Lab 7

VPN Tunneling Lab

Name：程昊

Student Number：57117128

Task1: Network Setup

◆ 实验流程：

这里我们设置两个网段，192.168.64.0/24，用于模拟 Internet，另一个网段 10.0.2.0/24，用于模拟私人网络。对于实验手册中的 Host U，这里我们设置其 IP 地址为 192.168.64.130，Host V 则设置其 IP 地址为 10.0.2.132，VPN 服务器的两个网卡分别连接到两个网段，ens33 的 IP 地址为 192.168.64.131，ens38 的 IP 地址为 10.0.2.129.

下面是 ifconfig 截图：

Host U(这里命名为 Client)：

```
[09/11/20]seed@Client:~$ ifconfig
ens33     Link encap:Ethernet  HWaddr 00:0c:29:40:c7:9a
          inet addr:192.168.64.130  Bcast:192.168.64.255  Mask:255.255.255.0
          inet6 addr: fe80::39ab:5ea4:17e8:49f5/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4933 errors:6 dropped:0 overruns:0 frame:0
          TX packets:2866 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1471478 (1.4 MB)  TX bytes:1106854 (1.1 MB)
          Interrupt:19 Base address:0x2000
```

Host V：

```
[09/11/20]seed@VM:~$ ifconfig
ens39     Link encap:Ethernet  HWaddr 00:0c:29:79:2e:c3
          inet addr:10.0.2.132  Bcast:10.0.2.255  Mask:255.255.255.
          inet6 addr: fe80::1b48:85aa:7f7f:3ecc/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:174 errors:0 dropped:0 overruns:0 frame:0
          TX packets:165 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:21690 (21.6 KB)  TX bytes:13667 (13.6 KB)
          Interrupt:17 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:200 errors:0 dropped:0 overruns:0 frame:0
          TX packets:200 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:30434 (30.4 KB)  TX bytes:30434 (30.4 KB)
```

VPN server：

```
[09/11/20]seed@Server:~$ ifconfig
ens33     Link encap:Ethernet  HWaddr 00:0c:29:be:4f:73
          inet addr:192.168.64.131  Bcast:192.168.64.255  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:5661 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3951 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2650638 (2.6 MB)  TX bytes:526456 (526.4 KB)
          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:63 errors:0 dropped:0 overruns:0 frame:0
          TX packets:72 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:8491 (8.4 KB)  TX bytes:8224 (8.2 KB)
          Interrupt:16 Base address:0x2080
```

接下来做测试，首先是 Host U 与 server 的连通性测试：

```
[09/11/20]seed@Client:~$ ping -c 2 192.168.64.131
PING 192.168.64.131 (192.168.64.131) 56(84) bytes of data.
64 bytes from 192.168.64.131: icmp_seq=1 ttl=64 time=0.547 ms
64 bytes from 192.168.64.131: icmp_seq=2 ttl=64 time=0.482 ms

--- 192.168.64.131 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.482/0.514/0.547/0.039 ms
```

```
[09/11/20]seed@Server:~$ ping -c 2 192.168.64.130
PING 192.168.64.130 (192.168.64.130) 56(84) bytes of data.
64 bytes from 192.168.64.130: icmp_seq=1 ttl=64 time=0.706 ms
64 bytes from 192.168.64.130: icmp_seq=2 ttl=64 time=0.796 ms

--- 192.168.64.130 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1012ms
rtt min/avg/max/mdev = 0.706/0.751/0.796/0.045 ms
```

接着是 Host V 与 server 的连通性测试：

```
[09/11/20]seed@HostV:~$ ping -c 2 10.0.2.129
PING 10.0.2.129 (10.0.2.129) 56(84) bytes of data.
64 bytes from 10.0.2.129: icmp_seq=1 ttl=64 time=0.560 ms
64 bytes from 10.0.2.129: icmp_seq=2 ttl=64 time=0.586 ms

--- 10.0.2.129 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1010ms
rtt min/avg/max/mdev = 0.560/0.573/0.586/0.013 ms
```

```
[09/11/20]seed@Server:~$ ping 10.0.2.132
PING 10.0.2.132 (10.0.2.132) 56(84) bytes of data.
64 bytes from 10.0.2.132: icmp_seq=1 ttl=64 time=0.566 ms
64 bytes from 10.0.2.132: icmp_seq=2 ttl=64 time=0.570 ms
64 bytes from 10.0.2.132: icmp_seq=3 ttl=64 time=0.646 ms
64 bytes from 10.0.2.132: icmp_seq=4 ttl=64 time=0.573 ms
```

最后是 Host U 与 Host V 之间的连通性测试：

```
[09/11/20]seed@VM:~$ ping -c 5 10.0.2.132
PING 10.0.2.132 (10.0.2.132) 56(84) bytes of data.
^C
--- 10.0.2.132 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4085ms
```

这是设置了静态路由的结果，且 Server 没有开启路由转发功能。也可以选择不设置静态路由，这样目标网络会变得不可达。

◆ 实验结论：

VPN tunneling lab 的实验环境已经配置完毕，此时假设 Host U(192.168.64.130)位于 Internet，而 Host V(10.0.2.130)则是一台内网主机。目标是 U 通过 VPN server 成功连通内网中的 V。

Task2: Create and Configure TUN Interface

◆ 实验流程：

运行 tun.py，然后在另一个终端里执行 ip address 命令，可以看到如下结果：

```
3: tun0: <POINTOPOINT,MULTICAST,NOARP> mtu 1500 qdisc noop state DOWN group defa
ult qlen 500
    link/none
```

当然我们之后将 tun0 改为 hao0.

运行以下两个命令：

```
[09/11/20]seed@VM:~$ sudo ip addr add 192.168.53.99/24 dev hao0
[09/11/20]seed@VM:~$ sudo ip link set dev hao0 up
```

再次利用 ip address 查看：

```
5: hao0: <POINTOPOINT,MULTICAST,NOARP,UP,LOWER_UP> mtu 1500 qdisc pfifo_
te UNKNOWN group default qlen 500
    link/none
    inet 192.168.53.99/24 scope global hao0
       valid_lft forever preferred_lft forever
    inet6 fe80::cea:9208:4027:b393/64 scope link flags 800
       valid_lft forever preferred_lft forever
```

ifconfig：

```
hao0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
-00
          inet addr:192.168.53.99  P-t-P:192.168.53.99  Mask:255.255.255.0
          inet6 addr: fe80::cea:9208:4027:b393/64 Scope:Link
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

这说明这个接口已经被指派了一个 IP 地址，并且处于开启状态了。

修改 tun.py 后，接下来我们尝试 ping 192.168.53.24：

```
###[ IP ]###
  version   = 4
  ihl       = 5
  tos       = 0x0
  len       = 84
  id        = 50082
  flags     = DF
  frag      = 0
  ttl       = 64
  proto     = icmp
  chksum    = 0x8b3a
  src       = 192.168.53.99
  dst       = 192.168.53.24
  \options   \
###[ ICMP ]###
     type      = echo-request
     code      = 0
     chksum    = 0xd140
     id        = 0x195e
     seq       = 0x1
###[ Raw ]###
        load      = '\xff\x16\\_\xc5\xe6\x01\x00\x08\t\n\x0b\x0c\r\x0e\x0f\x10\
11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./0123
```

Ping 10.0.2.0/24 的 ip 地址则无法 ping 通。

接下来我再次修改文件，尝试构造一个假包，看看是否能从这个虚拟接口发出：

```
1 2020-09-11 20:58:08.7051031… 192.168.53.99   192.168.53.24   ICMP   84 Echo (ping) request  id=0x1afb, seq=1/256, ttl=64 (no response found!)
2 2020-09-11 20:58:08.7076256… 1.2.3.4         192.168.53.99   ICMP   84 Echo (ping) request  id=0x1afb, seq=1/256, ttl=64 (no response found!)
```

可以看到假包被构造成功，并且成功从 tun0 接口发出。

按照实验要求，我们再尝试向接口写入任意字符：

```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, b'lab vpn')
```

```
1 2020-09-11 21:02:17.2260992… 192.168.53.99   192.168.53.24   ICMP   84 Echo (ping) request  id=0x1bfc, seq=1/256, ttl=64 (no response found!)
2 2020-09-11 21:02:17.2283206… N/A             N/A             N/A     7 Raw packet data[Malformed Packet]
```

```
00  6c 61 62 20 76 70 6e                              lab vpn
```

Wireshark 无法分析这个包，这个包的格式显然是错误的，根本不能称之为包。

◆ 实验结论：

当尝试 ping 一个 192.168.53.0/24 的 ip 地址时，内核会自动把包转发至 tun0 接口(因为 tun0 接口属于这个子网)，因此我们可以从/dev/net/tun 中读取到 IP 包。由于设置原因在这里无法 ping 通 10.0.2.0 的 ip(目标网络不可达)，

在这里就不做分析了。

接口相当于一个文件，对这个文件进行读入或写出相当于发送 pkt 和接受 pkt 的动作。

Task3: Send the IP Packet to VPN Server Through a Tunnel

◆ 实验流程：

首先修改 tun_client.py 程序和 tun_server.py 程序：

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
while True:
        packet = os.read(tun, 2048)
        if True: # Send the packet via the tunnel
                sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

```
#!/usr/bin/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))
```

接下来从主机 Host U 上 ping 192.168.53.24，查看 VPN_SERVER：

```
[09/11/20]seed@VM:~$ ping 192.168.53.24
PING 192.168.53.24 (192.168.53.24) 56(84) bytes of data.
^C
```

```
192.168.64.130:52909 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.24
192.168.64.130:52909 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.24
192.168.64.130:52909 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 192.168.53.24
```

符合预期。这是因为 ping 的目标地址属于 192.168.53.0/24，因此系统的路由表会自动将 ICMP 数据包传递给 tun0 接口，客户端程序则可以从 /dev/net/tun 中读取得到 ICMP 数据包，然后将其封装发往 Server.

我们先添加一条路由规则：

```
[09/11/20]seed@VM:~$ sudo ip route add 10.0.2.0/24 dev hao0
```

这条路由规则是告诉内核，将目的 IP 为 10.0.2.0/24 子网中的 IP，通过 hao0(tun0)接口转发。

此时我们尝试 telnet 10.0.2.122：

```
192.168.64.130:52909 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 10.0.2.122
192.168.64.130:52909 --> 0.0.0.0:9090
 Inside: 192.168.53.99 --> 10.0.2.122
```

可以看到服务端成功捕获到了这个 TCP Packet.

◆ 实验结论：

具体见上文分析。这里主要说明了 VPN 隧道的用处。

Task4: Setup the VPN Server

◆ 实验流程：

首先设置路由转发功能开启：

```
[09/11/20]seed@Server:~$ sudo sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
```

然后修改 tun_server.py 文件，主要是添加配置 tun0 以及包的写入：

```python
# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'hao%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 192.168.53.1/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))

while True:
        data, (ip, port) = sock.recvfrom(2048)
        os.write(tun,data)
```

然后在 Host U 和 VPN Server 上分别运行 tun_client.py 和 tun_server.py

程序，尝试 ping Host V：

以下是 V 上的 wireshark 截图，可以看到通过 tun 隧道，ICMP 的 ping 报文成功被转发至本来无法联通的 Host V 上。注意要在 Host V 上配置默认路由，否则会产生 no response found 的问题。

1	2020-09-11 23:30:02.1066584…	192.168.53.99	10.0.2.132	ICMP	98	Echo (ping) request	id=0x1c60, seq=1/256, ttl=63 (reply in 2)
2	2020-09-11 23:30:02.1066880…	10.0.2.132	192.168.53.99	ICMP	98	Echo (ping) reply	id=0x1c60, seq=1/256, ttl=64 (request in 1)
3	2020-09-11 23:30:03.1231560…	192.168.53.99	10.0.2.132	ICMP	98	Echo (ping) request	id=0x1c60, seq=2/512, ttl=63 (reply in 4)
4	2020-09-11 23:30:03.1231838…	10.0.2.132	192.168.53.99	ICMP	98	Echo (ping) reply	id=0x1c60, seq=2/512, ttl=64 (request in 3)
5	2020-09-11 23:30:04.1473290…	192.168.53.99	10.0.2.132	ICMP	98	Echo (ping) request	id=0x1c60, seq=3/768, ttl=63 (reply in 6)
6	2020-09-11 23:30:04.1473594…	10.0.2.132	192.168.53.99	ICMP	98	Echo (ping) reply	id=0x1c60, seq=3/768, ttl=64 (request in 5)
7	2020-09-11 23:30:05.1710261…	192.168.53.99	10.0.2.132	ICMP	98	Echo (ping) request	id=0x1c60, seq=4/1024, ttl=63 (reply in 8)
8	2020-09-11 23:30:05.1710551…	10.0.2.132	192.168.53.99	ICMP	98	Echo (ping) reply	id=0x1c60, seq=4/1024, ttl=64 (request in 7)
9	2020-09-11 23:30:06.1955257…	192.168.53.99	10.0.2.132	ICMP	98	Echo (ping) request	id=0x1c60, seq=5/1280, ttl=63 (reply in 10)
10	2020-09-11 23:30:06.1955511…	10.0.2.132	192.168.53.99	ICMP	98	Echo (ping) reply	id=0x1c60, seq=5/1280, ttl=64 (request in 9)

Task5: Handling Traffic in Both Directions

◆ 实验流程：

修改 Client 和 Server 的 VPN 程序：

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
while True:
        #packet = os.read(tun, 2048)
        #if True: # Send the packet via the tunnel
        #       sock.sendto(packet, (SERVER_IP, SERVER_PORT))
        ready, _, _ = select([sock, tun], [], [])
        for fd in ready:
                if fd is sock:
                        data, (ip, port) = sock.recvfrom(2048)
                        os.write(tun,data)
                if fd is tun:
                        packet = os.read(tun, 2048)
                        sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

```
ip = "192.168.64.130"
port = 10000

while True:
        ready, _, _ = select([sock, tun], [], [])
        for fd in ready:
                if fd is sock:
                        data, (ip, port) = sock.recvfrom(2048)
                        os.write(tun,data)
                if fd is tun:
                        packet = os.read(tun, 2048)
                        sock.sendto(packet, (ip, port))
```

尝试 ping 10.0.2.132：

```
[09/11/20]seed@VM:~$ ping 10.0.2.132
PING 10.0.2.132 (10.0.2.132) 56(84) bytes of data.
64 bytes from 10.0.2.132: icmp_seq=1 ttl=63 time=2.84 ms
64 bytes from 10.0.2.132: icmp_seq=2 ttl=63 time=1.59 ms
64 bytes from 10.0.2.132: icmp_seq=3 ttl=63 time=1.51 ms
64 bytes from 10.0.2.132: icmp_seq=4 ttl=63 time=1.62 ms
```

telnet：

```
[09/12/20]seed@VM:~$ telnet 10.0.2.132
Trying 10.0.2.132...
Connected to 10.0.2.132.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Fri Sep 11 16:13:24 EDT 2020 from 10.0.2.1 on pts/2
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.
```

◆ 实验结论：

Client 和 Server 的 VPN 程序比较镜像。这里说明一下整个流量的走向。当一个 pkt 从 Host U 发出后，由于路由规则的设定，这个包将从 tun0 接口发出，此时 src IP 变成了 tun0 接口的 ip.tun0 接口的一端连接着内核的网络协议栈，另一端连接着电缆(对于虚拟接口而言这里是软件程序)，也就是 tun0 的另一端连接着 tun.py 程序，我们可以通过 read 方式读取这个 pkt，接着将这个 pkt 封装到一个 udp 数据包中，发往 VPN Server。注意这个 UDP 数据包的 src IP 和 dst IP 是 Host U 的 IP 和 VPN Server 的 IP。到达服务器之后将 UDP 的负载提取出来，这是我们当时在 Host U 的内核中构造的 pkt，再把这个 pkt 从 tun0 接口写入，然后由内核处理，通过真实的网卡进行转发，到达了私人网络中的 Host V。Host V 做出应答，按照原路由返回，由 VPN Server 的 ens38 接口返回到 VPN Server，然后内核根据这个 reply pkt 的目的 IP，选择 tun0 接口进行转发，所以 reply pkt 又回到了 tun_server.py 程序中，由 read 函数读出，再次进行 udp 封装发回到 Client。最后数据则由 Client 获得。

Task6: Tunnel-Breaking Experiment

◆ 实验流程：

我们首先连接上 telnet：

```
[09/12/20]seed@VM:~$ telnet 10.0.2.132
Trying 10.0.2.132...
Connected to 10.0.2.132.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VM login: seed
Password:
Last login: Sat Sep 12 00:19:54 EDT 2020 on pts/1
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.
```

然后先输入两个字符 if：

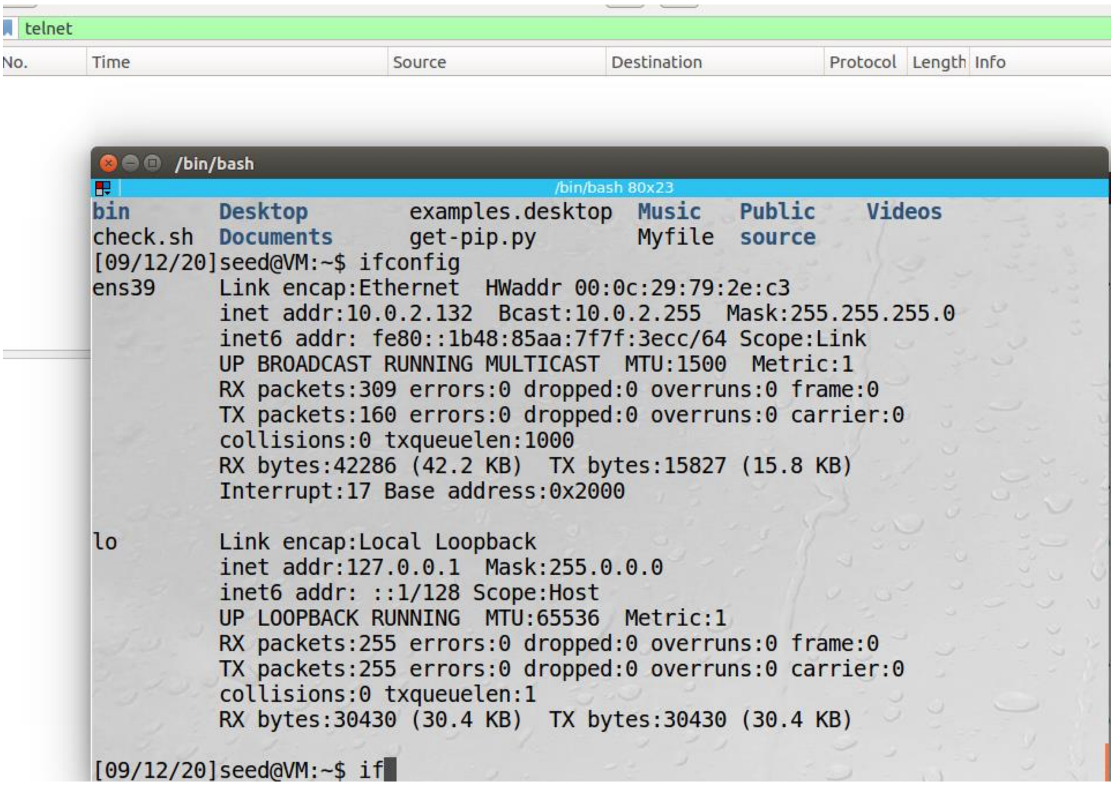
```
[09/12/20]seed@VM:~$ if
```

查看 wireshark：

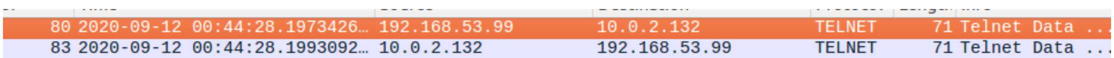
5	2020-09-12 00:36:43.8104157…	192.168.53.99	10.0.2.132	TELNET	69	Telnet Data ...
8	2020-09-12 00:36:43.8122792…	10.0.2.132	192.168.53.99	TELNET	69	Telnet Data ...
9	2020-09-12 00:36:43.8122957…	192.168.53.99	10.0.2.132	TCP	68	34604 → 23 [ACK] Seq=228126452 Ack
11	2020-09-12 00:36:44.2351370…	192.168.53.99	10.0.2.132	TELNET	69	Telnet Data ...
14	2020-09-12 00:36:44.2370607…	10.0.2.132	192.168.53.99	TELNET	69	Telnet Data ...
15	2020-09-12 00:36:44.2370761…	192.168.53.99	10.0.2.132	TCP	68	34604 → 23 [ACK] Seq=228126453 Ack

```
▸ Frame 11: 69 bytes on wire (552 bits), 69 bytes captured (552 bits) on interface 0
▸ Linux cooked capture
▸ Internet Protocol Version 4, Src: 192.168.53.99, Dst: 10.0.2.132
▸ Transmission Control Protocol, Src Port: 34604, Dst Port: 23, Seq: 228126452, Ack: 1858912981, Len: 1
▾ Telnet
    Data: f
```

可以看到 i 和 f 都已经完整的键入了，接下来断开客户端的 VPN，输入 ab：



这个图主要描述了，当中止 client 程序之后，从键盘输入 a 和 b 之后，客户端不仅没有正常发出 telnet TCP pkt，所以更加不可能显示字符，我们重新开启 tun_client.py 程序，然后再输入 1 个 c：



可以看到，之前输入的 a 和 b 已经新输入的 c 都同时出现了。

◆ 实验结论：

telnet 向对端发起连接时，用的并不是自己的真实 IP，而是 tun0 所设置的 IP，所以当 Client 的 VPN 程序结束时，telnet 无法正常构造 TCP 数据包发出，因为 IP 和 tun0 都不存在了，键入的 a 和 b 只能存在缓冲区。当重新开启 Client 程序后，再次键入下一个字符 c，此时 telnet 的客户端再次尝试将数据发往对端，此时发现可以成功发出 TCP 数据包，便将缓冲区的所有内容一并发往对端。(描述不是非常严谨，大概是这个流程，因为我也不了解 telnet)。

Task7: Routing Experiment on Host V

◆ 实验流程：

与 VirutalBox 不同，VMware 下如果是 host only 模式默认路由是不存在的，前面的实验都是我自己设置了默认将所有报文转发至 VPN 服务器的，而且这个默认路由 TTL 较短。这次我们添加一个具体的路由转发条目：

```
sudo ip route add 192.168.53.0/24 via 10.0.2.129 dev ens39
```

```
 6 2020-09-12 00:51:13.5823335… 192.168.53.99   10.0.2.132      ICMP   98 Echo (ping) request  id=0x2298, seq=2/512, ttl=63 (no response found
 7 2020-09-12 00:51:14.6074229… 192.168.53.99   10.0.2.132      ICMP   98 Echo (ping) request  id=0x2298, seq=3/768, ttl=63 (no response found
 8 2020-09-12 00:51:15.6310015… 192.168.53.99   10.0.2.132      ICMP   98 Echo (ping) request  id=0x2298, seq=4/1024, ttl=63 (no response four
21 2020-09-12 00:55:01.8962457… 192.168.53.99   10.0.2.132      ICMP   98 Echo (ping) request  id=0x22ae, seq=1/256, ttl=63 (reply in 22)
22 2020-09-12 00:55:01.8962740… 10.0.2.132      192.168.53.99   ICMP   98 Echo (ping) reply    id=0x22ae, seq=1/256, ttl=64 (request in 21)
23 2020-09-12 00:55:02.8988099… 192.168.53.99   10.0.2.132      ICMP   98 Echo (ping) request  id=0x22ae, seq=2/512, ttl=63 (reply in 24)
24 2020-09-12 00:55:02.8988382… 10.0.2.132      192.168.53.99   ICMP   98 Echo (ping) reply    id=0x22ae, seq=2/512, ttl=64 (request in 23)
25 2020-09-12 00:55:03.9003856… 192.168.53.99   10.0.2.132      ICMP   98 Echo (ping) request  id=0x22ae, seq=3/768, ttl=63 (reply in 26)
```

可以看到，在设置具体路由之前，ping 的结果都是 no response found，这是因为 Host V 在 host only 模式下，缺乏默认路由，所以认为 192.168.53.0/24(或者任何非本 LAN 的子网)的状态均是网络不可达。在设置了具体的路由转发规则之后，才生成了正常的回复报文。

◆ 实验结论：

设置路由表项的核心在于将 tun 接口所在的子网准确路由至 VPN server。

Task8: Experiment with the TUN IP Address

◆ 实验流程：

理论判断，根据反向路径过滤策略，包应该会在写入 Server 的 tun0 接口时被丢弃。因为包的源 IP 为 192.168.30.99，而 tun0 接口的 IP 为 192.168.53.1，根据反向路径过滤策略来说，对调包的源 IP 和目的 IP，输出接口不是 tun0(没有这条路由规则指明如果目的 IP 为 192.168.30.0/24 时应该从哪个口转发)。

尝试做实验证明一下，我们首先修改 Client 的 tun0 ip：

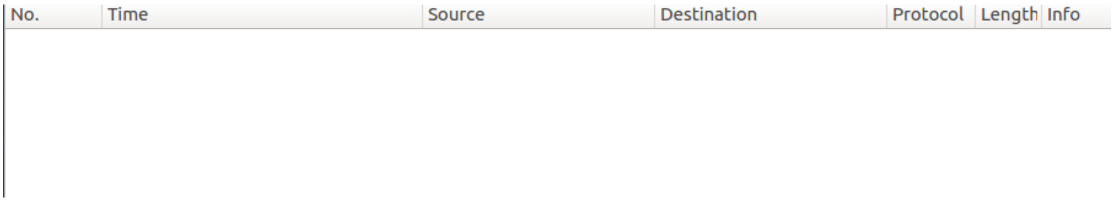
```python
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))
os.system("ip addr add 192.168.30.99/24 dev {}".format(ifname))
os.system("ip link set dev {} up".format(ifname))
os.system("ip route add 10.0.2.0/24 dev {}".format(ifname))

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

同时首先使用 WireShark 监控 Server 的 tun0 接口、ens38 接口，和 Host V 的 ens39 接口，看看数据包是否正常转发：

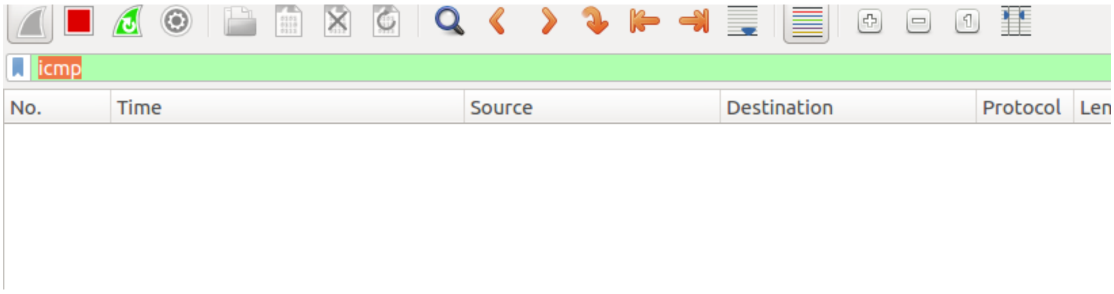
这里是 tun0 接口的 wireshark 监控结果：

No.	Time	Source	Destination	Protocol	Length	Info
1	2020-09-12 02:06:39.9593396…	192.168.30.99	10.0.2.132	ICMP	84	Echo (ping) request id=0x2acb, seq=1/256, ttl=64 (no response found!)
2	2020-09-12 02:06:40.9626851…	192.168.30.99	10.0.2.132	ICMP	84	Echo (ping) request id=0x2acb, seq=2/512, ttl=64 (no response found!)
3	2020-09-12 02:06:41.9865548…	192.168.30.99	10.0.2.132	ICMP	84	Echo (ping) request id=0x2acb, seq=3/768, ttl=64 (no response found!)
4	2020-09-12 02:06:43.0104782…	192.168.30.99	10.0.2.132	ICMP	84	Echo (ping) request id=0x2acb, seq=4/1024, ttl=64 (no response found!)

这里是 ens39 接口的 wireshark 监控结果：



说明 VPN Server 并没有把数据包转发至 Host V，我们再看一下 VPN Server 的 ens38 接口，这个接口属于子网 10.0.2.0/24：



同样什么也没有。这我们利用 write 语句写入 VPN Server 的 tun0 接口的包，被内核丢弃了，这和我们的理论判断一样。

解决方法，可以尝试在 VPN Server 上加一条路由，将目的 IP 为 192.168.30.0/24 的数据包通过 Server 的 tun0 接口发出：

```
[09/12/20]seed@Server:~$ sudo ip route add 192.168.30.0/24 dev hao0
[09/12/20]seed@Server:~$ ip route
10.0.2.0/24 dev ens38  proto kernel  scope link  src 10.0.2.129  metric 100
169.254.0.0/16 dev ens33  scope link  metric 1000
192.168.30.0/24 dev hao0  scope link
192.168.53.0/24 dev hao0  proto kernel  scope link  src 192.168.53.1
192.168.64.0/24 dev ens33  proto kernel  scope link  src 192.168.64.131  metric
100
```

再次尝试 ping：

```
[09/12/20]seed@VM:~$ ping 10.0.2.132
PING 10.0.2.132 (10.0.2.132) 56(84) bytes of data.
64 bytes from 10.0.2.132: icmp_seq=1 ttl=63 time=1.54 ms
64 bytes from 10.0.2.132: icmp_seq=2 ttl=63 time=1.68 ms
64 bytes from 10.0.2.132: icmp_seq=3 ttl=63 time=1.68 ms
```

看下 VPN server 的 ens38 和 Host V 的抓包结果：

Ens38：

1	2020-09-12 02:14:02.5550571…	192.168.30.99	10.0.2.132	ICMP	98	Echo (ping) request id=0x2b29, seq=1/256, ttl=63 (reply in 2)
2	2020-09-12 02:14:02.5556068…	10.0.2.132	192.168.30.99	ICMP	98	Echo (ping) reply id=0x2b29, seq=1/256, ttl=64 (request in 1)
3	2020-09-12 02:14:03.5567761…	192.168.30.99	10.0.2.132	ICMP	98	Echo (ping) request id=0x2b29, seq=2/512, ttl=63 (reply in 4)
4	2020-09-12 02:14:03.5574128…	10.0.2.132	192.168.30.99	ICMP	98	Echo (ping) reply id=0x2b29, seq=2/512, ttl=64 (request in 3)
5	2020-09-12 02:14:04.5589171…	192.168.30.99	10.0.2.132	ICMP	98	Echo (ping) request id=0x2b29, seq=3/768, ttl=63 (reply in 6)
6	2020-09-12 02:14:04.5594892…	10.0.2.132	192.168.30.99	ICMP	98	Echo (ping) reply id=0x2b29, seq=3/768, ttl=64 (request in 5)

Host V 的 ens39：

o.	Time	Source	Destination	Protocol	Length	Info
5	2020-09-12 07:16:16.5196340…	192.168.30.99	10.0.2.132	ICMP	98	Echo (ping) request id=0x2b29, seq=1/256, ttl=63 (reply in 6)
6	2020-09-12 07:16:16.5196645…	10.0.2.132	192.168.30.99	ICMP	98	Echo (ping) reply id=0x2b29, seq=1/256, ttl=64 (request in 5)
7	2020-09-12 07:16:17.5213751…	192.168.30.99	10.0.2.132	ICMP	98	Echo (ping) request id=0x2b29, seq=2/512, ttl=63 (reply in 8)
8	2020-09-12 07:16:17.5214016…	10.0.2.132	192.168.30.99	ICMP	98	Echo (ping) reply id=0x2b29, seq=2/512, ttl=64 (request in 7)
9	2020-09-12 07:16:18.5235035…	192.168.30.99	10.0.2.132	ICMP	98	Echo (ping) request id=0x2b29, seq=3/768, ttl=63 (reply in 10)
10	2020-09-12 07:16:18.5235337…	10.0.2.132	192.168.30.99	ICMP	98	Echo (ping) reply id=0x2b29, seq=3/768, ttl=64 (request in 9)

这说明这条路由规则确实打破了反向路径过滤。

◆ 实验结论：

反向路径过滤在 VPN 的 IP 隧道中也是生效了。这更加说明对于内核来说接口就是个文件而已。

Task9: Experiment with the TAP Interface

◆ 实验流程：

按照要求编写 tap.py 程序，用于创建 tap 接口：

```
# Create the tun interface
tun = os.open("/dev/net/tun", os.O_RDWR)
ifr = struct.pack('16sH', b'hao%d', IFF_TAP | IFF_NO_PI)
ifname_bytes = fcntl.ioctl(tun, TUNSETIFF, ifr)
print("stop here")
# 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 10.0.2.0/24 dev {}".format(ifname))


while True:
        packet = os.read(tap, 2048)
        if True:
                ether = Ether(packet)
                ether.show()
```

运行这个 python 程序，然后 ping 192.168.53.39：

```
[09/12/20]seed@VM:~$ ping 192.168.53.39
PING 192.168.53.39 (192.168.53.39) 56(84) bytes of data.
```

```
     hwlen      = 6
     plen       = 4
     op         = who-has
     hwsrc      = 96:a8:14:76:36:1b
     psrc       = 192.168.53.99
     hwdst      = 00:00:00:00:00:00
     pdst       = 192.168.53.39

###[ Ethernet ]###
  dst       = ff:ff:ff:ff:ff:ff
  src       = 96:a8:14:76:36:1b
  type      = 0x806
###[ ARP ]###
     hwtype     = 0x1
     ptype      = 0x800
     hwlen      = 6
     plen       = 4
     op         = who-has
     hwsrc      = 96:a8:14:76:36:1b
     psrc       = 192.168.53.99
     hwdst      = 00:00:00:00:00:00
     pdst       = 192.168.53.39
```

查看 ifconfig 命令：

```
hao0      Link encap:Ethernet  HWaddr 96:a8:14:76:36:1b
          inet addr:192.168.53.99  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::94a8:14ff:fe76:361b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:68 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:8072 (8.0 KB)
```

这说明了 tap 接口确实延展至了 Mac 层。因为在 ping 192.168.53.39 时，系统的 ARP 缓存里没有这个地址，所以会发出 ARP 广播，也就是上图所示的以太网报头的 dst mac（全 f 代表广播），源 mac 则是 hao0 本身。

Virtual Private Network (VPN) Lab

Task1: VM Setup

◆ 实验流程：

与上一个实验相同。只是本次实验中用到的是 C 编写的程序，原理和内容不变。

Task2: Creating a VPN Tunnel using TUN/TAP

◆ 实验流程：

与上一个实验相同。只是本次实验中用到的是 C 编写的程序，原理和内容不变。

Task3: Encrypting the Tunnel

◆ 实验流程：

这里并没有在 TLS 基础上编写 Minivpn，主要关注点在 TLS 的编程，IP 隧道加密的过程。

从本实验的 website 上下载 tls.zip，解压然后编译：

```
[09/12/20]seed@VM:~/.../lab7$ cd tls
[09/12/20]seed@VM:~/.../tls$ make
gcc -o tlsclient tlsclient.c -lssl -lcrypto
gcc -o tlsserver tlsserver.c -lssl -lcrypto
```

```
[09/12/20]seed@Server:~/Desktop$ cd tls
[09/12/20]seed@Server:~/.../tls$ make
gcc -o tlsclient tlsclient.c -lssl -lcrypto
gcc -o tlsserver tlsserver.c -lssl -lcrypto
[09/12/20]seed@Server:~/.../tls$
```

但是有个问题，这个 CA-Server(vpnlabserver)的证书过期了：

vpnlabserver.com

Identity: vpnlabserver.com

Verified by: seedlabca.com

Expires: 03/16/2019

‣ **Details**

我们修改一下系统时间，以便实验正常进行：

```
[09/12/20]seed@VM:~/.../tls$ sudo date -s "20190101 13:30:25"
Tue Jan  1 13:30:25 EST 2019
```

```
[01/01/19]seed@VM:~/.../tls$ sudo ./tlsclient vpnlabserver.com 4433
SSL connection is successful
SSL connection using AES256-GCM-SHA384
HTTP/1.1 200 OK
Content-Type: text/html

<!DOCTYPE html><html><head><title>Hello World</title></head><style>body {backgr
und-color: black}h1 {font-size:3cm; text-align: center; color: white;text-shado
: 0 0 3mm yellow}</style></head><body><h1>Hello, world!</h1></body></html>
```

```
[01/01/19]seed@VM:~/.../tls$ sudo ./tlsserver
SSL connection established!
Received: GET / HTTP/1.1
Host: vpnlabserver.com
```

Task4: Authenticating the VPN Server

◆ 实验流程：

首先看一下源码，判断三个验证分别是在代码的哪里实现的：

```
42    SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
43    if(SSL_CTX_load_verify_locations(ctx,NULL, CA_DIR) < 1){
44     printf("Error setting the verify locations. \n");
45     exit(0);
46    }
47    ssl = SSL_new (ctx);
48
49    X509_VERIFY_PARAM *vpm = SSL_get0_param(ssl);
50    X509_VERIFY_PARAM_set1_host(vpm, hostname, 0);
```

这是 TLS Client 的 TLS 连接配置函数，第 42 行和 43 行完成了第一个步的验证，加载证书并且判断证书是否有效；第 49 行则判断了 Server 是否是证书的

拥有者，第 50 行则是判断了服务器是否是为真(ip 是否与 hostname 对的上)。

接下来尝试构建一个我们自己的 VPN Server name，并为其构建 CA 证书。

Step 1: Becoming CA

我们为我们的 CA 生成一个自签名证书。这意味着这个 CA 是完全可信的，它的证书将作为根证书。该命令的输出是:CA 的私钥和 CA 的公钥证书。

```
[09/12/20]seed@VM:~/.../myca$ openssl req -new -x509 -keyout ca.key -out ca.crt
-config openssl.cnf
error on line -1 of openssl.cnf
3071174336:error:02001002:system library:fopen:No such file or directory:bss_fil
e.c:175:fopen('openssl.cnf','rb')
3071174336:error:2006D080:BIO routines:BIO_new_file:no such file:bss_file.c:178:
3071174336:error:0E078072:configuration file routines:DEF_LOAD:no such file:conf
_def.c:195:
[09/12/20]seed@VM:~/.../myca$ openssl req -new -x509 -keyout ca.key -out ca.crt
-config openssl.cnf
Generating a 1024 bit RSA private key
..............................................................................
....................................++++++
.............++++++
writing new private key to 'ca.key'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:CN
State or Province Name (full name) [Some-State]:JS
Locality Name (eg, city) []:NJ
Organization Name (eg, company) [Internet Widgits Pty Ltd]:SEU
Organizational Unit Name (eg, section) []:CS
Common Name (e.g. server FQDN or YOUR name) []:haovpn
Email Address []:haochworktime@gmail.com
```

Step 2: Creating a Certificate for vpnhao.com

接下来为我们的服务器域名 vpnhao.com 生成证书。首先生成 Server 的公钥与私钥对：

```
[09/12/20]seed@VM:~/.../myca$ openssl genrsa -aes128 -out server.key 1024
Generating RSA private key, 1024 bit long modulus
....................................++++++
.........++++++
e is 65537 (0x10001)
Enter pass phrase for server.key:
Verifying - Enter pass phrase for server.key:
```

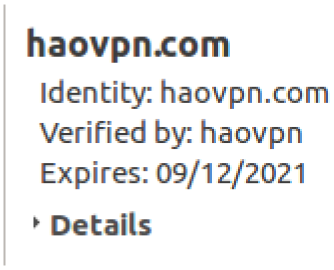
接下来再生成 Certificate Signing Request：

```
[09/12/20]seed@VM:~/.../myca$ openssl req -new -key server.key -out server.csr -
config openssl.cnf
Enter pass phrase for server.key:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:CN
State or Province Name (full name) [Some-State]:JS
Locality Name (eg, city) []:NJ
Organization Name (eg, company) [Internet Widgits Pty Ltd]:SEU
Organizational Unit Name (eg, section) []:CS
Common Name (e.g. server FQDN or YOUR name) []:haovpn.com
Email Address []:haochworktime@gmail.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:hao
string is too short, it needs to be at least 4 bytes long
A challenge password []:haovpn
An optional company name []:hao
```

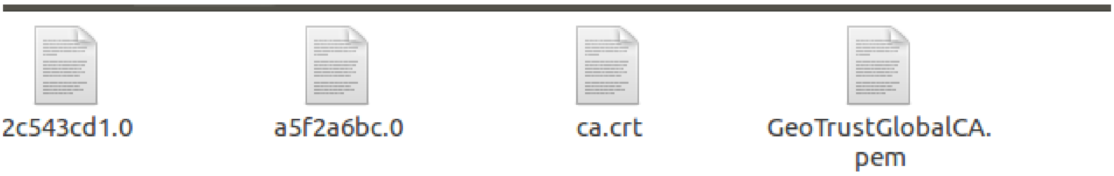
现在我们有了密钥文件，因此我们生成一个证书签名请求(CSR)，它基本上包含公司的公钥。CSR 将被发送到 CA，CA 将为密钥生成证书。

接下来生成 haovpn.com 的证书：

```
[09/12/20]seed@VM:~/.../myca$ openssl ca -in server.csr -out server.crt -cert ca
.crt -keyfile ca.key \-config openssl.cnf
Using configuration from openssl.cnf
Enter pass phrase for ca.key:
Check that the request matches the signature
Signature ok
Certificate Details:
        Serial Number: 4098 (0x1002)
        Validity
            Not Before: Sep 12 18:23:54 2020 GMT
            Not After : Sep 12 18:23:54 2021 GMT
        Subject:
            countryName               = CN
            stateOrProvinceName       = JS
            organizationName          = SEU
            organizationalUnitName    = CS
            commonName                = haovpn.com
            emailAddress              = haochworktime@gmail.com
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            Netscape Comment:
                OpenSSL Generated Certificate
            X509v3 Subject Key Identifier:
                DD:43:EB:82:50:C8:56:11:75:71:1B:58:56:08:19:3B:F0:E3:25:87
            X509v3 Authority Key Identifier:
                keyid:17:14:EA:33:29:22:95:AE:E8:9F:21:45:DD:47:2B:24:94:B7:0A:4
3

Certificate is to be certified until Sep 12 18:23:54 2021 GMT (365 days)
Sign the certificate? [y/n]:y


1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Updated
```



再接下来将 CA 证书复制到 Host U 的客户端文件中，生成符号链接：



尝试连接对端：

```
[09/13/20]seed@VM:~/.../tls$ sudo ./tlsclient haovpn.com 4433
SSL connection is successful
SSL connection using AES256-GCM-SHA384
HTTP/1.1 200 OK
Content-Type: text/html

<!DOCTYPE html><html><head><title>Hello World</title></head><style>body {backgro
und-color: black}h1 {font-size:3cm; text-align: center; color: white;text-shadow
: 0 0 3mm yellow}</style></head><body><h1>Hello, world!</h1></body></html>
```

```
[09/12/20]seed@VM:~/Desktop$ sudo ./tlsserver
Enter PEM pass phrase:
3073554112:error:14094412:SSL routines:ssl3_read_bytes:sslv3 alert bad certifica
te:s3_pkt.c:1487:SSL alert number 42
SSL connection established!
Received: GET / HTTP/1.1
Host: haovpn.com
```

这表明我们自己的 VPN server name 生效了。

如果与证书中的 common name 不同，则会提示证书认证失败，也无法建立连接：

```
[09/13/20]seed@VM:~/.../tls$ sudo ./tlsclient vpnlabserver.com 4433
3073222336:error:14090086:SSL routines:ssl3_get_server_certificate:certificate v
erify failed:s3_clnt.c:1264:
```

Task5: Authenticating the VPN Client

◆ 实验流程：

这里设置一个登陆函数：

```c
void login(char *user, char *passwd) {
    struct spwd *pw;
    char *epasswd;
    pw = getspnam(user);
    if (pw == NULL) { exit(0); }
    printf("Login name: %s\n", pw->sp_namp);
    printf("Passwd : %s\n", pw->sp_pwdp);
    epasswd = crypt(passwd, pw->sp_pwdp);
    if (strcmp(epasswd, pw->sp_pwdp)) { exit(0); }
}

void loginRequest(SSL *ssl, int sock){
    char buf[1024], usr[1024], pwd[1024];

    char *req1 = "Enter username:";
    SSL_write(ssl, req1, strlen(req1));
    int usrlen = SSL_read(ssl, usr, sizeof(usr) - 1);
    usr[usrlen] = '\0';

    char *req2 = "Enter password:";
    SSL_write(ssl, req2, strlen(req2));
    int pwdlen = SSL_read(ssl, pwd, sizeof(pwd) - 1);
    pwd[pwdlen] = '\0';
    login(usr, pwd);

}
```

在客户端的主函数添加如下代码，用于输入用户名和密码：

```c
int len;
char usrbuf[25];
char pwdbuf[25];
char username[25];
char pwd[25];
len = SSL_read (ssl, usrbuf, sizeof(usrbuf) - 1);
usrbuf[len] = '\0';
printf("%s", usrbuf);
scanf("%s", username);
SSL_write(ssl, username, strlen(username));

len = SSL_read (ssl, pwdbuf, sizeof(pwdbuf) - 1);
pwdbuf[len] = '\0';
printf("%s", pwdbuf);
scanf("%s", pwd);
SSL_write(ssl, pwd, strlen(pwd));
```

尝试连接 VPN 服务器：

```
[09/12/20]seed@VM:~/.../tls$ sudo ./tlsclient haovpn.com 4433
SSL connection is successful
SSL connection using AES256-GCM-SHA384
Enter username:seed
Enter password:dees
HTTP/1.1 200 OK
Content-Type: text/html

<!DOCTYPE html><html><head><title>Hello World</title></head><style>body {backgro
und-color: black}h1 {font-size:3cm; text-align: center; color: white;text-shadow
: 0 0 3mm yellow}</style></head><body><h1>Hello, world!</h1></body></html>
```

```
[09/12/20]seed@VM:~/Desktop$ sudo ./tlsserver
Enter PEM pass phrase:
SSL connection established!
user:seed
pwd:dees
seedstop1
Login name: seed
Passwd : $6$wDRrWCQz$IsBXp9.9wz9SGrF.nbihpoN5w.zQx02sht4cTY8qI7YKh00wN/sfYvDeCAc
Eo2QYzCfpZoaEVJ8sbCT7hkxXY/
Received: GET / HTTP/1.1
Host: haovpn.com
```

这样便添加了服务器的授权客户端过程。

Task6: Supporting Multiple Clients

◆ 实验结论：

Task6 的编程细节太复杂，我对管道编程也并不熟悉，所以最后我并没有实现出来。这里的主要实现思路是 VPN Server 的子进程去处理每个不同的隧道，也就是每个子进程会有一个 Socket 用于和不同的 Client 通信。当隧道建立的时候我们是可以对端的虚拟接口 IP 的，此时可以建立一张<Socket(ClientIP), Peer-tun0-IP>的字典，这个字典被父进程所维护。当父进程从网卡 tun0 接收到数据包后，根据目的 IP,找到对应的 socket 所在的子进程，将数据包通过管道传递给该子进程，该子进程再封装好，通过隧道发回 Client。

子进程维护用于和 Client 通信的 Socket，而父进程则只负责监听，子进程中的主要功能逻辑是将收到的数据包传递给父进程(再由父进程通过 tun0 接口转发至私人网络中的目标主机)，以及将父进程传递给子进程的数据包封装，通过隧道转发回 Client；父进程则保留一个监听的套接字(Server——Socket)，然后将建立连接的 Socket 分配给子进程(父进程自己则要关闭这个建立连接的 Socket)，同时维护字典，负责从 tun0 接口读入来自私人网络的数据再根据字典分配给对应的子进程。