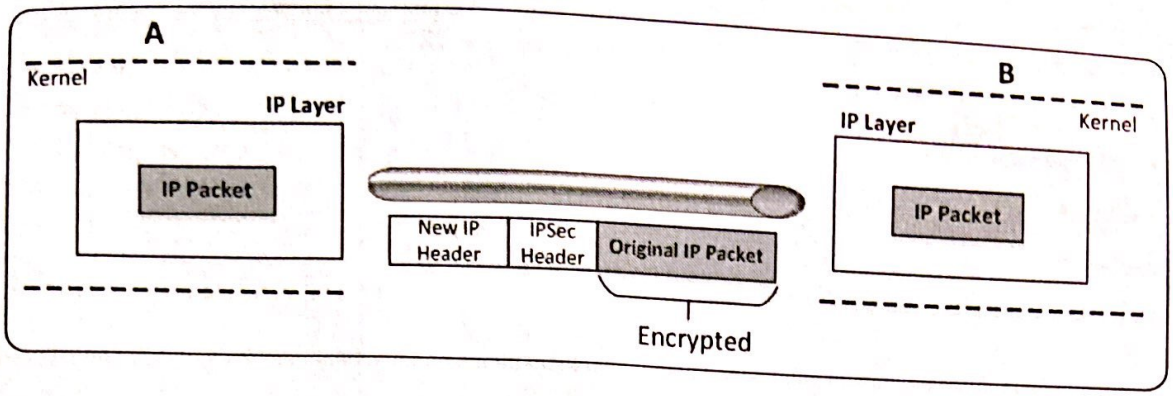
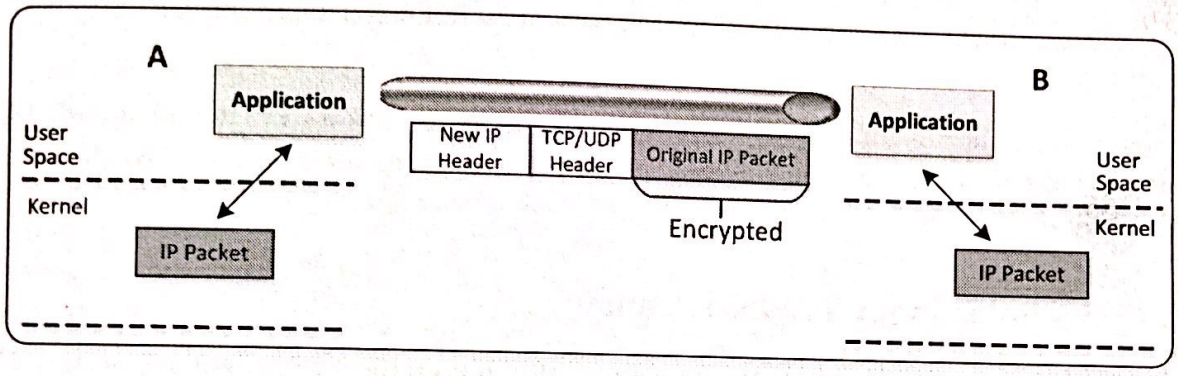


(a) IPSec Tunneling



(b) TLS/SSL Tunneling

Figure 5.2: IPSec and TLS/SSL Tunneling

5.2 An Overview of How TLS/SSL VPN Works

We use Figure 5.3 to give a high-level explanation of how the TLS/SSL VPN works. In this figure, we provide a generalized network scenario, which involves two private networks belonging to the same organization, but in two different geographic locations. In the past, for situation like this, we had to build or lease a dedicated line to connect these two locations. Not many organizations can afford such a dedicated line, so the communication between these two sites has to go through a public infrastructure such as the Internet. Using IP tunneling, we can build a dedicated virtual line between these two locations. As long as adversaries cannot see what is communicated on the line, and cannot inject their data into the line, the line has achieved what a dedicated physical line can do. This line is the tunnel that we want to build. With it, the two geographically separated private networks can form one single virtual private network. Hosts inside this virtual private network can communicate with one another just like those in the same physical private network, even though they are not.

Creating this virtual line, i.e., the tunnel, is the job of two computers, one called VPN client and the other called VPN server. The private network that owes the VPN server is the primary site, while the other one is the satellite site, which needs to join the primary site to form a virtual private network. The job of the VPN client and server can be divided into three main tasks: (1) establish a secure tunnel between them, (2) forward IP packets that need to go to the other

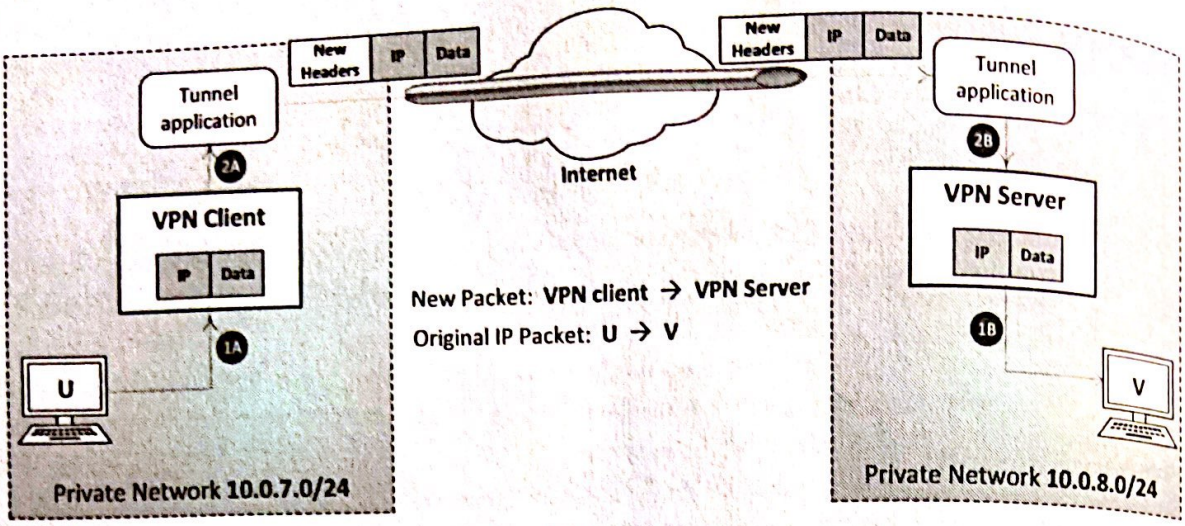


Figure 5.3: Overview of the operation of a tunneling application.

private network through this tunnel, (3) after receiving an IP packet from the other end of the tunnel, release it into the private network (the physical one), so the packet can reach its final destination.

5.2.1 Establishing A TLS/SSL Tunnel

A TLS/SSL tunnel is established by applications running on the VPN client and server. The tunnel is basically a TLS/SSL channel between the client and server applications, and the channel can be built on top of TCP or UDP. Before this channel is established, mutual authentications are needed: the server needs to authenticate the client, making sure that the client is allowed to join the private network. This is usually through password authentication or other types of credentials provided by the client. The client also needs to authenticate the server to make sure it joins the intended private network; the client does not want to send its password or private network traffic to a wrong server. This type of authentication is usually done through public key certificate.

Once the TLS/SSL channel is established, both client and server can send data through this channel to the other side. Due to TLS/SSL, data going through the channel are encrypted and Message Authentication Code (MAC) is used to prevent adversaries from tampering with the data. This is a secure channel, but not a tunnel yet. It is what goes inside the channel that makes the channel an IP tunnel.

5.2.2 Forwarding IP packets

Assume that a host U in one side of the channel wants to talk to host V in the other side. Packets will be generated, with the IP addresses in the header being U and V. Let us look at the direction from U to V first, to see how a packet can get from U to V securely. This packet cannot go through any arbitrary route when it passes through the Internet, or it will not be protected; it must go through the channel that is already established by the VPN client and server. The question is how to direct all the packets going to the other side of the private network to get to the VPN client first?

The answer is routing. Although both private networks form a single virtual private network, they will be configured accordingly, so each side still belongs to a different subnet. Let us

use `10.0.7.0/24` as the subnet for the client side, and use `10.0.8.0/24` as the subnet for the server side (see Figure 5.3). On the client side, we need to configure the routers, so all traffic going to `10.0.8.0/24` should be routed towards `VPN Client`. In the figure, it appears that `U` and `VPN Client` are on the same network, but this is not necessary. Both private networks can have a more complex network configuration that involves multiple subnets. Regardless of how complex the configuration is, all we need to do is to set up the routers, so all the `10.0.8.0/24`-bound traffic arrives at `VPN Client` first. Routers should be set up accordingly on the server side as well, so all the `10.0.7.0/24`-bound traffic arrives at `VPN Server` first.

After `VPN Client` receives a packet (from `U` to `V`), its job is to deliver the packet to `VPN Server` through the dedicated secure channel established by the tunnel application and its counterpart. The question is how the system can give the packet to an application that is running in the user space. This does not seem to be a hard problem, but it actually is. Let us see why.

When a packet arrives at `VPN Client`, it will go through the network stack in the kernel. Two reasons make it hard for the tunnel application to get the packet. First, the packet's destination is `V`, not `VPN Client`, so `VPN Client`, functioning as a router, will route the packet out, instead of giving the packet to its own application. Second, even if `VPN Client` decides to give the packet to the tunnel application, in a typical network stack, only the data part will be given to applications; both the IP header and the transport-layer header (TCP or UDP header) will be stripped away. To address these two problems, we may have to change the network stack, which is a not a desirable solution. IPSec VPNs do not have this problem, because their tunnels are established inside the kernel.

A good idea to solve the above problem is to make the tunnel application pretending to be a computer (instead of just an application); the "computer" connects to `VPN Client` through a network interface card (a virtual one). We then set up the routing table inside `VPN Client`, so all the `10.0.8.0/24`-bound packets will be routed to this "computer". Since the tunnel application gets the packets through the routing mechanism, it gets the entire packet, including the IP headers. The packet traverses through the network stack in a normal way, so there is no need to change the network stack.

How do we make an application to pretend to be a virtual computer that is connected to the host computer via a virtual network interface card? This is where we need a very important piece of technology, the TUN/TAP technology, which enables us to create virtual network interfaces. We will explain how this technology works in great details later in §5.3. At the high level, we just need to know that getting the entire IP packet from the kernel to applications is not an easy task, but it can be done via the TUN/TAP technology.

Once the tunnel application gets an IP packet, it passes the packet through the secure channel to its counterpart at the other end. Namely, the entire IP packet, encrypted, will be placed as the payload inside a TCP or UDP packet, so new headers will be added, including a transport-layer header and an IP header. Because we would like the new packet to reach `VPN Server`, the IP addresses in the new packet will be from `VPN Client` to `VPN Server`, while the encapsulated IP packet is still from `U` to `V`. We have an IP packet inside a different IP packet; that is why we call the channel an IP tunnel.

5.2.3 Releasing IP Packets

Once the new IP packet arrives at `VPN Server` through the tunnel, the network stack of `VPN Server` will strip off the new header, and give the payload, i.e., the encrypted IP packet, to the tunnel application. After decrypting the original IP packet and verifying its integrity, the tunnel

application needs to give it back to the kernel, where the packet will be routed out towards its final destination V. Here, we face another problem: how can an application give an IP packet to the system kernel? This is similar to the question of how an application can get an IP packet from the system kernel. The solution is the same. Using the TUN/TAP technology, the tunnel application, which functions like a computer, can get packets from and send packets to the system kernel.

At this point, we have successfully delivered the original IP packet from `VPN Client` to `VPN Server`, without exposing the packet to the untrusted outside world. When VPN server sees this packet, it sees that V is the destination IP, so it will route the packet out. As long as the routing tables are set up correctly within the private network `10.0.8.0/24`, the packet will eventually find its way to the final destination.

5.3 How TLS/SSL VPN Works: Details

As we have discussed in the previous section, the primary task for the tunnel application is to establish a TLS/SSL channel, get IP packets from the system and send them over the channel. Establishing the TLS/SSL channel and sending data through the channel is quite standard, and is covered in Chapter 12 (Transport Layer Security), so we will not repeat it this chapter. We will focus on how a tunnel application gets IP packets from the system. The enabling technology is the virtual network interfaces implemented by TUN/TAP. This section focuses on how to use TUN/TAP to implement TLS/SSL VPNs.

5.3.1 Virtual Network Interfaces

`Linux` and most operating systems support two types of network interface: physical and virtual. A physical network interface corresponds to physical Network Interface Card (NIC), which connects a computer to a physical network. A Virtual Network Interface (VIF) is a virtualized representation of computer network interfaces that may or may not correspond directly to a physical NIC. A familiar example of virtual network interface is the `loopback` device. Any traffic sent to this device is passed back to the kernel as if the packet comes from a network.

Another example of virtual network interface is the `TUN/TAP` interface. Like `loopback`, a `TUN/TAP` network interface is entirely virtual. User-space applications can interact with the virtual interfaces as if they were real. Unlike a physical network interface that connects a computer to a physical media, `TUN/TAP` interfaces connect a computer to a user-space program. They can be seen as a simple point-to-point network device, which connects two computers, except that one of the computers is only a user-space program that pretends to be a computer. Figure 5.4 illustrates the difference between physical and virtual network interfaces.

`TUN/TAP` consists of two types of interfaces, `TUN` interface and `TAP` interface. They are virtual interfaces at different levels.

- **`TUN` Interfaces:** `TUN` devices are created to work at the IP level or `OSI` layer 3 of the network stack. `TUN` devices support point-to-point (P2P) network communication by default, but they can be configured to support broadcast or multicast using flags during the creation. Sending any packet to the `TUN` interface will result in the packet being delivered to the user-space program, including the IP headers. We use `TUN` to build our VPN.
- **`TAP` Interfaces:** `TAP` devices, in contrast, work at the Ethernet level or `OSI` layer 2 and therefore behave very much like a network adaptor. Since they are running at layer 2,

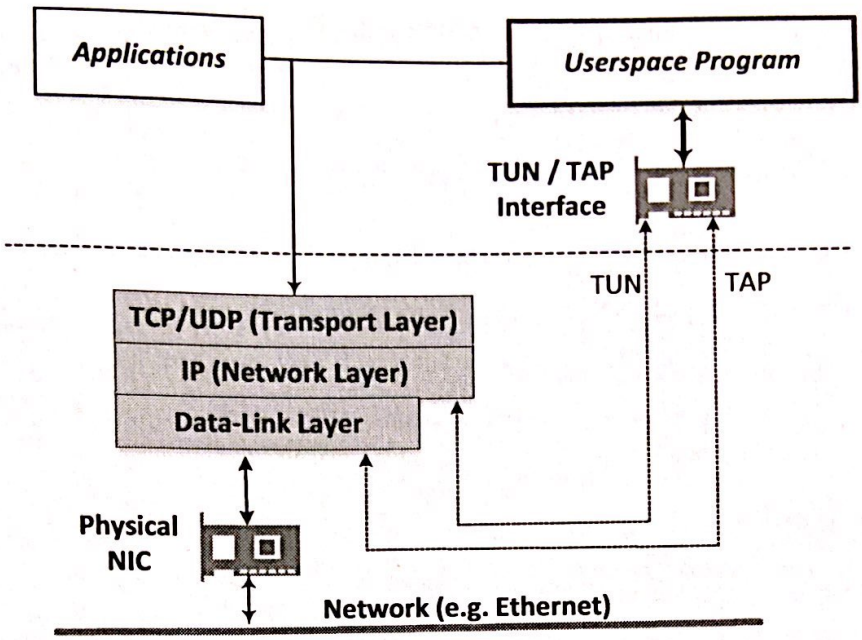


Figure 5.4: Virtual Network Interfaces

they can transport any layer 3 protocol and are not limited to point-to-point connections. A typical use of `TAP` devices is providing virtual network adaptors for multiple guest machines connecting to a physical device of the host machine. `TAP` interfaces are also used extensively for creating bridge networks because they can operate with Ethernet frames.

5.3.2 Creating a `TUN` Interface

A user-space program can create a `TUN/TAP` interface and attach itself to the virtual network interface. Packets sent by the operating system via the interface will be delivered to this user-space program. On the other hand, packets sent by the program via the virtual network interface are injected into the operating system's network stack. To the operating system, it appears that the packets come from an external source through the virtual network interface. Almost all modern `Linux` kernels come with pre-built support for `TUN/TAP` devices.

In order to use `TUN/TAP`, a program has to open the `/dev/net/tun` device and issue a corresponding `ioctl()` to register a network device with the kernel (see Lines ② and ③ of the sample code in Listing 5.1). A network device will appear as `tunNN` or `tapNN`, depending on the flags chosen (`NN` represents a number). The flag `IFF_TUN` set at Line ① specifies that we are creating a `TUN` device. It is important to note that in `Linux`, to create network devices, including `TUN/TAP` devices, a process needs to either be root or have the `CAP_NET_ADMIN` capability.

Listing 5.1: Code to create a `TUN` interface (`tundemo.c`)

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
```

```
#include <linux/if.h>
#include <linux/if_tun.h>
#include <sys/ioctl.h>

int createTunDevice()
{
   int tunfd;
   struct ifreq ifr;
   memset(&ifr, 0, sizeof(ifr));

   ifr.ifr_flags = IFF_TUN | IFF_NO_PI;   ①
   tunfd = open("/dev/net/tun", O_RDWR); ②
   ioctl(tunfd, TUNSETIFF, &ifr);         ③

   return tunfd;
}

int main () {
   int tunfd = createTunDevice();
   printf("TUN file descriptor: %d \n", tunfd);

   // We can interact with the device using this file descriptor.
   // In our experiement, we will do the interaction from a shell.
   // Therefore, we launch the bash shell here.
   char *argv[2];
   argv[0] = "/bin/bash"; argv[1] = NULL;
   execve("/bin/bash", argv, NULL);       ④

   return 0;
}
```

Once a TUN device is created, we can look it up using the `ifconfig` command. We need to use the `"-a"` option, or the command will only list the active interfaces. The newly created TUN interface will not be active until some further configuration is done. From the outcome, we can find the TUN interface. The first TUN interface will be called `tun0`, but if multiple TUN interfaces are created, they will be called `tun1`, `tun2`, etc.

```
$ ifconfig -a
tun0  Link encap:UNSPEC  HWaddr 00-00-00 ...
      POINTOPOINT NOARP MULTICAST  MTU:1500 ...
```

The virtual network interface needs to be configured before it can be used. First, we need to specify what network the interface is connected to. Second, we need to assign an IP address to the network interface. Finally, we will activate it. The first command in the following execution accomplishes these three tasks. It attaches the interface to the `10.0.8.0/24` network and assigns the IP address `10.0.8.99` to the interface. After running the command, if we run `ifconfig` again (without arguments), we can see the updated information on the interface.

```
$ sudo ifconfig tun0 10.0.8.99/24 up

$ ifconfig
tun0 Link encap:UNSPEC  HWaddr 00-00-00 ...
```

```
    inet addr: 10.0.8.99 P-t-P:10.0.8.99 Mask: 255.255.255.0
    UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1500 ...
```

It should be noted that the interface is transient in nature. That is, it will be destroyed when the process creating it terminates. There are ways to create a persistent `TUN` device, but that is out of the scope for the current discussion.

5.3.3 Routing Packets to a TUN Interface

Before we can use a `TUN` interface, we need to do one more thing. If we recall, the challenge we are trying to solve is to transfer all the packets between the system and tunneling application. Till now, we have discussed how to create a network interface and how to use it to interact with tunneling applications. We still need to make sure that the intended packets reach the `TUN` interface.

Before exploring this, let us see the movement of packets in further details using Figure 5.5. IP packets are generated by hosts on the private network (marked by ❷ and ❸); they are directed towards the gateway (one end of the VPN tunnel), where they are copied into the networking stack of the operating system. Now, the network stack decides where to direct the packets. This functionality is called routing and it is governed by a set of rules maintained in the routing table of the operating system.

Routing tables are traversed in a user-definable sequence until a matching route is found and the network stack will copy the packet into the buffer of the corresponding network interface. Since we want to send the IP packet to the `TUN` interface, we can simply add a rule to this table to let the network stack direct the packet towards the `TUN` interface (marked by ❶).

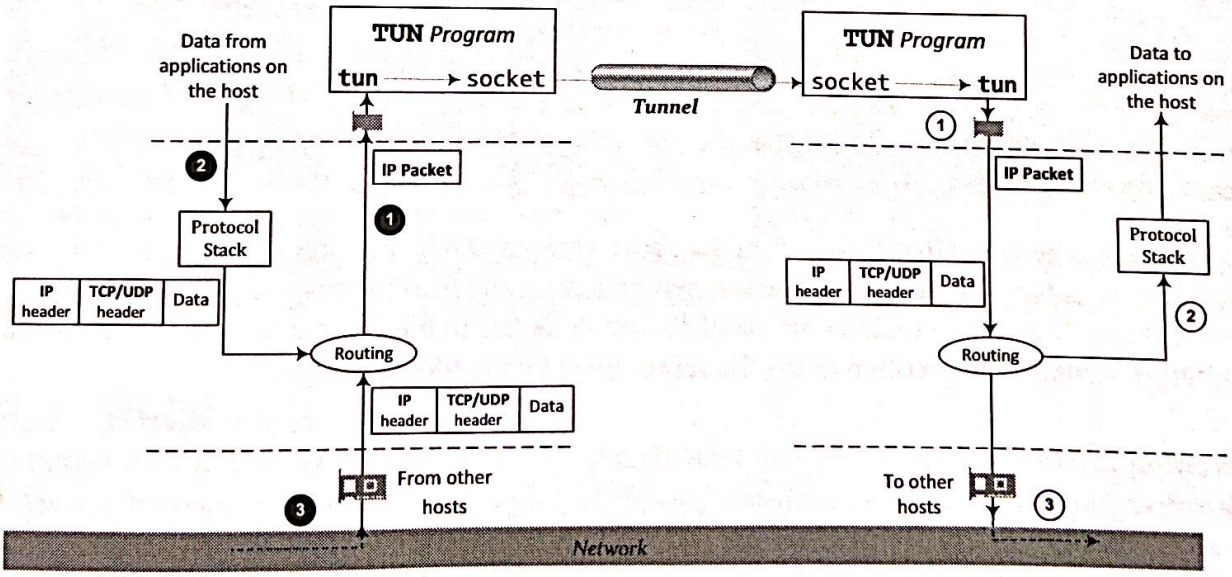


Figure 5.5: Detailed tunnel view

We can use the `route` command to add rules to the routing table. Our goal is to direct every packet destined to `10.0.8.0/24` to the `tun0` interface. The following command achieves the goal.

```
$ sudo route add -net 10.0.8.0/24 tun0
$ route -n
```

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         10.0.2.65       0.0.0.0         UG    100    0        0 enp0s3
10.0.2.64       0.0.0.0         255.255.255.192 U     100    0        0 enp0s3
10.0.8.0        0.0.0.0         255.255.255.0   U     0      0        0 tun0
```

5.3.4 Reading and Writing Operations on the TUN Interface

User-space programs can use the standard `read()` and `write()` system calls to receive packets from a virtual interface, or send packets to it.

Reading from TUN interface. Reading from a TUN interface gives us access to the entire IP packet, including the header. In the code presented in Listing 5.1, a shell is spawned at the end, and it can access the file descriptor `tunfd`. Through this file descriptor, we can read all the packets traveling to the server side. We did an experiment by sending a ping packet to `10.0.8.32`. Since this IP address belongs to the subnet `10.0.8.0/24`, the packet is delivered to the TUN interface, and from there to our program `xxd`, which is the command that we use to read the input and convert it into a hexdump. We can see that the output includes the IP header: `45` is usually the starting octet for most IPv4 packets; the packet is from `"0a 00 08 63"` to `"0a 00 08 20"`, i.e., from `10.0.8.99` to `10.0.8.32`. The result of our experiment is shown below.

```
$ sudo ./tundemo
TUN file descriptor: 3

# xxd <& 3
0000000: 4500 0054 0000 4000 4001 1627 0a00 0863  E..T..@.@..'...c
0000010: 0a00 0820 0800 3b19 10cf 0001 da1d 9f57  ... ..;........W
0000020: 439e 0400 0809 0a0b 0c0d 0e0f 1011 1213  C...............
0000030: 1415 1617 1819 1a1b 1c1d 1e1f 2021 2223  ............ !"#
0000040: 2425 2627 2829 2a2b 2c2d 2e2f 3031 3233  $%&'()*+,-./0123
0000050: 3435 3637
```

It should be noted that `"<& 3"` in the above command redirects the standard input device to file descriptor 3, so the `xxd` program can read from the file descriptor 3. In the following, we will use `">& 3"` to redirect the standard output device to file descriptor 3, so whatever the program prints will be written to the file represented by the file descriptor.

Writing to TUN interface. We can write data to TUN interfaces. Since the system kernel is expecting a packet from this interface, whatever we write to this interface needs to be a valid packet, or the system will report an error. We can create a valid packet using the same `xxd` command. Just copy and paste the above `xxd` output to a file called `hexfile`, and run `"xxd -r hexfile > packetfile"` to turn the data from the hexdump format back into the binary form; we save the result into a file called `packetfile`. We then write the content of the `packetfile` to the interface using the following command:

```
# cat packetfile >& 3
```

If we turn on the `Wireshark` sniffing program before running the above command, we can see an ICMP echo request packet from `10.0.8.99` to `10.0.8.32`, indicating whatever we wrote to the `TUN` interface has turned into a packet.

5.3.5 Forwarding Packets via the Tunnel

After a packet reaches the tunnel application via the `TUN` interface, it will be forwarded to the other side of the tunnel. Since the tunnel is built on top of TLS/SSL, the packet will be encrypted and tamper-proofing. TLS/SSL programming is covered in Chapter 12, so we will not get into the details on how to establish and use TLS/SSL in network communication. The tunnel application at the other end of the tunnel will extract the original IP packet from the tunnel, and then inject the IP packet back into the system kernel via the `TUN` interface. As we have discussed above, this is achieved by writing the IP packet to the `TUN` interface (see ① in Figure 5.5).

Once the packet is inside the kernel of `VPN Server`, like any other packet, it will go through routing. If the packet's destination is `VPN Server` itself, the packet will be sent to the transport layer, and eventually reach an application on `VPN Server`. In most cases, the packet's destination is another computer inside the private network, so `VPN Server` will route it out through one of the network interfaces (not including the `TUN` interface). Therefore, the packet will be released into the private network on the server side. From then on, the packet is not protected any more, but it is traveling inside a protected private network. Eventually, the packet will reach its final destination. The details are depicted in Figure 5.5 (marked by ② and ③).

5.3.6 Packet's Return Trip

When a packet arrives at its destination, usually a response packet will be generated and transmitted back to the sender. The packet flow in both directions are similar. We do need to ensure that the response packets also go through the same VPN tunnel, i.e., the response packets need to be routed towards `VPN Server`, and eventually to its `TUN` interface. The private network at the client side of the VPN has an IP prefix `10.0.7.0/24`, so we need to set up the routing tables on the server side of the VPN, directing all `10.0.7.0/24`-bound traffic towards the VPN tunnel, more specifically, towards the VPN server and its `TUN` interface.

5.4 Building a VPN

In this section, let us build a simple VPN program (both client and server), and use it to gain a first-hand experience on VPN. We would like to keep the code to the minimum, so it is easy for readers to understand and to write such a program themselves. In particular, we have done the following simplification: (1) Tunnels in real TLS/SSL VPN programs should be secured using the TLS/SSL protocol on top of TCP or UDP; our code only builds a UDP-based tunnel, leaving the task to secure the tunnel to readers. (2) Typically, when making a system call, we should check the return value to see whether the invocation is successful or not; we have removed error checkings from our code, so the code is not crowded by the error checking logic.

Figure 5.6(a) depicts the main flow of our VPN program, which first creates a `TUN` interface (already covered in the previous section), and works with the other end (client or server) to establish a tunnel using a UDP socket. After that, the program monitors the `TUN` interface and

the socket interface. If data comes from the TUN interface, it is a packet destined for the other side of the virtual private network, and should be protected before sending out to the Internet. VPN program will encrypt this packet (omitted in our code), and send the encrypted packet to the other end via the tunnel (Figure 5.6(b)). If data comes from the socket interface, it is a UDP datagram arriving from the other side of the virtual private network, and an IP packet is encapsulated in the payload. The VPN program retrieves the IP packet from the UDP payload, decrypts it, and then gives it to the kernel via the TUN interface (Figure 5.6(c)). The kernel will then route the packet out towards its final destination within the private network.

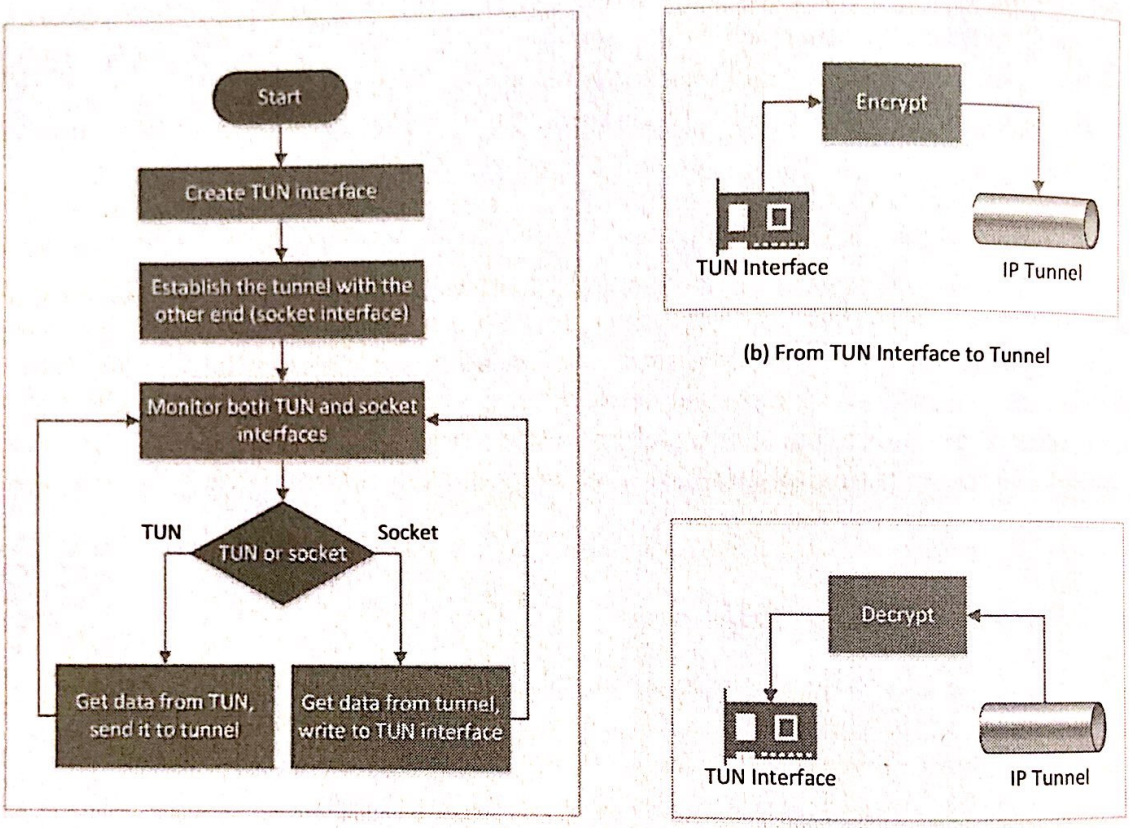


Figure 5.6: How a sample VPN program is implemented

5.4.1 Establish the Tunnel

Assuming that we are using a UDP-based tunnel. Setting up a socket for UDP communication is a simple step using the `socket()` API. On the server side, we need to bind the socket to a port number (see Line ① in Listing 5.2). Then, we can read the UDP data arriving at the port using `recvfrom()` (Line ②). Our simplified server code only supports one tunnel. It establishes a tunnel with whoever sends the UDP request first. Once a request is received, the server stores the client information in a variable `peerAddr`. For the sake of simplicity, we have also removed the authentication and encryption logic. In real VPN software, clients need to be authenticated by the server, and the communication channel needs to be encrypted.

Listing 5.2: UDP server (part of vpn_server.c)

```
#define PORT_NUMBER 55555
struct sockaddr_in peerAddr;

int initUDPServer() {
    int sockfd;
    struct sockaddr_in server;
    char buff[100];

    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    server.sin_port = htons(PORT_NUMBER);

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    bind(sockfd, (struct sockaddr*) &server, sizeof(server)); ①

    // Wait for the VPN client to "connect".
    bzero(buff, 100);
    int peerAddrLen = sizeof(struct sockaddr_in);
    int len = recvfrom(sockfd, buff, 100, 0,                    ②
                (struct sockaddr *) &peerAddr, &peerAddrLen);

    printf("Connected with the client: %s\n", buff);
    return sockfd;
}
```

On the client side, we want to send data through the UDP socket to a server. Similar to the server, we create a UDP socket first, and then use `sendto()` to send data to the server. Our simplified UDP client implementation is shown in Listing 5.3. It first sends a "hello" message to the server to establish the tunnel.

Listing 5.3: UDP client (part of vpn_client.c)

```
#define PORT_NUMBER 55555
#define SERVER_IP "10.0.2.69"
struct sockaddr_in peerAddr;

int connectToUDPServer(){
    int sockfd;
    char *hello="Hello";
    memset(&peerAddr, 0, sizeof(peerAddr));
    peerAddr.sin_family = AF_INET;
    peerAddr.sin_port = htons(PORT_NUMBER);
    peerAddr.sin_addr.s_addr = inet_addr(SERVER_IP);

    sockfd = socket(AF_INET, SOCK_DGRAM, 0);

    // Send a hello message to "connect" with the VPN server
    sendto(sockfd, hello, strlen(hello), 0,
                (struct sockaddr *) &peerAddr, sizeof(peerAddr));
    return sockfd;
}
```

5.4.2 Monitoring File Descriptors

Once the tunnel is established, there is no difference between client and server; they are simply two ends of a tunnel that stream data back and forth. The user-space VPN program has to handle two file descriptors simultaneously, one belonging to the TUN device and other to the socket. To transfer packets between these two file descriptors, we need to monitor both of them. One way to do that is to keep polling them, and see whether there are 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 are no data. When data become available, the process will be unblocked, and its execution will continue. This way, it does not waste CPU time when there is no data.

The read-based blocking mechanism works well for one interface. If a program is waiting on multiple interfaces, we cannot block on just one of the interfaces. We have to block on all of them altogether. Linux has a system call called `select()`, which allows a program to monitor multiple file descriptors simultaneously. To use `select()`, we need to store all the file descriptors to be monitored in a set using the `FD_SET` macro (see Lines ① and ② in Listing 5.4). We then give the set to the `select()` system call (Line ③), which will block the process until data are available on one of the file descriptors in the set. We can then use the `FD_ISSET` macro to figure out which file descriptor has received data. We use `select()` to monitor the TUN and socket file descriptors in the following code.

Listing 5.4: Using `select()` to monitor the TUN and socket descriptors

```
fd_set readFDSet;
int ret, sockfd, tunfd;

FD_ZERO(&readFDSet);
FD_SET(sockfd, &readFDSet);                                    ①
FD_SET(tunfd, &readFDSet);                                     ②
ret = select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL);        ③

if (FD_ISSET(sockfd, &readFDSet){
   // Read from sockfd and write to tunfd
}

if (FD_ISSET(tunfd, &readFDSet){
   // Read from tunfd and write to sockfd
}
```

5.4.3 From TUN To Tunnel

If the TUN interface has data, that means the kernel has forwarded an IP packet to the tunnel application via the TUN interface. We use `read()` to get the data—actually an IP packet—from the TUN interface, encrypt it, and then put the encrypted IP packet into the tunnel. Since our tunnel is UDP-based, putting the encrypted IP packet into the tunnel means putting it as the payload of a UDP packet to be sent to the other end of the tunnel. In our simplified code, we omit the encryption part, and simply put the original packet into the payload of a UDP packet. The code is described below.

Listing 5.5: From TUN to tunnel

```
#define BUFF_SIZE 2000
void tunSelected(int tunfd, int sockfd){
    int  len;
    char buff[BUFF_SIZE];

    printf("Got a packet from TUN\n");

    bzero(buff, BUFF_SIZE);
    len = read(tunfd, buff, BUFF_SIZE);
    sendto(sockfd, buff, len, 0, (struct sockaddr *) &peerAddr,
                  sizeof(peerAddr));
}
```

5.4.4 From Tunnel to TUN

If the socket interface has data, that means a UDP packet has just arrived from the other end of the tunnel. The payload of the UDP packet contains an encrypted IP packet from the other side of the private network. We retrieve the payload from the UDP packet using `recvfrom()`, decrypt it, and then inject the decrypted IP packet to the kernel via the `TUN` interface. The kernel will route the packet towards its final destination. We have omitted the decryption part in our code below.

Listing 5.6: From tunnel to TUN

```
#define BUFF_SIZE 2000
void socketSelected (int tunfd, int sockfd){
    int  len;
    char buff[BUFF_SIZE];

    printf("Got a packet from the tunnel\n");

    bzero(buff, BUFF_SIZE);
    len = recvfrom(sockfd, buff, BUFF_SIZE, 0, NULL, NULL);
    write(tunfd, buff, len);
}
```

5.4.5 Bring Everything Together

Let us put all the pieces together in our `main()` function. The complete code is shown below. After the interface is created and the tunnel is established, the program enters an infinite loop. Inside the loop, it blocks at the `TUN` and socket interface. After being unblocked, it relays data between the two interfaces.

Listing 5.7: VPN client program (`vpn_client.c`)

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <fcntl.h>
#include <arpa/inet.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <sys/ioctl.h>

#define PORT_NUMBER 55555
#define SERVER_IP "10.0.2.69"
#define BUFF_SIZE 2000
struct sockaddr_in peerAddr;

int createTunDevice() { See Listing 5.1 }
int connectToUDPServer() { See Listing 5.3 }
void tunSelected(int tunfd, int sockfd) { See Listing 5.5 }
void socketSelected (int tunfd, int sockfd) { See Listing 5.6 }

int main (int argc, char * argv[]) {
   int tunfd, sockfd;

   tunfd  = createTunDevice();
   sockfd = connectToUDPServer();

   while (1) {
     fd_set readFDSet;
     FD_ZERO(&readFDSet);
     FD_SET(sockfd, &readFDSet);
     FD_SET(tunfd, &readFDSet);
     select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL);

     if (FD_ISSET(tunfd,  &readFDSet)) tunSelected(tunfd, sockfd);
     if (FD_ISSET(sockfd, &readFDSet)) socketSelected(tunfd, sockfd);
   }
}
```

5.5 Setting Up a VPN

In the previous section, we have already developed a simple tunneling program. In this section, we will use it to set up a real VPN. Our goal is to see a VPN in action and get a first-hand experience. Although we can use some existing and more sophisticated VPN products, such as OpenVPN [openvpn.net, 2017], these products tend to hide the technical details, making it difficult for us to see what is actually going on under the hood.

5.5.1 Network Configuration

Figure 5.3 depicts a general VPN setup, where the client and server sides have their own private network. This is a setup for connecting multiple private networks of an organization. When a user, traveling or working from home, uses VPN, the setup is more similar to Figure 5.7, where there is only one computer (Host U) on the client side, which serves as both `VPN Client` and user's working machine. This is a special case of what is depicted in Figure 5.3, but it is a very common setup. In our experiment, we will use the setup in Figure 5.7.

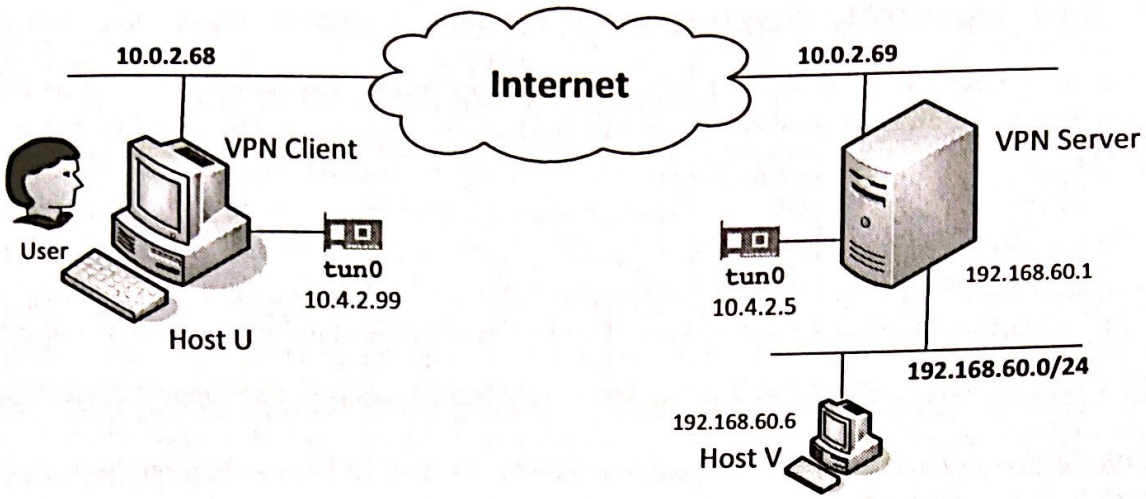


Figure 5.7: Network setup for a VPN.

Three VMs (running on the same host computer) are used in the experiment setup: Host U (also serving as `VPN Client`), `VPN Server`, and Host V. The `VPN Client` and `VPN Server` VMs are supposed to communicate with each other over the Internet, but setting up that will not be easy using one host computer. In our experiment, we directly connect these two VMs to the same network to emulate the Internet. These two VMs are attached to the same `"NAT Network"` adapter in VirtualBox.

We created a private network using an `"Internal Network"` adapter in VirtualBox. The IP Prefix of this network is `192.168.60.0/24`, and both `VPN Server` and Host V are connected to it. This private network is not exposed to the outside, so only computers inside can access Host V. Our goal is to help `User` on Host U to securely communicate with Host V over the emulated Internet. Basically, we will set up a VPN from Host U to `VPN Server`, allowing Host U to access Host V via the VPN tunnel.

It should be noted that in VirtualBox, the `"Internal Network"` adapter does not have a DHCP associated with it, so VMs attached to this type of adapter do not get their IP addresses automatically; we need to manually assign a static IP address to each VM. We run the following commands on Host V, which has only one network interface (`enp0s3`). The first command assign `192.168.60.6` to this interface, while the second command sets the default gateway to `192.168.60.1`:

```
$ sudo ifconfig enp0s3 192.168.60.6/24 up
$ sudo route add default gw 192.168.60.1
```

`VPN Server` has two network interfaces, one is attached to `"NAT Network"` adapter, and the other is attached to the `"Internal Network"` adapter. The first one already has an IP address, so we only need to assign an IP address to the second one (we can use `ifconfig` to find out which one does not have an IP address). On our machine, this interface is `enp0s8`. The following command assign `192.168.60.1` to this interface.

```
$ sudo ifconfig enp0s8 192.168.60.1/24 up
```

5.5.2 Configure VPN Server

VPN Server needs to forward IP packets to other machines, essentially serving as a router. Most computers, including our VMs, are configured as a host, not a router. The difference is that, unlike routers, a host machine does not forward other hosts' packets, so if an incoming packet is not for the host, the host "thinks" that there must be a mistake, because it is not a designated router, and should not be able to receive packets destined for other machines. Therefore, the host simply drops the packet. We can change its behavior by turning on IP Forwarding, so the host will forward other machine's packets. The following command turns on IP Forwarding.

```
$ sudo sysctl -w net.ipv4.ip_forward=1
```

In the setup depicted in Figure 5.7, we only need to turn on IP Forwarding on VPN Server; however, if the setup is like what is shown in Figure 5.3, we also need to do it on the client side, because VPN Client also needs to forward packets.

On VPN Server, we first run the server program described in the previous section. The program creates a TUN interface (tun0 in our case), and then waits for a tunnel connection request from a client. We need to configure the tun0 interface first, just like what we did in §5.3. We decide to use 10.4.2.0/24 as the IP prefix for the TUN interface (for both VPN Client and VPN Server). The IP address assigned to the TUN interface on VPN Server is 10.4.2.5. The following two commands assign the IP address to the tun0, bring it up, and then add a corresponding route to the routing table.

```
$ sudo ifconfig tun0 10.4.2.5/24 up
$ sudo route add -net 10.4.2.0/24 tun0
```

5.5.3 Configure VPN Client

On VPN Client, we first run the client program described in the previous section. The program creates a TUN interface (tun0 in our case), and then sends a "hello" message to VPN Server to establish a VPN tunnel. Similar to the setup on the server side, we assign 10.4.2.99 to the TUN interface at the client side, and add a route for the 10.4.2.0/24 network. This is not enough. We also need to tell VPN Client that all the traffic to the private network 192.168.60.0/24 should go through the tunnel. The third command in the following listing adds a route, so all packets for 192.168.60.0/24 are routed to the tun0 interface, from where they will be sent through the VPN tunnel.

```
$ sudo ifconfig tun0 10.4.2.99/24 up
$ sudo route add -net 10.4.2.0/24 tun0
$ sudo route add -net 192.168.60.0/24 tun0
```

5.5.4 Configure Host V

We need to configure Host V so when it replies to Host U, the reply packets can get back to Host U. For that purpose, we need to answer the following question: when a packet is sent from Host U to Host V, what is the source IP address of the packet? Host U has at least two network interfaces, a real ethernet network interface and a virtual TUN interface, each having its own IP address. Which address should be used when a packet is created on Host U? When a computer has multiple network interfaces (and hence multiple IP addresses), the source IP address of its

packet is decided based on which network interface is used to send out the packet. In our case, packets from Host U to Host V go out via the tun0 interface on Host U, so the source IP will be 10.4.2.99, which is the IP address assigned to tun0.

When Host V sends reply packets to Host U, the destination IP address of the packet will be 10.4.2.99, but since Host V knows nothing about the 10.4.2.0/24 network, it does not know where to send the packet. More importantly, reply packets cannot go back via any arbitrary route; they must go back from the same VPN tunnel; otherwise they will not be protected by the tunnel. To ensure that, we need to route all the packets for the 10.4.2.0/24 network toward the tunnel. For Host V, we route such packets to VPN Server, i.e., packets going to 10.4.2.0/24 should be routed toward the router 192.168.60.5. We add the following routing entry to Host V (Host V connects to the 192.168.60.0/24 network via the enp0s3 network interface):

```
$ sudo route add -net 10.4.2.0/24 gw 192.168.60.5 enp0s3
```

5.6 Testing VPN

After we have set up everything, our VPN will start working. Host U can now access Host V; all the network traffic between Host U and Host V will go through a tunnel between VPN Client and VPN Server. We test our VPN using two commands: ping and telnet.

5.6.1 Ping Test

We ping Host V from Host U. Before the VPN is established, there would be no response from Host V, because it is not reachable from Host U. After the VPN is set up, we can see the following result:

```
seed@User(10.0.2.68):$ ping 192.168.60.6
PING 192.168.60.6 (192.168.60.6) 56(84) bytes of data.
64 bytes from 192.168.60.6: icmp_req=1 ttl=63 time=2.41 ms
64 bytes from 192.168.60.6: icmp_req=2 ttl=63 time=1.48 ms
```

Figure 5.8 shows the packets generated when we ping Host V (192.168.60.6). We have captured the traffic on all the interfaces of VPN Client using Wireshark. Packet No. 1 is generated by the ping command. Due to the routing setup, the ICMP packet is routed to the TUN interface. That is why the source IP is 10.4.2.99, the one assigned to tun0. The tunnel application gets the ICMP packet, and then feeds it into its tunnel, i.e., putting it inside a UDP packet (Packet No. 2) towards VPN Server (10.0.2.69). Since this UDP packet goes out from VPN Client's normal interface, its source IP address is 10.0.2.68, which is VPN Client's ethernet address.

Packet No. 3 is the return UDP packet from VPN Server, inside which there is an encapsulated ICMP echo reply packet from 192.168.60.6. The tunnel application on VPN Client gets this UDP packet, and takes out the encapsulated ICMP packet, and gives it to the kernel via the tun0 interface. That becomes Packet No. 4. The computer realizes that the destination IP address 10.4.2.99 is its own, so it passes the ICMP echo reply message to the ping program. Packets No. 5 to 8 are triggered by another ICMP echo request message.

No.	Source	Destination	Protocol	Info
1	10.4.2.99	192.168.60.6	ICMP	Echo (ping) request id=0x0286,
2	10.0.2.68	10.0.2.69	UDP	54915 → 55555 Len=84
3	10.0.2.69	10.0.2.68	UDP	55555 → 54915 Len=84
4	192.168.60.6	10.4.2.99	ICMP	Echo (ping) reply id=0x0286,
5	10.4.2.99	192.168.60.6	ICMP	Echo (ping) request id=0x0286,
6	10.0.2.68	10.0.2.69	UDP	54915 → 55555 Len=84
7	10.0.2.69	10.0.2.68	UDP	55555 → 54915 Len=84
8	192.168.60.6	10.4.2.99	ICMP	Echo (ping) reply id=0x0286,

Figure 5.8: Packets generated when pinging Host V from Host U

5.6.2 Telnet Test

We use `telnet` to test whether TCP works over the VPN. The following result shows that we can successfully connect to the `telnet` server on Host V.

```
seed@User(10.0.2.68):$ telnet 192.168.60.6
Trying 192.168.60.6...
Connected to 192.168.60.6.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
ubuntu login: seed
Password:
  ......
seed@HostV(192.168.60.6):$          ← Successfully logged in!
```

After we successfully `telnet` to Host V, we break the VPN tunnel by stopping the VPN program on the server side. Immediately, the `telnet` program becomes unresponsive. Whatever we type in the `telnet` program does not show up, and it seems that the program freezes. Actually, `telnet` is still working, but since the packets it sends out via the broken VPN tunnel goes nowhere, TCP, which is the underlying transport-layer protocol used by `telnet`, will keep resending packets. These TCP retransmissions will be encapsulated into new IP packets, and be sent via the UDP channel to port `55555` on `VPN Server`. However, since the server program has stopped, no application is listening on this port, `VPN Server` will drop these UDP packets, and send back ICMP error messages, telling `VPN Client` that the port is not reachable. That is why we see multiple ICMP error messages in Figure 5.9.

No.	Source	Destination	Protocol	Info
32	10.4.2.99	192.168.60.6	TELNET	Telnet Data ...
33	10.0.2.68	10.0.2.69	UDP	37674 → 55555 Len=54
34	10.0.2.69	10.0.2.68	ICMP	Destination unreachable (Port unreachable)
35	10.4.2.99	192.168.60.6	TCP	[TCP Retransmission] 45654 → 23 [PSH, ACK] Seq=340884658
36	10.0.2.68	10.0.2.69	UDP	37674 → 55555 Len=54
37	10.0.2.69	10.0.2.68	ICMP	Destination unreachable (Port unreachable)
38	10.4.2.99	192.168.60.6	TCP	[TCP Retransmission] 45654 → 23 [PSH, ACK] Seq=340884658
39	10.0.2.68	10.0.2.69	UDP	37674 → 55555 Len=54
40	10.0.2.69	10.0.2.68	ICMP	Destination unreachable (Port unreachable)
41	10.4.2.99	192.168.60.6	TCP	[TCP Retransmission] 45654 → 23 [PSH, ACK] Seq=340884658
42	10.0.2.68	10.0.2.69	UDP	37674 → 55555 Len=54
43	10.0.2.69	10.0.2.68	ICMP	Destination unreachable (Port unreachable)
44	10.4.2.99	192.168.60.6	TCP	[TCP Retransmission] 45654 → 23 [PSH, ACK] Seq=340884658

Figure 5.9: Network traffic after we break up a VPN connection

Whatever we have typed blindly into `telnet` are actually not lost; they are buffered, waiting to be sent to the `telnet` server. When the server receives a character, it echoes the character back to the `telnet` client, which will then print out the character to the terminal. That is how a character typed by a user is displayed, so if a character cannot reach the `telnet` server, it will not be printed out on the client side.

If we now reconnect the VPN tunnel, those characters that we typed blindly into `telnet` will eventually reach the `telnet` server due to TCP re-transmission, and all these characters will suddenly show up on the client side.

5.7 Using VPN to Bypass Egress Firewall

VPN was originally developed for security purposes, but interestingly, nowadays, it has been widely used to defeat another security solution, i.e., bypassing firewalls. Many organizations or countries conduct egress filtering on their firewalls to prevent their internal users from accessing certain websites, for a variety of reasons, including discipline, safety, and politics. For instance, many K-12 schools in the United States block social network sites, such as Facebook, from their networks, so students do not get distracted during school hours. Another example is the "Great Firewall of China", which blocks a number of popular sites, including Google, YouTube and Facebook.

These firewalls typically inspect the destination address of each outgoing packet; if the address is on their blacklist, the packet will be dropped. Some firewalls also inspect packet payloads. Therefore, these firewalls only work if they can see the actual addresses and payloads. If we can hide those data items, we can bypass firewalls. VPN becomes a very natural solution for this purpose because it can hide a disallowed packet inside a packet that is allowed. The actual packet becomes the encrypted payload inside the allowed packet, so firewalls cannot tell what is inside. In this section, we use our own VPN program to explain and demonstrate how VPNs can be actually used to bypass firewalls.

5.7.1 Network Setup

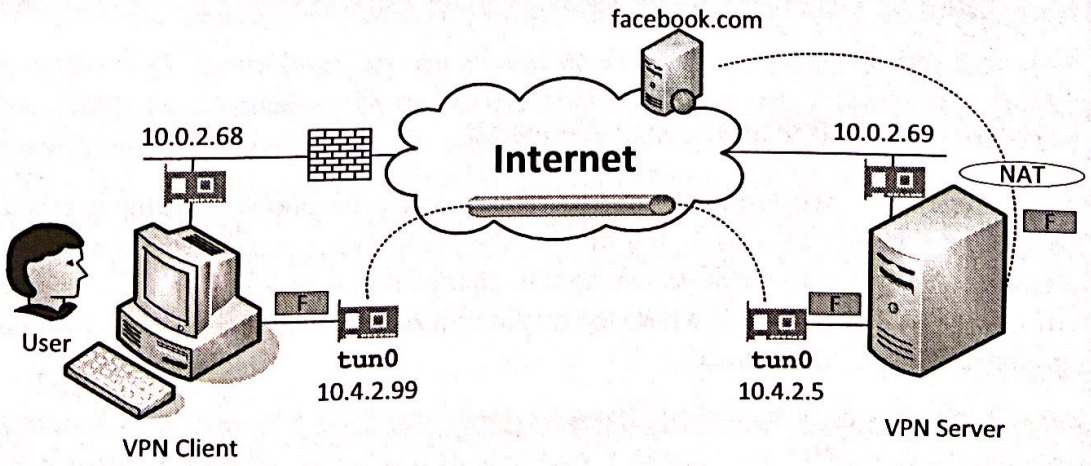


Figure 5.10: Bypassing firewall using VPN

We only need two VMs, the user machine (also serving as VPN Client) and VPN Server. Figure 5.10 shows the network setup. There is a firewall between the user machine and the Internet, blocking access to Facebook. In our experiment setup, instead of using a dedicated VM as the firewall, we simply set up the firewall on the user machine. It should be noted that Facebook has many IP prefixes, so a complete blocking requires blocking all these IP prefixes. One can run the following command to get the list of IP prefixes owned by Facebook (AS32934 is the Autonomous System number assigned to Facebook):

```
$ whois -h whois.radb.net -- '-i origin AS32934'
```

For the experiment purpose, there is no need to block all those IP prefixes. We simply run the dig command: "dig www.facebook.com" several times, and get the IP addresses returned by Facebook's DNS server. These IP addresses may change, but over a short period of time, they are quite stable, especially their IP prefixes. We get both 31.13.71.36 and 31.13.74.36 quit consistently during our experiment, so we will block 31.13.0.0/16 in our setup. We use iptables to set up firewall rules.

```
$ sudo iptables -A OUTPUT -d 31.13.0.0/16 -o enp0s3 -j DROP
```

In the commands listed above, we block all the outgoing traffic to 31.13.0.0/16. It should be noted that we have to limit our blocking to the network interface enp0s3, which is the one used for the Internet traffic on the user machine. If we do not specify an interface, the rule will be applied to all the interfaces, including our TUN interface, preventing the VPN tunnel application from even getting the packet. After this step, if we visit Facebook from a browser, we cannot reach the web site. If you can still see the Facebook page, the content very likely comes from the browser's cache; you need to go to your browser to clear all the cached data. We can use ping to check whether we can still access Facebook. From the results shown below, we can see that Facebook is blocked from our computer.

```
seed@User(10.0.2.68):~$ ping www.facebook.com
PING star-mini.c10r.facebook.com (31.13.71.36) 56(84) bytes of data.
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
```

5.7.2 Setting Up VPN to Bypass Firewall

Now, let us set a VPN to bypass the firewall, so we can access Facebook from the user machine. The VPN setup is the same as what is described in the previous section. In addition, we need to add a routing entry to the user machine, asking it to change the route for all the Facebook traffic: instead of going through enp0s3, where the traffic will be blocked, Facebook-bound packets should go through the TUN interface.

```
$ sudo route add -net 31.13.0.0/24 tun0
```

After the above setup, if we go visit Facebook from the user machine, we can see that packets are now going through our tunnel, and can successfully reach VPN Server. Although the tunnel traffic (UDP packets) still goes through enp0s3, which is the only interface for packets

to get out to the Internet from the user machine, the firewall does not block these packets because their destination is VPN Server, not Facebook. The following ping command shows that our packets to Facebook have successfully passed the firewall. However, nothing has come back from Facebook at this point.

```
seed@User(10.0.2.68):~$ ping www.facebook.com
PING star-mini.c10r.facebook.com (31.13.71.36) 56(84) bytes of data.
^C
--- star-mini.c10r.facebook.com ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 5016ms
```

From the wireshark trace, we can see that our VPN Server has actually forwarded the packets to Facebook, and Facebook has also replied, but the reply never reaches VPN Server. The reason is the source IP address of the packet. As we have discussed before, when a packet is sent out via the VPN tunnel from the user machine, the packet takes the TUN interface's IP address as its source IP address, which is 10.4.2.99 in our setup. When this packet is sent out by VPN Server over its enp0s3 interface (the one attached to the "NAT Network" adapter in VirtualBox), it will go through VirtualBox's NAT (Network Address Translation) server, where the source IP address will be replaced by the IP address of the host computer. The packet will eventually arrive at Facebook, and the reply packet will come back to our host computer, and then be given to the same NAT server, where the destination address is translated back 10.4.2.99. This is where we will encounter a problem.

VirtualBox's NAT server knows nothing about the 10.4.2.0/24 network, because this is the one that we create for our TUN interface, so it has no idea where this packet should go, much less knowing that the packet should be given to VPN Server. As a result, the reply packet from Facebook will be dropped. That is why we do not see anything back from the VPN tunnel.

To solve this problem, we will set up our own NAT server on VPN Server, so when packets from 10.4.2.99 go out, their source IP addresses are always replaced by VPN Server's IP address (10.0.2.69). We can use the following commands to create a NAT server on the enp0s3 interface of VPN Server (the first two commands delete all the existing rules if there is any).

```
$ sudo iptables -F
$ sudo iptables -t nat -F
$ sudo iptables -t nat -A POSTROUTING -j MASQUERADE -o enp0s3
```

Once the NAT server is set up, we should be able to connect to Facebook from the user machine. We have successfully bypassed the firewall. This is pretty much how a VPN is used to bypass firewalls. In the real world, once a server is identified as being used for bypassing firewalls, it will be put on the blacklist. Therefore, VPN servers used for firewall-bypassing purposes need to change their addresses constantly, so if one server gets blocked, another one will come up with a different IP address. This is a like a never-ending cat-and-mouse game.

5.8 Summary

VPN enables us to build a virtual private network over a public network, such that the traffic among the peers within this virtual network are protected, even though the traffic goes through unprotected public networks. There are two typical techniques to build a VPN, using IPSec and using TLS/SSL, with the latter approach becoming more popular and being a technically