VPN 隧道实验

1 概述

虚拟专用网络（VPN）是建立在公共网络（通常是因特网）之上的专用网络。VPN 内的计算机可以安全地进行通信，就像它们在一个真实的、物理上与外界隔离的专用网络上，即使它们的流量会通过公共网络。VPN 使得员工们在旅行时可以安全的接入公司网络；它使公司能够将他们的专用网络扩展到全国各地乃至世界各地。

本实验的目标是帮助学生理解 VPN 的工作原理。我们专注于一种特定类型的，建立在传输层之上的，同时也是最常见的 VPN。我们将从头开始构建一个非常简单的 VPN，并使用该过程来解释 VPN 技术的每个部分是如何工作的。一个真正的 VPN 项目有两个基本部分，隧道和加密。本实验只关注隧道部分，帮助学生理解隧道技术，所以在这个实验中的隧道是没有被加密的。另外还有一个更加全面且包括加密部分的 VPN 实验（VPN Lab）。本实验涵盖以下主题：

- VPN(虚拟专用网络)
- TUN/TAP 虚拟接口
- IP 隧道
- 路由

书籍和视频 TUN/TAP 虚拟接口的详细介绍以及 VPN 的工作原理可参见以下内容：

- SEED Book 的第 19 章, *Computer & Internet Security: A Hands-on Approach*, 3rd Edition, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net`.
- SEED Lecture 的第 8 部分, *Internet Security: A Hands-on Approach*, by Wenliang Du. 详情请见 `https://www.handsonsecurity.net/video.html`.

相关实验 本实验仅涵盖 VPN 隧道部分，而完整的 VPN 还需要保护其隧道。我们有一个另外单独的实验叫做 VPN Lab，这是一个综合性的实验，包括隧道和保护部分。学生可以先完成这个隧道实验。在学习了 PKI 和 TLS 之后，他们可以再做综合性的 VPN 实验。

实验环境 本实验在 SEED Ubuntu 20.04 VM 中测试可行。您可以从 SEED 网站上下载我们预先构建好的镜像并在您自己的电脑上运行 SEED VM。然而，大多数 SEED 实验可以在云端进行，您可以按照我们的说明在云端创建 SEED VM。

2 Task 1: 设置网络

我们将在计算机（客户端）和网关之间建立一条 VPN 隧道，允许计算机安全地通过网关访问专用网络。我们需要至少 3 台机器：VPN 客户端（也用作主机 U ）、VPN 服务器（路由器/网关）和一台在专用网络中的主机（主机 V ）。网络设置如图 1 所示。

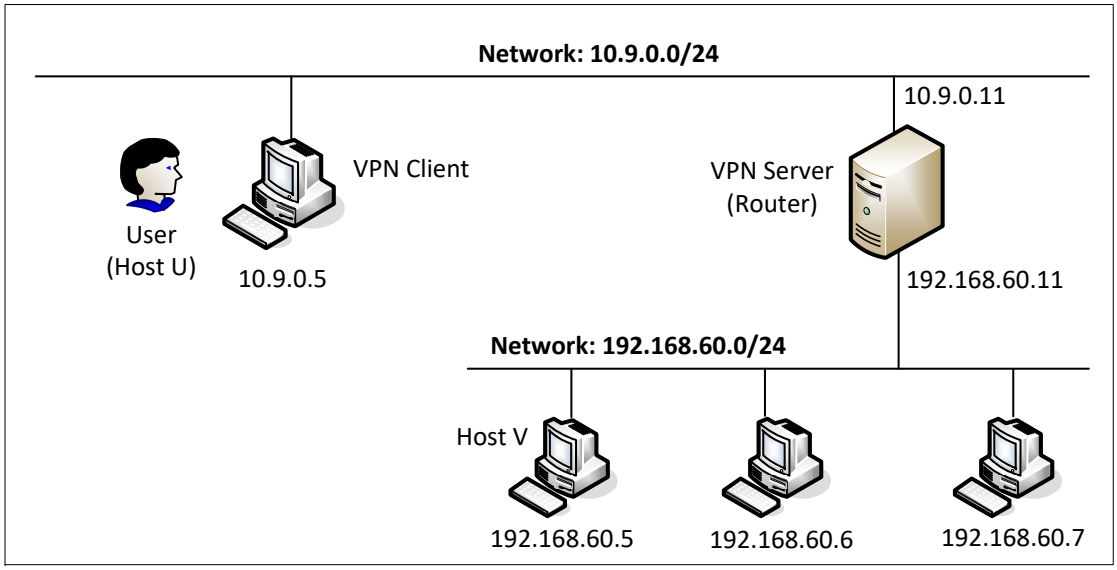

图 1: 实验环境设置

实际上，VPN 客户端和 VPN 服务端是通过互联网连接的。为了简单起见，我们在这个实验直接将这两台机器连接到同一个局域网，用这个局域网模拟互联网。

第三台机器，主机 V ，是专用网络内的一台计算机。主机 U 上的用户（专用网络之外）希望通过 VPN 隧道与主机 V 通信。为了模拟这个设置，我们将主机 V 连接到 VPN 服务器（也用作网关）。在这个设置里，主机 V 不能直接从互联网访问，也不能从主机 U 直接访问。

实验设置。 请从实验的网站下载 `Labsetup.zip` 文件到你的 VM 中，解压它，进入 `Labsetup` 文件夹，然后用 `docker-compose.yml` 文件安装实验环境。对这个文件及其包含的所有 `Dockerfile` 文件中的内容的详细解释都可以在链接到本实验网站的用户手册 [1] 中找到。如果这是您第一次使用容器设置 SEED 实验环境，那么阅读用户手册非常重要。

在下面，我们列出了一些与 Docker 和 Compose 相关的常用命令。由于我们将非常频繁地使用这些命令，因此我们在 `.bashrc` 文件（在我们提供的 SEED Ubuntu 20.04 虚拟机中）中为它们创建了别名。

```
$ docker-compose build  # 建立容器镜像
$ docker-compose up     # 启动容器
$ docker-compose down   # 关闭容器
```

[1] 如果你在部署容器的过程中发现从官方源下载容器镜像非常慢，可以参考手册中的说明使用当地的镜像服务器

```
// 上述 Compose 命令的别名
$ dcbuild       # docker-compose build 的别名
$ dcup          # docker-compose up 的别名
$ dcdown        # docker-compose down 的别名
```

所有容器都在后台运行。要在容器上运行命令，我们通常需要获得容器里的 Shell 。首先需要使用 `docker ps` 命令找出容器的 ID ，然后使用 `docker exec` 在该容器上启动 Shell 。我们已经在 `.bashrc` 文件中为这两个命令创建了别名。

```
$ dockps        // docker ps --format "{{.ID}}  {{.Names}}" 的别名
$ docksh <id>   // docker exec -it <id> /bin/bash 的别名

// 下面的例子展示了如何在主机 C 内部得到 Shell
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// 注: 如果一条 docker 命令需要容器 ID, 你不需要
//     输入整个 ID 字符串。只要它们在所有容器当中
//     是独一无二的, 那只输入前几个字符就足够了。
```

如果你在设置实验环境时遇到问题，可以尝试从手册的 “Miscellaneous Problems” 部分中寻找解决方案。

共享文件夹。 在这个实验里，我们需要写自己的代码，并在容器里运行。在虚拟机中进行代码编辑比在容器中更为方便，因为我们可以使用我们喜欢的编辑器。为了使虚拟机和容器共享文件，我们使用 Docker `volumes` 在虚拟机和容器之间创建了一个共享文件夹。如果你查看 Docker Compose 文件，就会发现我们已经在某些容器中添加了以下条目。它表示将主机（即 VM）上的 `./volumes` 文件夹挂载到容器内的 `/volumes` 文件夹。我们在虚拟机上将代码写入 `./volumes` 文件夹，就可以在容器内使用它们。

```
volumes:
      - ./volumes:/volumes
```

数据包嗅探。 在本实验中，能够嗅探数据包是非常重要的。因为如果事情没有按预期进行，能够查看数据包的去向可以帮助我们找到问题所在。这里有几种不同的方式来进行数据包嗅探：

- 在容器上运行 `tcpdump`。我们已经在每个容器上安装了 `tcpdump`。要嗅探通过特定接口的数据包，我们只需找出该接口的名称，然后执行以下操作（假设接口名为 `eth0`）：

  ```
  # tcpdump -i eth0 -n
  ```

需要注意的是，在容器内部，由于 Docker 创建的隔离，当我们在一个容器内运行 `tcpdump` 时，我们只能嗅探进出该容器的数据包，而不能嗅探其他容器之间的数据包。然而，如果容器在网络设置中使用了 `host` 模式，则可以嗅探其他容器的数据包。

- 在 VM 上运行 `tcpdump`。如果我们在虚拟机上运行 `tcpdump`，则不会受到容器的限制，并且可以嗅探所有容器之间传递的数据包。与容器不同，VM 中网络接口名称有所不同。在容器中，每个接口名通常以 `eth` 开头；而在 VM 中，由 Docker 创建的网络接口名称以 `br-` 开头，后面跟着网络的 ID。您总是可以使用 `ip address` 命令在 VM 和容器上获取接口名称。
- 我们还可以在 VM 上运行 Wireshark 来嗅探数据包。类似于 `tcpdump`，我们需要选择要嗅探的接口。

测试 请进行以下测试以确保实验环境设置正确：

- 主机 `U` 可以与 VPN Server 通信。
- VPN Server 可以与主机 `V` 通信。
- 主机 `U` 应该不能与主机 `V` 通信。
- 在路由器上运行 `tcpdump`，并嗅探每个网络上的流量以测试你是否可以捕获数据包。

3 Task 2: 创建和配置 TUN 接口

我们将要构建的 VPN 隧道基于 TUN/TAP 技术。TUN 和 TAP 是虚拟网络内核驱动程序，他们实现了完全由软件支持的网络设备。TAP 模拟以太网设备，它针对的是网络第 2 层数据包（例如以太网帧），TUN 模拟网络层设备，它针对的是网络第 3 层数据包（例如 IP 数据包）。使用 TUN/TAP，我们可以创建虚拟网络接口。

用户空间程序通常连接到 TUN/TAP 虚拟网络接口。操作系统通过 TUN/TAP 网络接口发送的数据包被传递到用户空间程序。另一方面，程序通过 TUN/TAP 网络接口发送的数据包被注入到操作系统网络堆栈。对于操作系统来说，数据包来自一个通过虚拟网络接口的外部源。

当程序连接到 TUN/TAP 接口时，内核发送到该接口的数据包将通过管道传送到程序中。另一方面，通过程序写入接口的数据包将通过进入内核，就好像它们是通过这个虚拟网络接口从外部来的一样。该程序可以使用标准的 `read()` 和 `write()` 系统调用从虚拟接口接收数据包或将数据包发送到虚拟接口。

本任务的目的是熟悉 TUN/TAP 技术。我们将通过几个任务来了解 TUN/TAP 接口的技术细节。我们将使用以下 Python 程序作为实验的基础，并在整个实验过程中修改此代码。该代码已包含在 zip 文件的 `volumes` 文件夹中。

Listing 1: 创建 TUN 接口 (`tun.py`)

```
#!/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

# 创建一个tun接口
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)

# 获取接口的名称
ifname = ifname_bytes.decode('UTF-8')[:16].strip("\x00")
print("Interface Name: {}".format(ifname))

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

3.1 Task 2.a: 接口名称

我们将在主机 U 上运行 tun.py 程序。我们需要把上述 tun.py 程序设置成可执行，并使用 root 权限来运行它。请参阅以下命令：

```
// 使程序可执行
# chmod a+x tun.py

// 使用root权限运行它
# tun.py
```

一旦程序执行，它将进入阻塞状态。你可以切换到另一个终端，并在容器中打开一个新的命令行界面。接着，打印出机器上所有的网络接口。请在执行以下命令后，报告你的观察结果：

```
# ip address
```

你应该能够找到一个名为 tun0 的接口，你的任务是修改 tun.py 程序，并使用你的姓氏（拼音）作为其名称的前缀。例如，如果你姓张，则应使用 zhang 作为前缀。如果你的姓太长，可以使用前五个字符。请展示你的结果。

3.2 Task 2.b: 设置 TUN 接口

此时，TUN 接口是不可用的，因为它还没有被配置。在使用接口之前，我们需要做两件事。首先，我们需要为其分配一个 IP 地址。其次，我们需要打开该接口，因为接口还处于关闭（down）状态。我们可以使用以下两个命令进行配置：

```
// 为接口分配IP地址
# ip addr add 192.168.53.99/24 dev tun0

// 打开该接口
# ip link set dev tun0 up
```

为了方便起见，同学们可以在 tun.py 中添加如下两行代码，这样就可以由程序自动进行配置了。

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

运行上面两个命令后，再次运行"ip address" 命令，报告你的观察结果。这与运行配置命令之前有何不同?

3.3 Task 2.c: 从 TUN 接口读取数据包

在此任务中，我们将从 TUN 接口读取数据。从 TUN 接口读出来的任何内容都是 IP 数据包。我们可以将从接口接收到的数据转换成一个 Scapy IP 对象，这样我们就可以打印出 IP 数据包的每个字段。请使用以下 while 循环替换 tun.py 中的循环：

```
while True:
  # 从tun接口获取一个数据包
  packet = os.read(tun, 2048)
  if packet:
    ip = IP(packet)
    print(ip.summary())
```

请在主机 U 上运行修改后的 tun.py 程序，并相应配置 TUN 接口，然后进行以下实验。请描述一下你观察到的现象：

- 在主机 U 上，ping 一下 192.168.53.0/24 网络中的主机。tun.py 程序打印出了什么？发生了什么事？为什么？
- 在主机 U 上，ping 内部网络 192.168.60.0/24 中的主机，tun.py 是否打印出任何内容？为什么？

3.4 Task 2.d: 将数据包写入 TUN 接口

在这个任务中，我们将把数据写入 TUN 接口。由于这是一个虚拟网络接口，因此应用程序写入该接口的任何内容都将以 IP 数据包的形式出现在内核中。

我们将修改 `tun.py` 程序，以便从 TUN 接口得到一个数据包后我们可以根据收到的数据包构造一个新的数据包。然后我们将新数据包写入 TUN 接口。如何构建新数据包取决于学生。以下代码显示了如何将一个 IP 数据包写入 TUN 接口的示例。

```
# 使用tun接口发送一个伪造的数据包。
newip = IP(src='1.2.3.4', dst=ip.src)
newpkt = newip/ip.payload
os.write(tun, bytes(newpkt))
```

请根据以下要求修改 `tun.py` 代码：

- 从 TUN 接口得到一个数据包后，如果这个包是 ICMP echo request 数据包，则构造一个对应的 echo reply 数据包写入到 TUN 接口。请证明你的代码按预期工作了。
- 不要将 IP 数据包写入接口，而是将一些任意数据写入接口，并报告你的观察结果。

4 Task 3: 通过隧道将 IP 数据包发送到 VPN 服务器

在这个任务中，我们将从 TUN 接口接收到的 IP 数据包放入一个新的 IP 数据包的 UDP 载荷字段中，并将其发送到另一台计算机。也就是说，我们将原始数据包放在一个新的数据包里面。这被称为 IP 隧道技术。隧道的实现仅仅是标准的客户端/服务器编程。它可以建立在 TCP 或 UDP 之上。在这个任务中，我们将使用 UDP。也就是说，我们将一个 IP 数据包放入 UDP 数据包的载荷字段中。

服务器程序 `tun_server.py` 我们将在 VPN Server 上运行 `tun_server.py` 程序。这个程序只是一个标准的 UDP 服务器程序。它监听端口 `9090` 并打印出接收到的任何内容。程序假设 UDP 有效负载字段中的数据是一个 IP 数据包，因此它将有效负载转换为 Scapy `IP` 对象，并打印出这个 IP 数据包的源 IP 地址和目标 IP 地址。

Listing 2: `tun_server.py`

```
#!/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))
```

实现客户端程序 tun_client.py 首先我们需要修改 TUN 程序 tun.py。把它重命名为 tun_client.py。使用 UDP 向另一台计算机发送数据可以使用标准套接字（socket）编程。

将程序中的 while 循环中的 SERVER_IP 和 SERVER_PORT 替换为 VPN Server 上运行的服务器程序的实际 IP 地址和端口号。

```
# Create UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

while True:
   # 从tun接口获取一个数据包
   packet = os.read(tun, 2048)
   if packet:
      # 通过隧道传输一个数据包
      sock.sendto(packet, (SERVER_IP, SERVER_PORT))
```

测试 在 VPN Server 上运行 tun_server.py 程序，然后在主机 U 上运行 tun_client.py。为了测试隧道是否工作，请 ping 任何一个属于 192.168.53.0/24 网络的 IP 地址。VPN Server 上打印的内容是什么？为什么？

我们的最终目标是使用隧道访问私有网络 192.168.60.0/24 内的主机。让我们 ping 主机 V，看看 ICMP 数据包是否通过隧道发送到了 VPN Server。如果没有，问题出在哪？你需要解决这个问题，这样才能通过隧道发送 ping 数据包。这是通过路由完成的，去往 192.168.60.0/24 网络的数据包应该被路由到 TUN 接口并被交给 tun_client.py 程序。以下命令显示如何在路由表中添加新的条目。

```
# ip route add <network> dev <interface> via <router ip>
```

请证明当你 ping 一个在 192.168.60.0/24 网络中的 IP 地址时，ICMP 数据包是通过隧道到达的。

5 Task 4: 设置 VPN 服务器

在 tun_server.py 从隧道中获取数据包后，它需要将数据包提供给内核，以便内核可以将数据包路由到其最终目的地。这需要通过一个 TUN 接口来完成，就像我们在 Task 2 中所做的一样。请修改 tun_server.py，使其可以执行以下操作：

- 创建一个 TUN 接口并对其进行配置。
- 从 socket 接口获取数据，将接收到的数据视为 IP 数据包。
- 将数据包写入 TUN 接口。

在运行修改后的 tun_server.py 之前，我们需要打开 IP 转发（IP forwarding）。除非有特别配置，否则计算机将仅充当主机，而不充当网关。VPN 服务器需要在私网和隧道之间转发数据包，所以需要起到网关的作用。我们需要为计算机启用 IP 转发，使其行使网关的功能。我们在路由器容器上已经开启了 IP 转发。你可以在 docker-compose.yml 中看到路由器容器的以下条目：

```
sysctls:
        - net.ipv4.ip_forward=1
```

测试 如果一切设置正确，我们可以从主机 U ping 主机 V 。ICMP 回显请求数据包最终应该通过隧道到达主机 V 。请展示你的验证结果。需要注意的是，虽然主机 V 会响应 ICMP 数据包，但 reply 不会回到主机 U ，因为我们还没有设置好一切。因此，对于此任务，显示（使用 Wireshark 或 tcpdump）ICMP 数据包已到达主机 V 就足够了。

6 Task 5: 处理双向流量

到这里，我们的隧道的一个方向就完成了，即我们可以通过隧道将数据包从主机 U 发送到主机 V 。如果我们查看主机 V 上的 Wireshark，可以看到主机 V 已发出响应，但数据包在某处被丢弃。这是因为我们的隧道只设置好了一个方向，我们还需要设置它的另一个方向，返回的流量才能通过隧道返回到主机 U 。

为了设置好另外一个方向，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。所有这些接口都由文件描述符（file descriptor）表示，所以我们需要监视它们以查看是否有来自它们的数据。一种方法是不断去查询它们，看看每个接口上是否有数据。这种方法的性能是不可取的，因为当没有数据时，进程必须仍然做着空循环。另一种方法是轮流从接口上读取数据。在默认情况下，读是阻塞的，即如果没有数据，进程将等待在那。当有数据可用时，进程才能继续执行。这样就不会在没有数据的时候浪费 CPU 时间。

基于读取的阻塞机制适用于一个接口。如果一个进程需要从多个接口上读取数据，它不能只在其中一个接口上阻塞，否则别的接口有数据时，进程没法去处理。进程必须能监控所有接口。Linux 有一个称为 select() 的系统调用，它允许程序同时监视多个文件描述符。为了使用 select()，进程需要将所有要监视的文件描述符存储在一个数组中，然后将它给 select() 系统调用。系统会阻塞该进程，直到其中的一个文件描述符上有数据可用。我们可以检查哪个文件描述符接收到了数据。在下面的 Python 代码片段中，我们使用 select() 来监控 TUN 和 socket 文件描述符。

```
# 我们假设 sock 和 tun 文件描述符已经被创建。

while True:
  # 程序将被阻塞，直到其中至少有一个接口上有数据
  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))
       ... (code needs to be added by students) ...

    if fd is tun:
```

```
        packet = os.read(tun, 2048)
        pkt = IP(packet)
        print("From tun    ==>: {} --> {}".format(pkt.src, pkt.dst))
        ... (code needs to be added by students) ...
```

同学们可以用上面的代码替换 TUN 客户端和服务端程序中的 `while` 循环。代码是不完整的，缺的部分留给同学们自己完成。

测试。 完成上面的操作后，我们应该能够在主机 U 与 V 之间通信，VPN 隧道（未加密）就算完成了。请使用 textttping 和 `telnet` 命令来验证，并用 wireshark 提供证明。在你的报告中，需要指出你的数据包是如何流动的。

7 Task 6: 隧道中断实验

在主机 U 上远程登录到主机 V 。在保持 `telnet` 连接的同时，我们通过中断 `tun_client.py` 或 `tun_server.py` 程序来中断 VPN 隧道。然后我们在 `telnet` 窗口中输入一些内容。你看到输入的内容了吗？TCP 连接会发生什么？连接是否断开？

现在让我们重新连接 VPN 隧道(不要等待太久)。我们再次运行 `tun_client.py` 和 `tun_server.py` 程序，并设置它们的 TUN 接口和路由。一旦隧道重新建立，`telnet` 连接会发生什么？请描述并解释你的观察结果。

8 Task 7: 主机 V 上的路由实验

在真实的 VPN 系统中，流量会被加密（本实验不涉及这部分）。这意味着返回流量必须从同一隧道返回。如何获取从主机 V 到 VPN 服务器的返回流量并非易事。我们的设置简化了这种情况。在我们的设置中，主机 V 的路由表有一个默认设置：去往任何目的地的数据包，除了 `192.168.60.0/24` 网络，将被自动路由到 VPN 服务器。

在现实世界中，主机 V 可能距离 VPN 服务器有多跳，默认路由条目可能无法保证将返回的数据包路由回 VPN 服务器。我们必须正确设置专用网络内的路由表，以确保将去往隧道另一端的数据包路由到 VPN 服务器。为了模拟这种情况，我们从主机 V 中删除默认条目，然后在路由表中添加一个更具体的条目，以便可以将返回的数据包路由回 VPN 服务器。学生可以使用以下命令删除默认条目并添加新条目：

```
// 删除默认条目
# ip route del default

// 添加一个条目
# ip route add <network prefix> via <router ip>
```

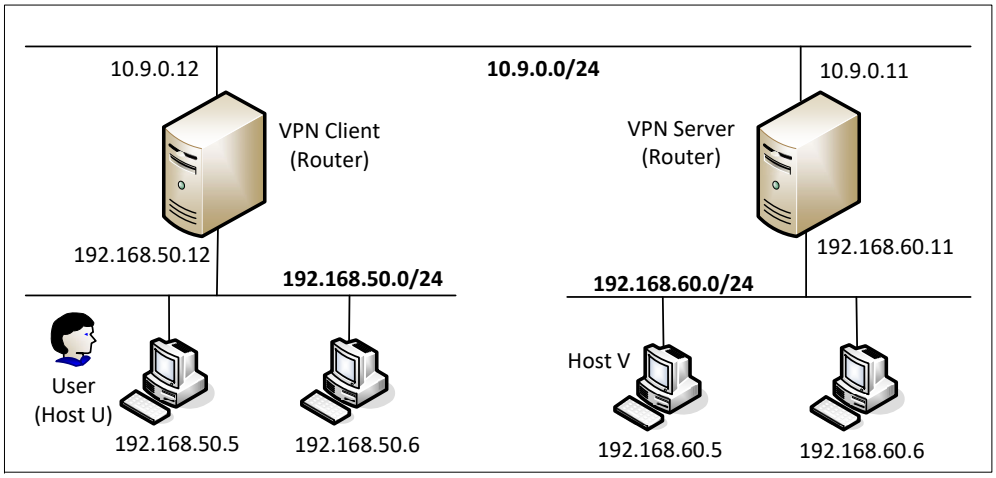

图 2: 两个专用网络之间的 VPN

9 Task 8: 专用网络之间的 VPN

在此任务中,我们将在两个专用网络之间建立 VPN。设置如图 2所示。整个设置放在了 `docker-compose2.yml` 文件中，我们可以使用`"-f docker-compose2.yml"` 选项让 `docker-compose` 使用该文件，否则默认的 `docker-compose.yml` 文件会被使用。

```
$ docker-compose -f docker-compose2.yml build
$ docker-compose -f docker-compose2.yml up
$ docker-compose -f docker-compose2.yml down
```

此设置模拟了一个机构有两个站点的情况，每个站点都有一个专用网络。连接这两个网络的唯一方法是通过互联网。你的任务是在这两个站点之间建立 VPN，因此这两个网络之间的通信将通过 VPN 隧道。你可以使用之前开发的代码，但你需要考虑如何设置正确的路由，以便这两个专用网络之间的数据包可以路由到 VPN 隧道。在你的报告中，请描述并解释你的工作。你需要提供证据证明这两个专用网络之间的数据包确实通过了 VPN 隧道。

10 Task 9: 用 TAP 接口实验

在这个任务中，我们将用 TAP 接口做一个简单的实验，让学生对这种类型的接口有所了解。TAP 接口的工作方式与 TUN 接口非常相似。主要区别在于 TUN 接口的内核端挂接到 IP 层，而 TAP 接口的内核端挂接到 MAC 层。因此，通过 TAP 接口的数据包包含 MAC 头，而通过 TUN 接口的数据包只包含 IP 头。使用 TAP 接口，应用程序除了获取包含 IP 数据包的帧外，还可以获取其他类型的帧，例如 ARP 帧。

我们将使用以下程序进行实验，我们将只使用 VPN 客户端容器（两个实验的环境设置任何一个都可以）。创建 TUN 接口和 TAP 接口的代码是比较相似的，唯一的区别在于接口类型。对于 TAP 接口，我们使用 `IFF_TAP`，而对于 TUN，我们使用 `IFF_TUN`。其余代码相同，以下不再赘述。TAP 接口的

配置方法与 TUN 接口的配置方法完全相同。

```
...

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")
...

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

上面的代码只是从 TAP 接口读取。然后它将数据转换为 Scapy `Ether` 对象，并打印出其所有字段。请 `ping` 一下 `192.168.53.0/24` 网络中的一个 IP 地址，在报告中描述并解释你观察到的现象。

我们可以做一件有趣的事，一旦你从 TAP 接口得到一个以太网帧，你可以检查它是否是一个 ARP 请求，如果是，则生成相应的 ARP 回复并将其写入 TAP 接口。下面提供了一个示例代码：

```
while True:
   packet = os.read(tun, 2048)
   if packet:
      print("--------------------------------")
      ether = Ether(packet)
      print(ether.summary())

      # 发送一个伪造的ARP请求
      FAKE_MAC   = "aa:bb:cc:dd:ee:ff"
      if ARP in ether and ether[ARP].op == 1 :
         arp       = ether[ARP]
         newether  = Ether(dst=ether.src, src=FAKE_MAC)
         newarp    = ARP(psrc=arp.pdst, hwsrc=FAKE_MAC,
                         pdst=arp.psrc, hwdst=ether.src, op=2)
         newpkt     = newether/newarp

         print("***** Fake response: {}".format(newpkt.summary()))
         os.write(tun, bytes(newpkt))
```

为了测试你的 TAP 程序，你可以对任何 IP 地址运行 `arping` 命令。该命令通过指定的接口向指定的 IP 地址发出 ARP 请求。如果你伪造 arp 回复的程序工作正常，你应该能够得到响应。请参阅以下示例。

```
arping -I tap0 192.168.53.33
arping -I tap0 1.2.3.4
```

11 提交

你需要提交一份带有截图的详细实验报告来描述你所做的工作和你观察到的现象。你还需要对一些有趣或令人惊讶的观察结果进行解释。请同时列出重要的代码段并附上解释。只是简单地附上代码不加以解释不会获得学分。