VPN Tunneling Lab

Contents

Lab Setup

Please download the Labsetup.zip file from the below link to your VM, unzip it, enter the Labsetup folder, and use the docker-compose.yml file to set up the lab environment.

https://seedsecuritylabs.org/Labs_20.04/Files/VPN_Tunnel/Labsetup.zip

We will create a VPN tunnel between a computer (client) and a gateway, allowing the computer to securely access a private network via the gateway.
We need at least three machines: VPN client (also serving as Host U), VPN server (the router/gateway), and a host in the private network (Host V).
In practice, the VPN client and VPN server are connected via the Internet. For the sake of simplicity, we directly connect these two machines to the same LAN in this lab, i.e., this LAN simulates the Internet. The third machine, Host V, is a computer inside the private network. Users on Host U (outside of the private network) want to communicate with Host V via the VPN tunnel. To simulate this setup, we connect Host V to VPN Server (also serving as a gateway). In such a setup, Host V is not directly accessible from the Internet; nor is it directly accessible from Host U

Lab Overview

A Virtual Private Network (VPN) is a private network built on top of a public network, usually the Internet. Computers inside a VPN can communicate securely, just like if they were on a real private network that is physically isolated from outside, even though their traffic may go through a public network. VPN enables employees to securely access a company’s intranet while traveling; it also allows companies to expand their private networks to places across the country and around the world.

The objective of this lab is to help students understand how VPN works. We focus on a specific type of VPN (the most common type), which is built on top of the transport layer. We will build a very simple VPN from the scratch, and use the process to illustrate how each piece of the VPN technology works. A real VPN program has two essential pieces, tunneling and encryption. This lab only focuses on the tunneling part, helping students understand the tunneling technology, so the tunnel in this lab is not encrypted. There is another more comprehensive VPN lab that includes the encryption part.

This lab covers the following topics:

- Virtual Private Network
- The TUN/TAP virtual interface
- IP tunneling
- Routing

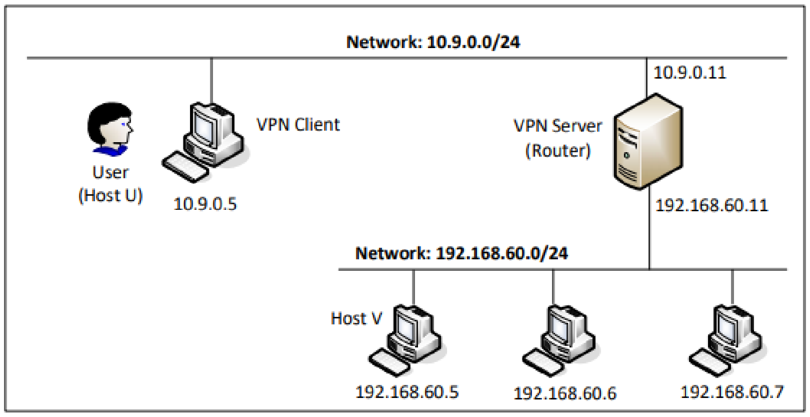

Figure 1: The Network Setup

Task 1: Network Setup

We are only using docker-compose.yml, please delete docker-compose2.yml and open all the docker container terminals using the given Labsetup folder.

Shared Folder - In this lab, we need to write our own code and run it inside containers. Code editing is more convenient inside the VM than in containers, because we can use our favorite editors. In order for the VM and container to share files, we have created a shared folder between the VM and the container using the Docker volumes. If you look at the Docker Compose file, you will find out that we have added the following entry to some of the containers. It indicates mounting the ./volumes folder on the host machine (i.e., the VM) to the /volumes folder inside the container. We will write our code in the ./volumes folder (on the VM), so they can be used inside the containers

Packet sniffing - Being able to sniff packets is very important in this lab, because if things do not go as expected, being able to look at where packets go can help us identify the problems.

- Running tcpdump on containers. We have already installed tcpdump on each container. To sniff the packets going through a particular interface, we just need to find out the interface name, and then do the following (assume that the interface name is eth0):
  Command:
  ```
  # tcpdump -i eth0 -n
  ```

Testing

Please conduct the following testings to ensure that the lab environment is set up correctly:
• Host U - 10.9.0.5 can communicate with VPN Server (server-router)

On Client-10.9.0.5
Command:
```
# ping server-router
```

. • VPN Server (server-router) can communicate with Host V (host-192.168.60.5)

On server-router

Command :

```
# ping 192.168.60.5
```

• Host U (Client - 10.9.0.5) should not be able to communicate with Host V (host 192.168.60.5)

On Client 10.9.0.5

Command:

```
# ping 192.168.60.5
```

• Run tcpdump on the router, and sniff the traffic on each of the networks. Show that you can capture packets.

On server-router run -

Command:

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

On Client - 10.9.0.5

Command:

```
# ping server-router
```

Take screenshots of each step.

Task 2: Create and Configure TUN Interface

The VPN tunnel that we are going to build is based on the TUN/TAP technologies. TUN and TAP are virtual network kernel drivers; they implement network devices that are supported entirely in software. TUN (as in network TUNnel) simulates a network layer device and it operates with layer-3 packets such as IP packets. With TUN/TAP, we can create virtual network interfaces.

A user-space program is usually attached to the TUN virtual network interface. Packets sent by an operating system via a TUN network interface are delivered to the user-space program. On the other hand, packets sent by the program via a TUN network interface are injected into the operating system network stack. To the operating system, it appears that the packets come from an external source through the virtual network interface

The objective of this task is to get familiar with the TUN technology. We will conduct several experiments to learn the technical details of the TUN interface. We will use the following Python program as the basis for the experiments, and we will modify this base code throughout this lab. The code is already included in the volumes folder in the zip file.

Task 2.a: Name of the Interface

We will run the tun.py program on Host U (Client-10.9.0.5). Make the above tun.py program executable, and run it using the root privilege.

On Client - 10.9.0.5 **(change directory to ./volumes)**

Command:

```
# chmod a+x tun.py
# ./tun.py &
# ip addr
```

Take screenshots and explain your understanding.

You should be able to find an interface called **tun0**.

Kill the earlier Tunnel Process -

On Client - 10.9.0.5

Command:

```
# kill %1
```

(In case you get an error, run

```
# jobs
```

Note down the number for ./tun.py, then run

```
# kill %[the number] )
```

Your job in this task is to **change the tun.py program**, so instead of using tun as the prefix of the interface name, use the last 5 digits of your SRN as the prefix. **Please show your results with appropriate screenshots.**

In **tun.py** replace "tun" with the last 5 characters of your SRN in line 16.

Line 16 - ifr = struct.pack('16sH', b'tun%d', IFF_TUN | IFF_NO_PI)

Now run the following commands again and see the new tunnel interface, it should be "**SRN0**"

On Client - 10.9.0.5

Command:

```
# chmod a+x tun.py
# ./tun.py &
# ip addr
```

Take screenshots.

Task 2.b: Set up the TUN Interface

At this point, the TUN interface is not usable, because it has not been configured yet. There are two things that we need to do before the interface can be used. First, we need to assign an IP address to it. Second, we need to bring up the interface, because the interface is still in the down state. We can use the following two commands for the configuration:

On Client - 10.9.0.5

Command:

```
# ip addr add 192.168.53.99/24 dev <SRN>0
# ip link set dev <SRN>0 up
```

Replace <SRN> with what you have used to change link 16 in tun.py

Task 2.c: Read from the TUN Interface

In this task, we will read from the TUN interface. Whatever coming out from the TUN interface is an IP packet. We can cast the data received from the interface into a Scapy IP object, so we can print out each field of the IP packet.

Please use the following while loop to replace the **one in tun.py**:

Replace the following in tun.py -

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

With -

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

Kill the earlier Tunnel Process -

On Client - 10.9.0.5

Command:

```
# kill %1
```

Set up the TUN Interface -

On Client - 10.9.0.5

Command:

```
#  ip addr add 192.168.53.99/24 dev <SRN>0
#  ip link set dev <SRN>0 up
```

Replace <SRN> with what you have used to change link 16 in tun.py

Run the revised tun.py program -

On Client - 10.9.0.5

Command:

```
# ./tun.py &
```

On Host U, ping a host in the 192.168.53.0/24 network. What is printed out by the tun.py program? What has happened? Why?

On Client - 10.9.0.5

Command:

```
# ping 192.168.53.5
```

On Host U, ping a host in the internal network 192.168.60.0/24, Does tun.py print out anything? Why?

On Client - 10.9.0.5

Command:

```
# ping 192.168.60.5
```

Provide appropriate screenshots along with your explanations.

Kill the earlier Tunnel Process -

On Client - 10.9.0.5

Command:

```
# kill %1
```

Task 2.d: Write to the TUN Interface

In this task, we will write to the TUN interface. Since this is a virtual network interface, whatever is written to the interface by the application will appear in the kernel as an IP packet.

We will modify the tun.py program, so after getting a packet from the TUN interface, we construct a new packet based on the received packet. We then write the new packet to the TUN interface.

Your job in this task is to **change the tun1.py program**, so instead of using tun as the prefix of the interface name, use the last 5 digits of your SRN as the prefix. **Please show your results with appropriate screenshots.**

In **tun1.py** replace "tun" with the last 5 characters of your SRN in line 16.
Line 16 - ifr = struct.pack('16sH', b'tun%d', IFF_TUN | IFF_NO_PI)

On Client - 10.9.0.5

Command:

```
# chmod a+x tun.py
# ./tun1.py &
# ip addr
```

After getting a packet from the TUN interface, if this packet is an ICMP echo request packet, construct a corresponding echo reply packet and write it to the TUN interface. Please provide evidence to show that the code works as expected

On Client - 10.9.0.5

Command:

```
# ping 192.168.53.5
```

Take appropriate screenshots of each step with the required observations.

Kill the earlier Tunnel Process -

On Client - 10.9.0.5

Command:

```
# kill %1
```

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

In this task, we will put the IP packet received from the TUN interface into the UDP payload field of a new IP packet, and send it to another computer. Namely, we place the original packet inside a new packet. This is called IP tunneling. The tunnel implementation is just standard client/server programming. It can be built on top of TCP or UDP. In this task, we will use UDP. Namely, we put an IP packet inside the payload field of a UDP packet.

The server program tun_server.py - We will run the tun_server.py program on VPN Server. This program is just a standard UDP server program. It listens to port 9090 and prints out whatever is received. The program assumes that the data in the UDP payload field is an IP packet, so it casts the payload to a Scapy IP object, and prints out the source and destination IP address of the enclosed IP packet.

On server-router run **(change directory to ./volumes)**

Command:

```
# chmod a+x tun_server.py
# ./tun_server.py
```

Implement the client program tun_client.py - First, we need to modify the TUN program tun.py. Let's rename it, and call it tun_client.py. Sending data to another computer using UDP can be done using the standard socket programming. Replace the while loop in the program with the following: The SERVER IP and SERVER PORT should be replaced with the actual IP address and port number of the server program running on VPN Server

Note - In **tun_client.py** replace "tun" with the last 5 characters of your SRN in line 16.

Run tun_client.py on Client - 10.9.0.5

Command:

```
# chmod a+x tun_client.py
# ./tun_client.py &
# ip addr
```

To test whether the tunnel works or not, ping any IP address belonging to the 192.168.53.0/24 network. What is printed out on VPN Server? Why?

On Client - 10.9.0.5

Command:

```
# ping 192.168.53.5
```

Let us ping Host V, and see whether the ICMP packet is sent to VPN Server through the tunnel.

On Client - 10.9.0.5

Command:

```
# ping 192.168.60.5
```

This is done through routing, i.e., packets going to the 192.168.60.0/24 network should be routed to the TUN interface and be given to the tun_client.py program. The routing has already been added to the tun_client.py program.

To check the routing run the following command on Client - 10.9.0.5

Command:

```
# ip route
```

Please provide screenshots with detailed explanations.

Task 4: Set Up the VPN Server

After tun_server.py gets a packet from the tunnel, it needs to feed the packet to the kernel, so the kernel can route the packet towards its final destination. This needs to be done through a TUN interface, just like what we did in Task 2. We have modified tun_server.py, so it can do the following:

- Create a TUN interface and configure it.
- Get the data from the socket interface; treat the received data as an IP packet.
- Write the packet to the TUN interface.

Note - In **tun_server1.py** replace “tun” with the last 5 characters of your SRN in line 18.

On server-router run

Command:

```
# chmod a+x tun_server1.py
# ./tun_server1.py
```

On host-192.168.60.5 run -

Command:

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

On Client - 10.9.0.5 run -

Command:

```
# ping 192.168.60.5
```

Note - If you haven't set up the tun_client (client side tunnel) then please do the same, by following the previous task. (Running tun_client.py)

If everything is set up properly, we can ping Host V (192.168.60.5) from Host U (10.9.0.5). The ICMP echo request packets should eventually arrive at Host V through the tunnel. Please show your proof. It should be noted that although Host V will respond to the ICMP packets, the reply will not get back to Host U, because we have not set up everything yet. Therefore, for this task, it is sufficient to show (tcpdump) that the ICMP packets have arrived at Host V.

Take screenshots of all the terminals and explain your observation.

Kill the earlier Tunnel Process -

On Client - 10.9.0.5

Command:

```
# kill %1
```

Task 5: Handling Traffic in Both Directions

After getting to this point, one direction of your tunnel is complete, i.e., we can send packets from Host U to Host V via the tunnel. If we look at the Wireshark trace on Host V, we can see that Host V has sent out the response, but the packet gets dropped somewhere. This is because our tunnel is only one directional; we need to set up its other direction, so returning traffic can be tunneled back to Host U.

To achieve that, our TUN client and server programs need to read data from two interfaces, the TUN interface and the socket interface. All these interfaces are represented by file descriptors, so we need to monitor them to see whether there is data coming from them. One way to do that is to keep polling them, and see whether there is data on each of the interfaces. The performance of this approach is undesirable, because the process has to keep running in an idle loop when there is no data. Another way is to read from an interface. By default, read is blocking, i.e The process will be suspended if there is no data. When data becomes available, the process will be unblocked, and its execution will continue. This way, it does not waste CPU time when there is no data.

We use two new programs **tun_client_select.py** and **tun_server_select.py**. In **both the mentioned programs** replace "tun" with the last 5 characters of your SRN in line 15 (client) and line 19 (server).
The line is - " ifr = struct.pack('16sH', b'tun%d', IFF_TUN | IFF_NO_PI)"

Open **two terminals of the Client - 10.9.0.5** Machine, this improves comprehensibility for what we are about to execute.

You will **need wireshark** for this task, capturing the packets on the client interface.

On the server-router run -
Command:

```
# chmod a+x tun_server_select.py
# ./tun_server_select.py
```

On one terminal of Client - 10.9.0.5 run -
Command:

```
# chmod a+x tun_client_select.py
# ./tun_client_select.py
```

On the other terminal of Client - 10.9.0.5 run -
Command:

```
# ping 192.168.60.5
```

Capture the packets on Wireshark and take required screenshots of all terminals.

Now we Telnet into 192.168.60.5 - (same terminal as the one who've pinged from)

Command:

```
# telnet 192.168.60.5
```

Capture the packets on Wireshark and take required screenshots of all terminals.

Please show your wireshark proof using ping and telnet commands. In your proof, you need to point out how your packets flow.

Do not close the VPN connections (server and client), as we require it for the next task as well.

Task 6: Tunnel-Breaking Experiment

On Host U (10.9.0.5), telnet to Host V (192.168.60.5) .

(Perform Task 5 again, incase you have closed both the Client and Server Tunnel Connections)

On Client -10.9.0.5

Command:

```
# telnet 192.168.60.5
```

- Log on and while keeping the telnet connection alive, we break the VPN tunnel by stopping the tun_server_select.py program - **Ctrl + C in server-router**

We then type something in the telnet window.

Do you see what you type? What happens to the TCP connection? Is the connection broken? Explain.

Let us now reconnect the VPN tunnel (do not wait for too long).

On the server-router run -

Command:

```
# ./tun_server_select.py
```

Once the tunnel is re-established, what is going to happen to the telnet connection? Please describe and explain your observations.

Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.

.