Virtual Private Network (VPN) Lab

Copyright © 2006 - 2014 Wenliang Du, Syracuse University.
The development of this document is/was funded by three grants from the US National Science Foundation: Awards No. 0231122 and 0618680 from TUES/CCLI and Award No. 1017771 from Trustworthy Computing.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation. A copy of the license can be found at http://www.gnu.org/licenses/fdl.html.

1 Overview

A Virtual Private Network (VPN) is used for creating a private scope of computer communications or providing a secure extension of a private network into an insecure network such as the Internet. VPN is a widely used security technology. VPN can be built upon IPSec or Secure Socket Layer (SSL). These are two fundamentally different approaches for building VPNs. In this lab, we focus on the SSL-based VPNs. This type of VPNs is often referred to as SSL VPNs.

The learning objective of this lab is for students to master the network and security technologies underlying SSL VPNs. The design and implementation of SSL VPNs exemplify a number of security principles and technologies, including crypto, integrity, authentication, key management, key exchange, and Public-Key Infrastructure (PKI). To achieve this goal, students will implement a simple SSL VPN for `Ubuntu`.

2 Lab Environment

We need to use OpenSSL package in this lab. The package includes the header files, libraries, and commands. The package was not installed in our pre-built VM image, but it can be easily installed using the following command.

```
$ apt-get source openssl
```

After downloading the source package, unpack the `.tar.gz` file, and then follow the standard steps (`"./config"`, `"make"`, `"make install"`) to build and install the OpenSSL package. Read the `README` and `INSTALL` files in the package for detailed instructions.

Note for Instructors

If the instructor plans to hold lab sessions for this lab, we suggest that the following background information be covered in the lab sessions:

1. How to write programs using OpenSSL libraries.
2. How to use the TUN/TAP devices to set up network tunnel.
3. How to use the `route` and `ifconfig` commands.
4. How to do port forwarding in VirtualBox (or other virtual machine software).
5. How to use VirtualBox to set up a network using several VMs.

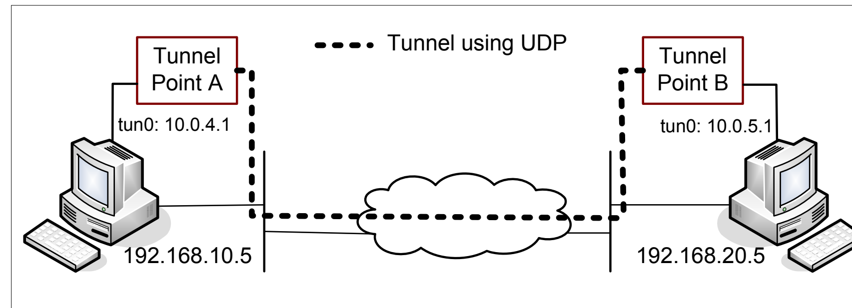


Figure 1: Host-to-Host Tunnel

3 Lab Tasks

In this lab, students need to implement a simple VPN for `Linux`. We will call it `miniVPN`.

3.1 Task 1: Create a Host-to-Host Tunnel using TUN/TAP

The enabling technology for the TLS/SSL VPNs is TUN/TAP, which is now widely implemented in modern operating systems. TUN and TAP are virtual network kernel drivers; they implement network device that are supported entirely in software. TAP (as in network tap) simulates an Ethernet device and it operates with layer-2 packets such as Ethernet frames; 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/TAP virtual network interface. Packets sent by an operating system via a TUN/TAP network interface are delivered to the user-space program. On the other hand, packets sent by the program via a TUN/TAP network interface are injected into the operating system netwrok stack; to the operating system, it appears that the packets come from an external source through the virtual network interface.

When a program is attached to a TUN/TAP interface, the IP packets that the computer sends to this interface will be piped into the program; on the other hand, the IP packets that the program sends to the interface will be piped into the computer, as if they came from the outside through this virtual network interface. The program can use the standard `read()` and `write()` system calls to receive packets from or send packets to the virtual interface.

Davide Brini has written an excellent tutorial article on how to use TUN/TAP to create a tunnel between two machines. The URL of the tutorial is `http://backreference.org/2010/03/26/tuntap-interface-tutorial`. The tutorial provides a program called `simpletun`, which connects two computers using the TUN tunneling technique. Students should go read this tutorial. When you compile the sample code from the tutorial, if you see error messages regarding `linux/if.h`, try to change `"<linux/if.h>"` to `"<net/if.h>"` in the `include` statement.

For the convenience of this lab, we have modified the Brini's `simpletun` program, and linked the code in the lab web page. Students can simply download this C program and run the following command to compile it. We will use `simpletun` to create tunnels in this lab:

```
$ gcc -o simpletun simpletun.c
```

Creating Host-to-Host Tunnel. The following procedure shows how to create a host-to-host tunnel using the `simpletun` program. The `simpletun` program can run as both a client and a server. When it is running with the `-s` flag, it acts as a server; when it is running with the `-c` flag, it acts as a client.

1. **Launch two virtual machines.** For this task, we will launch these two VMs on the same host machine. The IP addresses for the two machines are `192.168.10.5`, and `192.168.20.5`, respectively (you can choose any IP addresses you like). See the configuration in Figure 1.

2. **Tunnel Point A:** we use Tunnel Point A as the server side of the tunnel. Point A is on machine `192.168.10.5` (see Figure 1). It should be noted that the client/server concept is only meaningful when establishing the connection between the two ends. Once the tunnel is established, there is no difference between client and server; they are simply two ends of a tunnel. We run the following command (the `-d` flag asks the program to print out the debugging information):

   ```
   On Machine 192.168.10.5:
   # ./simpletun -i tun0 -s -d
   ```

   After the above step, your computer now have multiple network interface, one is its own Ethernet card interface, and the other is the virtual network interface called `tun0`. This new interface is not yet configured, so we need to configure it by assigning an IP address. We use the IP address from the reserved IP address space (`10.0.0.0/8`).

   It should be noted that the above command will block and wait for connections, so, we need to find another window to configure the `tun0` interface. Run the following commands (the first command will assign an IP address to the interface `"tun0"`, and the second command will bring up the interface):

   ```
   On Machine 192.168.10.5:
   # ip addr add 10.0.4.1/24 dev tun0
   # ifconfig tun0 up
   ```

3. **Tunnel Point B:** we use Tunnel Point B as the client side of the tunnel. Point B is on machine `192.168.20.5` (see Figure 1). We run the following command on this machine (The first command will connect to the server program running on `192.168.10.5`, which is the machine that runs the Tunnel Point A. This command will block as well, so we need to find another window for the second and the third commands):

   ```
   On Machine 192.168.20.5:
   # ./simpletun -i tun0 -c 192.168.10.5 -d
   # ip addr add 10.0.5.1/24 dev tun0
   # ifconfig tun0 up
   ```

4. **Routing Path:** After the above two steps, the tunnel will be established. Before we can use the tunnel, we need to set up the routing path on both machines to direct the intended outgoing traffic through the tunnel. The following routing table entry directs all the packets to the `10.0.5.0/24` network (`10.0.4.0/24` network for the second command) through the interface `tun0`, from where the packet will be hauled through the tunnel.

   ```
   On Machine 192.168.10.5:
   # route add -net 10.0.5.0 netmask 255.255.255.0 dev tun0

   On Machine 192.168.20.5:
   # route add -net 10.0.4.0 netmask 255.255.255.0 dev tun0
   ```

5. **Using Tunnel:** Now we can access `10.0.5.1` from `192.168.10.5` (and similarly access `10.0.4.1` from `192.168.20.5`). We can test the tunnel using `ping` and `ssh` (note: do not forget to start the `ssh` server first):

```
On Machine 192.168.10.5:
$ ping 10.0.5.1
$ ssh 10.0.5.1

On Machine 192.168.20.5:
$ ping 10.0.4.1
$ ssh 10.0.4.1
```

UDP Tunnel: The connection used in the `simpletun` program is a TCP connection, but our VPN tunnel needs to use UDP. Therefore you need to modify `simpletun` and turn the TCP tunnel into a UDP tunnel. You need to think about why it is better to use UDP in the tunnel, instead of TCP. Please write your answer in the lab report.

3.2 Task 2: Create a Host-to-Gateway Tunnel

Now you have succeeded in setting up the tunnel on the two VMs within a single host machine, you should set up a similar tunnel on two VMs on two different host machines. You can use port forwarding for this purpose. Please see the guidelines regarding port forwarding.

In this task, you need to create a tunnel between a computer and a gateway, allowing the computer to access the private network that is connected to the gateway. To demonstrate this, you need two physical computers. On one computer, you run several VMs within the computer to set up the gateway and the private network. You then use a VM in the other computer to communicate to the hosts on the private network. Please refer to the guideline section to see how to set up the gateway and the private network. Because you need two physical computers, you can team up with another student if you have only one computer. However, you must do your work independently. Teaming up is only for the demonstration purpose.

3.3 Task 3 Create a Gateway-to-Gateway Tunnel

In this task, you need to go a step further to establish a tunnel between two gateways of different private networks. With this tunnel, any host from one private network can communicate with the hosts on the other private network using the tunnel. The setup for such a gateway-to-gateway tunnel is depicted in Figure 2.

3.4 Task 4: Create a Virtual Private Network (VPN)

At this point, you have learned how to create a network tunnel. Now, if you can secure this tunnel, you will essentially get a VPN. This is what we are going to achieve in this task. To secure this tunnel, we need to achieve two goals, confidentiality and integrity. The confidentiality is achieved using encryption, i.e., the contents that go through the tunnel is encrypted. A real VPN software usually supports a number of different encryption algorithm. For the `MiniVPN` in this lab, we only need to support the AES encryption algorithm, and we use the Cipher Block Chaining (CBC) mode.

The integrity goal ensures that nobody can tamper with the traffic in the tunnel or launch a replay attack. Integrity can be achieved using various methods. In this lab, we only need to support the Message Authentication Code (MAC) method. The AES encryption algorithm and the HMAC-SHA256 algorithm

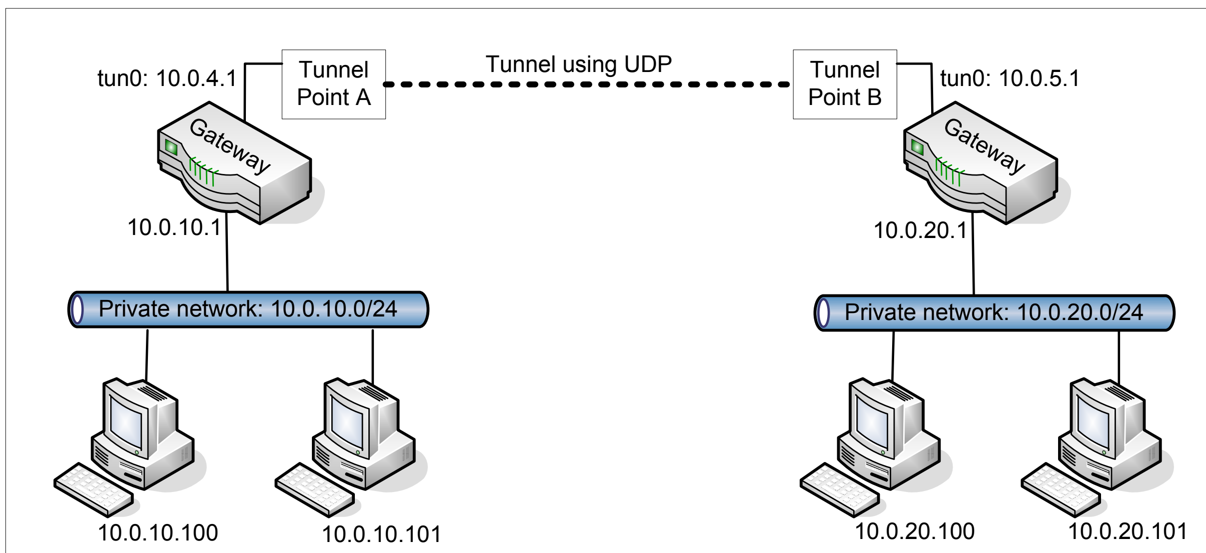


Figure 2: Gateway to Gateway

are both implemented in the OpenSSL library. There are plenty of online documents explaining how to use the OpenSSL's crypto libraries.

Both encryption and MAC need a secret key. Although the keys can be different for encryption and MAC, for the sake of simplicity, we assume that the same key is used. This key has to be agreed upon by both sides of the VPN. For this task, we assume that the key is already provided. Agreeing upon the key will be implemented in the next task.

For encryption, the client and the server also need to agree upon an Initial Vector (IV). For security purpose, you should not hard-code the IV in your code. The IV should be randomly generated for each VPN tunnel. Agreeing upon the IV will also be implemented in the next task.

3.5 Task 5: Authentication and Key Exchange

Before a VPN is established, the VPN client must authenticate the VPN server, making sure that the server is not a fraudulent one. On the other hand, the VPN server must authenticate the client (i.e. user), making sure that the user has the permission to create such a VPN tunnel. After the authentication is done, the client and the server will agree upon a session key for the VPN tunnel. This session key is only known to the client and the server. The process of deriving this session key is called key exchange.

Step 1: Authenticating VPN Server A typical way to authenticate the server is to use public-key certificates. The VPN server needs to first get a public-key certificate from a Certificate Authority (CA), such as Verisign. When the client makes the connection to the VPN server, the server will use the certificate to prove it is the intended server. The `HTTPS` protocol in the Web uses a similar way to authenticate web servers, ensuring that you are talking to an intended web server, not a fake one. After this step, you should have a clear idea how the authentication in `HTTPS` works.

In this lab, `MiniVPN` should use such a method to authenticate the VPN server. You can implement an authentication protocol (such as SSL) from the scratch, using the crypto libraries in `OpenSSL` to verify certificates. Or you can use the `OpenSSL`'s SSL functions to directly make an SSL connection between the client and the server, in which case, the verification of certificates will be automatically carried out by the

SSL functions. Guidelines on making such a connection can be found in the next section.

Step 2: Authenticating VPN Client (i.e. User) There are two common ways to authenticate the user. One is using the public-key certificates. Namely, users need to get their own public-key certificates. When they try to create a VPN with the server, they need to send their certificates to the server, which will verify whether they have permissions for such a VPN. `OpenSSL`'s SSL functions also support this option if you specify that the client authentication is required.

Since users usually do not have their public-key certificates, a more common way to authenticate users is to use the traditional user name and password approach. Namely, after the client and the server have established a secure TCP connection between themselves, the server can ask the client to type the user name and the password, and the server then decide whether to allow the user to proceed depending on whether the user name and password matches with the information in the server's user database.

In this lab, you can pick either of them to implement.

Step 3: Key Exchange. If you use OpenSSL's SSL functions, after the authentication, a secure channel will be automatically established (by the OpenSSL functions). However, we are not going to use this TCP connection for our tunnel, because our VPN tunnel uses UDP. Therefore, we will treat this TCP connection as the control channel between the client and the server. Over this control channel, the client and the server will agree upon a session key for the data channel (i.e. the VPN tunnel). They can also use the control channel for other functionalities, such as updating the session key, exchanging the Initial Vector (IV), terminating the VPN tunnel, etc.

At the end of this step, you should be able to use the session key to secure the tunnel. In other words, you should be able to test Task 4 and Task 5 together.

Step 4: Dynamic Reconfiguration. You should implement some commands at the client side, to allow the client to do the following:

- Change the session key on the client end, and inform the server to make the similar change.
- Change the IV on the client end, and inform the server to make the similar change.
- Break the current VPN tunnel. The server needs to be informed, so it can release the correspondign resources.

You are encouraged to implement other features for your `MiniVPN`. Bonus points will be given to useful features. However, whatever features you add to your implementation, you need to ensure that the security is not compromised.

3.6 Task 6: Supporting Multiple VPN Tunnels

In the real world, one VPN server often supports multiple VPN tunnels. Namely, the VPN server allows more than one clients to connect to it simultaneously; each client has its own VPN tunnel with the server, and the session keys used in different tunnels should be different. Your VPN server should be able to support multiple clients. You cannot assume that there is only one tunnel and one session key.

When a packet arrives at the VPN server through a VPN tunnel, the server needs to figure out from which VPN tunnel the packet come from. Without this information, the server cannot know which decryption key (and IV) should be used to decrypt the packet; using a wrong key is going to cause the packet to be dropped, because the HMAC will not match. You can take a look at the IPSec protocol and think about how IPSec can support multiple tunnels. You can use a similar idea.

4 Guidelines

4.1 Port Forwarding

Recall that all our VMs are configured using the NAT option [1]. As we know, for such a configuration, the VMs are inaccessible by external computers. That is, external computers will not be able to connect to the VPN server that is running within the VM. To allow the VPN server to be accessible externally, we can use port forwarding to make certain port of the VM accessible to the outside.

Let us assume that the host machine's IP address is 128.230.10.10, the guest VM's IP address is 192.168.20.5, and the VPN server is running on the UDP port 4457 of the VM. If we forward the host machine's UDP port 4457 to the guest VM's UDP port 4457, all packets with the target 128.230.10.10:4457 will be forwarded to 192.168.20.5:4457. This way, external VPN clients only need to make its VPN connection to the port 4457 of the host machine; the packets will be forward to the VM.

Port forwarding can be easily configured in VirtualBox. Go to `Settings` of the VM image, select `Network`, choose `NAT` network adapter tab, expand `Advanced` option, and click `Port Forwarding`. You can then add a port forwarding rule through clicking on the plus button and filling the rule details.

4.2 Create a private network using VMs

We would like to create a private network for a company. The private network's prefix is `10.0.20.0/24` (see Figure 2). The network cannot be accessed from the outside. This provides a nice protection for this private network. The private network is connected to a gateway `10.0.20.1`, which connects to another networks via another network interface `192.168.20.5`. The VPN server will be installed on the gateway, which allows external computers to access the `10.0.20.0/24` private network. We will use VirtualBox to set up this private network. We need at least two VMs, one is the gateway, the other is a host in the private network. If your computer has enough memory, you can create more than one hosts in the private network, making it more realistic.

Gateway. For the gateway, we need two network interfaces. By default, a VM only has one network interface. We can go to the `Settings` of the VM, and add another network interface. For the first interface, we choose NAT. For the second interface, we need to activate it first through checking `Enabling Network Adapter`, then we specify its type through selecting `Attached` to `Internal Network` and assigning it a name.

Unless specifically configured, a computer will only act as a host, not as a gateway. In `Linux`, we need to enable the IP forwarding for a computer to behave like a gateway. IP forwarding can be enabled using the following command:

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

We also need to configure the routing table of the gateway, so it can behave correctly. Details of the configuration are left to students. You can use the command `route` to configure the routing table. Here is an example:

```
$ sudo route add -net 10.0.10.0 netmask 255.255.255.0 gw 10.0.20.1
```

Hosts in `10.0.20.0/24`. For these hosts, when configuring the network interface, we choose `Attached` to `Internal Network` and we assign it the same network name as the gateway.

[1] Although we can use the Bridge option, it does not work all the time for various reasons; for instance, a campus network might only assign an IP address to a computer with a registered MAC number.

4.3 Create certificates

In order to use OpenSSL to create certificates, you have to have a configuration file. The configuration file usually has an extension .cnf. It is used by three OpenSSL commands: ca, req and x509. The manual page of it can be found using Google Search. You can also get a copy of the configuration file from /usr/lib/ssl/openssl.cnf. After copying this file into your current directly, you need to create several sub-directory as specified in the configuration file (look at the [CA_default] section):

```
dir             = ./demoCA          # Where everything is kept
certs           = $dir/certs        # Where the issued certs are kept
crl_dir         = $dir/crl          # Where the issued crl are kept
new_certs_dir   = $dir/newcerts     # default place for new certs.

database        = $dir/index.txt    # database index file.
serial          = $dir/serial       # The current serial number
```

For the index.txt file, simply create an empty file. For the serial file, put a single number in string format (e.g. 1000) in the file. Once you have set up the configuration file openssl.cnf, you can create certificates for the three parties involved, the Certificate Authority (CA), the server, and the client.

Certificate Authority (CA). We will let you create your own CA, and then you can use this CA to issue certificates for servers and users. We will create a self-signed certificate for the CA. This means that this CA is totally trusted, and its certificate will serve as the root certificate. You can run the following command to generate the self-signed certificate for the CA:

```
$ openssl req -new -x509 -keyout ca.key -out ca.crt -config openssl.cnf
```

You will be prompted for information and a password. Do not lose this password, because you will have to type the passphrase each time you want to use this CA to sign another certificate. You will also be asked to fill in some information, such as the Country Name, Common Name, etc. The output of the command are stored in two files: ca.key and ca.crt. The file ca.key contains the CA's private key, while ca.crt contains the public-key certificate.

Server. Now we have our own trusted CA, we can now ask the CA to issue a public-key certificate for the server. First, we need to create a public/private key pair for the server. The server should run the following command to generate an RSA key pair (both private and public keys). You will also be required to provide a password to protect the keys. The keys will be stored in the file server.key:

```
$ openssl genrsa -des3 -out server.key 1024
```

Once you have the key file, you can generates a Certificate Signing Request (CSR). The CSR will be sent to the CA, who will generate a certificate for the key (usually after ensuring that identity information in the CSR matches with the server's true identity).

```
$ openssl req -new -key server.key -out server.csr -config openssl.cnf
```

Client. The client can follow the similar step to generate an RSA key pair and a certificate signing request:

```
$ openssl genrsa -des3 -out client.key 1024
$ openssl req -new -key client.key -out client.csr -config openssl.cnf
```

Generating Certificates. The CSR file needs to have the CA's signature to form a certificate. In the real world, the CSR files are usually sent to a trusted CA for their signature. In this lab, we will use our own trusted CA to generate certificates:

```
$ openssl ca -in server.csr -out server.crt -cert ca.crt -keyfile ca.key \
             -config openssl.cnf
$ openssl ca -in client.csr -out client.crt -cert ca.crt -keyfile ca.key \
             -config openssl.cnf
```

If `OpenSSL` refuses to generate certificates, it is very likely that the names in your requests do not match with those of CA. The matching rules are specified in the configuration file (look at the `[policy_match]` section). You can change the names of your requests to comply with the policy, or you can change the policy. The configuration file also includes another policy (called `policy_anything`), which is less restrictive. You can choose that policy by changing the following line:

```
"policy = policy_match"  change to "policy = policy_anything".
```

4.4 Create a secure TCP connection using OpenSSL

In this lab, students need to know how to use OpenSSL APIs to establish a secure TCP connection. There are many online tutorials on OpenSSL, so we will not give another one here. The followings are a few tutorials that are useful for this lab. These tutorials are also linked in the web page of this lab.

- OpenSSL examples: `http://www.rtfm.com/openssl-examples/`
- `http://www.ibm.com/developerworks/linux/library/l-openssl.html`

The first link above gives a client/server example. You can download the programs, and play with them. To make the downloaded programs work, you need to do the following:

- Untar the package.
- Run "`./configure`" to generated the Makefile.
- Open the generated `Makefile`, find the following line (about the 4th line):

  ```
  LD=-L/usr/local/ssl/lib  -lssl -lcrypto
  ```

  Add `-ldl` to the end of this line (`dl` means dynamic library). Without it, the compilation will fail. The line should now look like the following:

  ```
  LD=-L/usr/local/ssl/lib  -lssl -lcrypto -ldl
  ```

- Run "`make`", and then you should be able to get the programs compiled.
- When you run the example code, it should be noted that the certificates included in the example have already expired, so the authentication will fail. You need to replace the certificates with the ones you created.

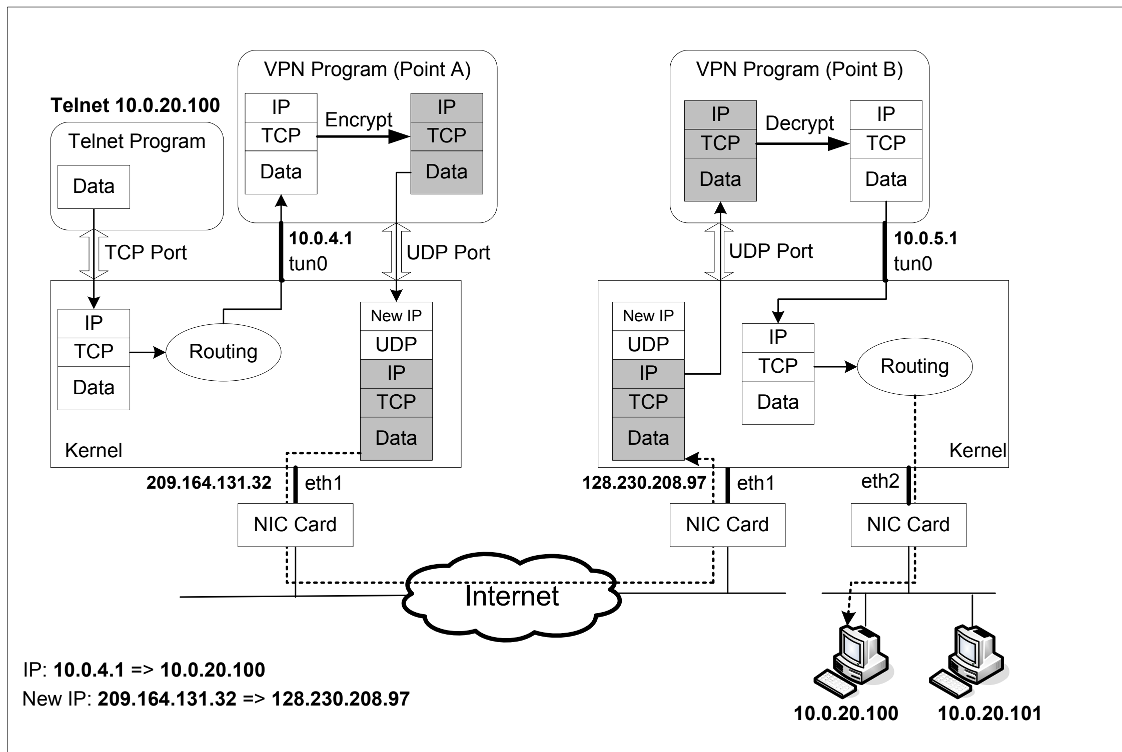
We also provide example codes, cli.cpp and serv.cpp in demo_openssl_api.tar.gz, to help you to understand how to use OpenSSL API to build secure TCP connections. It includes how to get peer's certificate, how to verify the certificate, how to check the private key for a certificate, etc.

4.5 An example: using `telnet` in our VPN

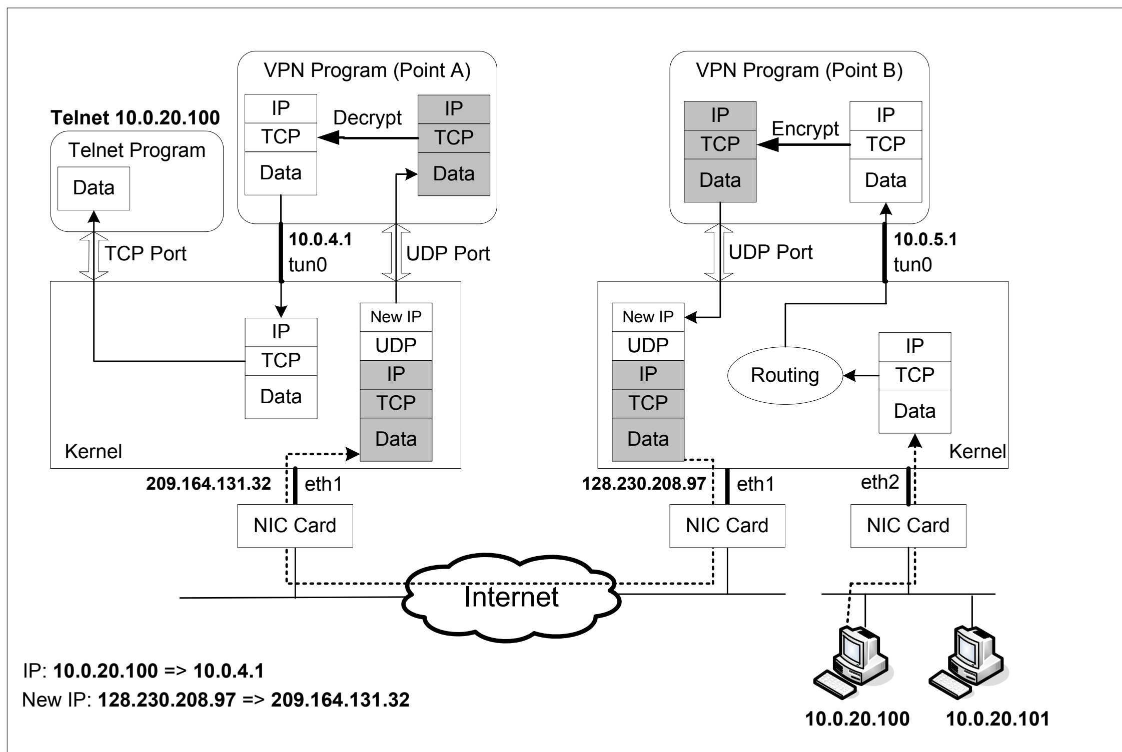
To help you fully understand how packets from an application flow to its destination through our `MiniVPN`, we have drawn two figures to illustrate the complete packet flow path when users run `telnet 10.0.20.100` from a host machine, which is the Point A of a host-to-gateway VPN. The other end of the VPN is on a gateway, which is connected to the `10.0.20.0/24` network, where our `telnet` server `10.0.20.100` resides.

Figure 3(a) shows how a packet flow from the `telnet` client to the server. Figure 3(b) shows how a packet flow from the `telnet` server back to the client. We will only describe the path in Figure 3(a) in the following. The return path is self-explained from Figure 3(b) once you have understood the path in Figure 3(a).

1. The data of the packet starts from the `telnet` program.
2. The kernel will construct an IP packet, with the destination IP address being `10.0.20.100`.
3. The kernel needs to decide which network interface the packet should be routed through: `eth1` or `tun0`. You need to set up your routing table correctly for the kernel to pick `tun0`. Once the decision is made, the kernel will set the source IP address of the packet using the IP address of the network interface, which is `10.0.4.1`.
4. The packet will reach our VPN program (Point A) through the virtual interface `tun0`, then it will be encrypted, and then be sent back to the kernel through a UDP port (not through the `tun0` interface). This is because our VPN program use the UDP as our tunnel.
5. The kernel will treat the encrypted IP packet as UDP data, construct a new IP packet, and put the entire encrypted IP packet as its UDP payload. The new IP's destination address will be the other end of the tunnel (decided by the VPN program we write); in the figure, the new IP's destination address is `128.230.208.97`.
6. You need to set up your routing table correctly, so the new packet will be routed through the interaface `eth1`; therefore, the source IP address of this new packet should be `209.164.131.32`.
7. The packet will now flow through the Internet, with the original `telnet` packet being entirely encrypted, and carried in the payload of the packet. This is why it is called a *tunnel*.
8. The packet will reach our gateway `128.230.208.97` through its interface `eth1`.
9. The kernel will give the UDP payload (i.e. the encrypted IP packet) to the VPN program (Point B), which is waiting for UDP data. This is through the UDP port.
10. The VPN program will decrypt the payload, and then feed the decrypted payload, which is the original `telnet` packet, back to the kernel through the virtual network interface `tun0`.
11. Since it comes through a network interface, the kernel will treat it as an IP packet (it is indeed an IP packet), look at its destination IP address, and decide where to route it. Remember, the destination IP address of this packet is `10.0.20.100`. If your routing table is set up correctly, the packet should be routed through `eth2`, because this is the interface that connects to the `10.0.20.0/24` network.
12. The `telnet` packet will now be delivered to its final destination `10.0.20.100`.



(a) An Example of packet flow from telnet client to server in Host-to-Gateway Tunnel



(b) An Example of packet flow from telnet server to client in Host-to-Gateway Tunnel

Figure 3: An Example of Packet Flow in VPN.

4.6 Miscenllanous notes

Our client (or server) program is going to listen to both TCP and UDP ports, these two activities may block each other. It is better if you can `fork()` two processes, one dealing with the TCP connection, and the other dealing with UDP. These processes need to be able to communicate with each other. You can use the Inter-process call (IPC) mechanisms for the communication. The simplest IPC mechanism is unnamed pipe, which should be sufficient for us. You can learn IPC from online documents.

5 Submission and Demonstration

You should submit a detailed lab report to describe your design and implementation. You should also describe how you test the functionalities and security of your system. You also need to demonstrate your system to us. Please sign up a demonstration time slot with the TA. Please take the following into consideration when you prepare for demonstraiton:

- The total time of the demo will be 15 minutes, no more additional time would be given. So prepare your demonstration so you can cover the important features.
- You are entirely responsible for showing the demo. We will NOT even touch the keyboard during the demonstration; so you should not depend on us to test your system. If you fail to demo some important features of your system, we will assume that your system does not have those features.
- You need to practice before you come to the demonstration. If the system crashes or anything goes wrong, it is your own fault. We will not debug your problems, nor give you extra time for it.
- During the demo, you should consider yourself as salesmen, and you want to sell your system to us. You are given 15 minutes to show us how good your system is. So think about your sales strategies. If you have implemented a great system, but fail to show us how good it is, you are not likely to get a good grade.
- Do turn off the messages your system prints out for debugging purposes. Those messages should not appear in a demonstration.