: 192.168.53.99-->10.42.0.101
From socket: 10.42.0.101-->192.168.53.99
^CTraceback (most recent call last):
  File "./tun_client.py", line 28, in <module>
    ready, _, _ = select.select([sock, tun], [], [])
KeyboardInterrupt
[09/22/20]seed@VM:~/code$
```

```
8 packets transmitted, 0 received, 100% packet loss, time 7122ms

[09/22/20]seed@VM:~/code$ sudo route add -net 10.42.0.0/24 tun
SIOCADDRT: No such device
[09/22/20]seed@VM:~/code$ sudo route add -net 10.42.0.0/24 tun0
[09/22/20]seed@VM:~/code$ ping 10.42.0.101
PING 10.42.0.101 (10.42.0.101) 56(84) bytes of data.
64 bytes from 10.42.0.101: icmp_seq=1 ttl=63 time=12.4 ms
64 bytes from 10.42.0.101: icmp_seq=2 ttl=63 time=3.45 ms
64 bytes from 10.42.0.101: icmp_seq=3 ttl=63 time=3.26 ms
64 bytes from 10.42.0.101: icmp_seq=4 ttl=63 time=3.37 ms
```

图 5.2 主机 Uping 主机 V

在主机 U 上 ping 主机 V（10.42.0.101），可以看到 ICMP 程序的回显，同时也能在 tun_client 的回显中看到报文在 tun 和 sock 之间的转发。

```
[09/23/20]seed@VM:~$ telnet 10.42.0.101
Trying 10.42.0.101...
Connected to 10.42.0.101.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Thu Sep 17 07:23:38 EDT 2020 from 192.168.220.133 on pts/4
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.

[09/23/20]seed@VM:~$ ifconfig
ens33     Link encap:Ethernet  HWaddr 00:0c:29:73:43:db
          inet addr:10.42.0.101  Bcast:10.42.0.255  Mask:255.255.255.0
          inet6 addr: fe80::f517:6227:78a7:2efa/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:115 errors:0 dropped:0 overruns:0 frame:0
          TX packets:241 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:8633 (8.6 KB)  TX bytes:20637 (20.6 KB)
          Interrupt:19 Base address:0x2000
```

图 5.3 主机 U 通过 telnet 连接主机 V

主机 U 通过 telnet 连接主机 V，可以看到已经成功登录主机 V，运行 ifconfig 命令可以查看主机 V 的 IP 地址，即 10.42.0.101。

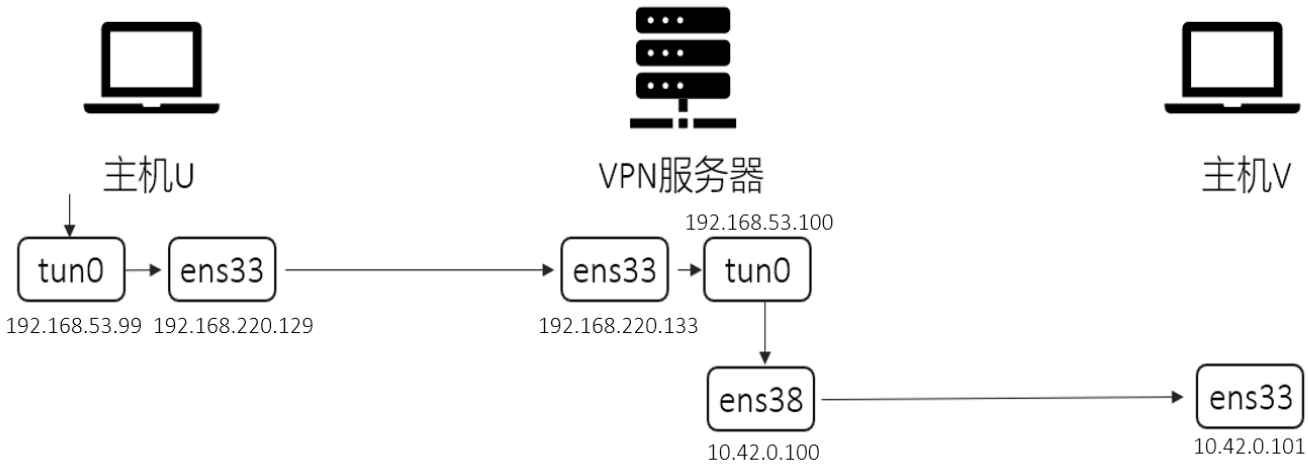

图 5.4 报文路径

由主机 U 发往主机 V 的报文路径如图所示，下面在除主机 V 之外的网卡上抓包分析。主机 V 的 ens33 网卡只是简单的 ICMP 请求和响应，无需过多解释。同时应答报文的路径刚好与上述相反，因此也不赘述。

Source	Destination	Protocol	Length	Info
192.168.53.99	10.42.0.101	ICMP	84	Echo (ping) request
10.42.0.101	192.168.53.99	ICMP	84	Echo (ping) reply

图 5.5 主机 U-tun0

ICMP 请求报文首先从主机 U 的 tun0 发出，以 10.42.0.101 为目的地址，源地址是网卡的地址。

Source	Destination	Protocol	Length	Info
192.168.220.129	192.168.220.133	UDP	126	47418 → 9090 Len=84
192.168.220.133	192.168.220.129	UDP	126	9090 → 47418 Len=84

图 5.6 主机 U-ens33

vpn_client 程序监视 tun0 网卡，一旦发现数据，就把它打包进 UDP 发往 vpn_server。UDP 数据包是通过公网网卡发往服务器公网地址的，因此是 192.168.220.129 发往 192.168.220.133。

192.168.220.129	192.168.220.133	UDP	126	47418 → 9090 Len=84
192.168.220.133	192.168.220.129	UDP	126	9090 → 47418 Len=84

图 5.7 服务器-ens33

服务器通过公网地址接收到主机 U 发来的 UDP 数据包。

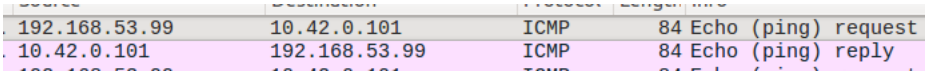

图 5.8 服务器-tun0

vpn_server 程序监测 socket，捕获到数据后就解包并转发到 tun0，因此在 tun0 上出现原始的 ICMP 请求包。

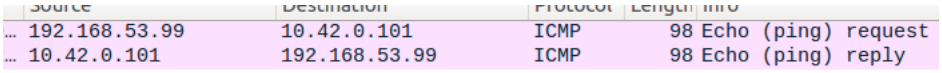

图 5.9 服务器-ens38

根据 ICMP 请求包的宿地址，将相应的包转发到 ens38 网卡并最终发往主机 V。

Task 6: Tunnel-Breaking Experiment

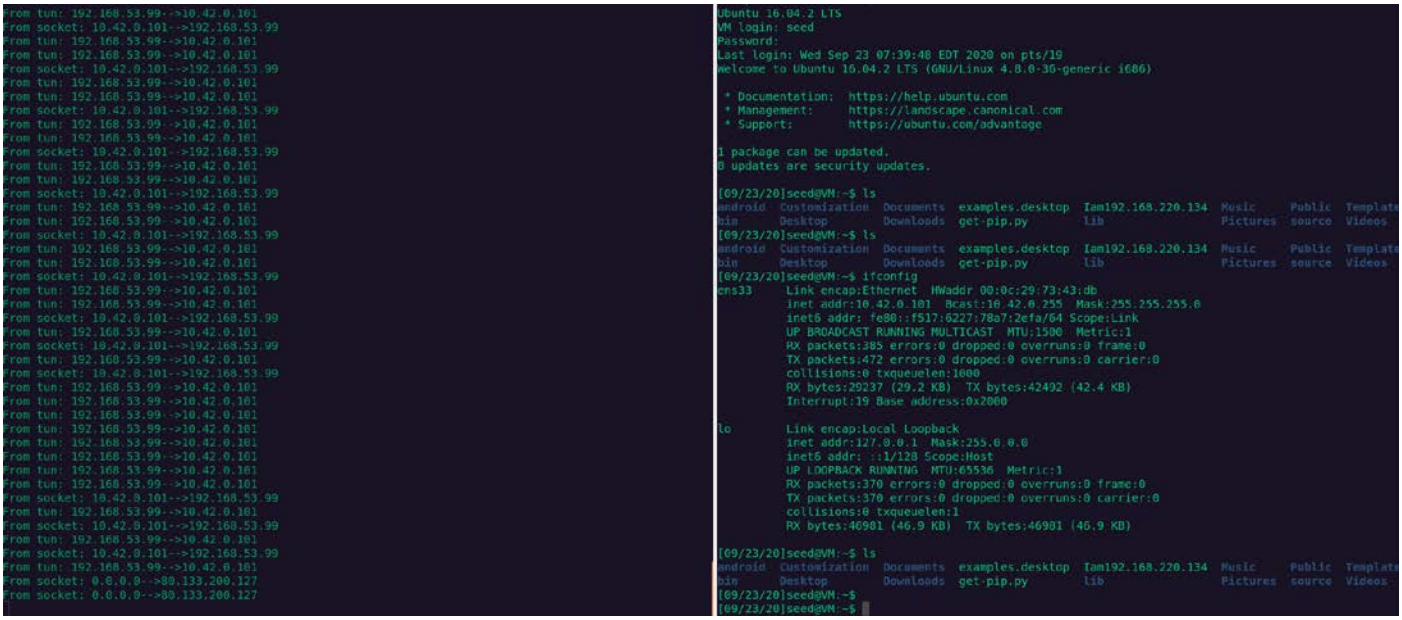

图 6.1 连接中断开隧道

当 telnet 正常连接到主机 V 时，将 vpn_server 中止。此时 telnet 的终端依然存在，但是像是卡死了一样，无法执行命令。输入 ls 命令并回车，终端没有任何变化。通过观察左边 vpn_client 的输出，不断地有 192.168.53.99 至 10.42.0.101 的报文在重传。而当重新运行 vpn_server 之后，之前输入的命令立即都得到了运行。

这是因为虽然 vpn_server 短暂离线了，但是 TCP 连接并没有直接断开。双方仍然在维护当次 TCP 会话，只是数据遭到了阻塞。于是数据全部暂存在发送方的 TCP 缓存中，并根据 TCP 重传协定定时重传。当连接（隧道）恢复，缓存清空，之前键入的数据全部得到了响应。

Task 7: Routing Experiment on Host V

```
[09/23/20]seed@VM:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.42.0.100     0.0.0.0         UG    100    0        0 ens33
10.42.0.0       0.0.0.0         255.255.255.0   U     100    0        0 ens33
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens33
[09/23/20]seed@VM:~$ route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
^C
[09/23/20]seed@VM:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.42.0.100     0.0.0.0         UG    100    0        0 ens33
10.42.0.0       0.0.0.0         255.255.255.0   U     100    0        0 ens33
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens33
[09/23/20]seed@VM:~$ sudo ip route del 0.0.0.0/0
[09/23/20]seed@VM:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.42.0.0       0.0.0.0         255.255.255.0   U     100    0        0 ens33
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens33
[09/23/20]seed@VM:~$ sudo ip route add 192.168.53.0/24 dev ens33 via 10.42.0.100
[09/23/20]seed@VM:~$ sudo ip route add 10.42.0.00/24 dev ens33 via 10.42.0.100
[09/23/20]seed@VM:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.42.0.0       10.42.0.100     255.255.255.0   UG    0      0        0 ens33
10.42.0.0       0.0.0.0         255.255.255.0   U     100    0        0 ens33
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 ens33
192.168.53.0    10.42.0.100     255.255.255.0   UG    0      0        0 ens33
```

图 7.1 修改默认路由

Task 8: Experiment with the TUN IP Address

```
os.system("ip addr add 192.168.60.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))
```

图 8.1 修改脚本

修改运行在主机 U 上的 vpn_client 脚本，改变 tun0 网卡的 IP 地址为 192.168.60.99/24，而服务器的 tun0 网卡 IP 地址仍为 192.168.53.100/24，即隧道两端的网卡不在同一个网段之下。

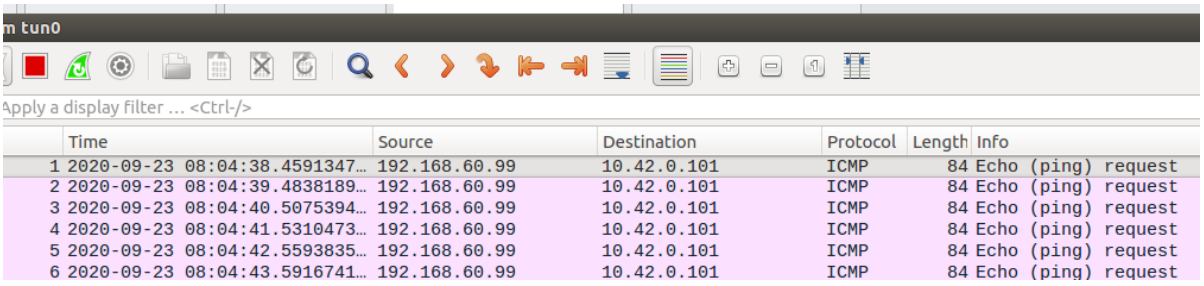

图 8.2 服务器-tun0

图 8.3 服务器-ens38

通过在服务器的网卡上抓包可以看到，ICMP 报文到达了服务器的 tun0 网卡，且源地址是 192.168.60.99，宿地址是 10.42.0.101。这条报文本应该在服务器内部转发到 ens38 网卡，进而发送至主机 V，但是实际并没有转发，而是在服务器内部被丢弃了。

其原因在于服务器开启了反向路径检查。即对于每一个到来的报文，检查其源地址是否匹配当前路由表的最佳路由，如果不匹配则直接丢弃。实验中报文的源地址是 192.168.60.99，而服务器中根本没有指向此网段的路由，所以报文从 tun0 出来之后直接被丢弃。

解决方法是关闭服务器中的反向路径检查，同时设置 192.168.60.0/24 网段指向 tun0 网卡（ICMP reply 报文需要依据此路由信息导入隧道）。

```
[09/23/20]seed@VM:~$ sudo sysctl -w net.ipv4.conf.all.rp_filter=0
net.ipv4.conf.all.rp_filter = 0
[09/23/20]seed@VM:~$ sudo sysctl -w net.ipv4.conf.default.rp_filter=0
net.ipv4.conf.default.rp_filter = 0
[09/23/20]seed@VM:~$ sudo route add -net 192.168.60.0/24 tun0
SIOCADDRT: File exists
```

图 8.4 修改服务器参数

图 8.5 ping 测试成功

Task 9: Experiment with the TAP Interface

```python
import fcntl
import struct
import os
import time
from scapy.all import *
from os import write

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

tap = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack("16sH", b"tap%d", IFF_TAP | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tap, TUNSETIFF, ifr)
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:
    packet = os.read(tap, 2048)
    if True:
        ether = Ether(packet)
        ether.show()
```

图 9.1 创建 tap 虚拟网卡的代码

```
tap0      Link encap:Ethernet  HWaddr 86:68:42:66:9e:46
          inet addr:192.168.53.99  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::8468:42ff:fe66:9e46/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:39 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:4740 (4.7 KB)
```

图 9.2 虚拟网卡 tap 的信息

```
[09/23/20]seed@VM:~$ ping 192.168.53.100
PING 192.168.53.100 (192.168.53.100) 56(84) bytes of data.
From 192.168.53.99 icmp_seq=1 Destination Host Unreachable
From 192.168.53.99 icmp_seq=2 Destination Host Unreachable
From 192.168.53.99 icmp_seq=3 Destination Host Unreachable
```

图 9.3 ping 192.168.53.0/24 网段主机

```
###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 86:68:42:66:9e:46
  type      = ARP
###[ ARP ]###
     hwtype    = 0x1
     ptype     = IPv4
     hwlen     = 6
     plen      = 4
     op        = who-has
     hwsrc     = 86:68:42:66:9e:46
     psrc      = 192.168.53.99
     hwdst     = 00:00:00:00:00:00
     pdst      = 192.168.53.100
```

图 9.4 tap.py 回显

创建虚拟网卡 tap 之后，可以在这张网卡上监听到数据链路层的报文。即报文将会带有 Ethernat 头（以太网环境下）。

实验中测试了 ping 192.168.53.100，因为主机并没有指向此网段的路由，因此对于此 IP 地址发出 ARP 报文尝试解析出 MAC 地址。而实际上没有这样一台主机，因此 ARP 报文无人回应，ping 显示主机不可达。另一边，在 tap.py 的回显中可以完整的看到 ARP 报文，它有一个 Ethernet 头部以及“who-has”类型的 ARP 查询报文体。