Virtual Private Network (VPN) Project

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 Transport Layer Security (TLS) / Secure Socket Layer (SSL). IPsec and TLS/SSL are two fundamentally different approaches for building VPNs. In this project, the primary focus is on the SSL-based VPNs where you will carry out your own SSL-VPN implementation. In the IPSec section of the project, you are expected to perform practical installation and configuration of an already existing VPN implementation for fulfilling given company requirements.

The learning objective of this project is for students to master the network and security technologies underlying SSL VPNs and have a fair insight on IPSec 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, you will implement a simple SSL VPN for Ubuntu.

2 Project Environment

You are going to implement this project on a pre-built virtual machine (VM). Please refer to Canvas for download instructions. VM instructions are prepared for Oracle VirtualBox, which you need to install on your computers so that you can import the VM. Make sure that the first thing you do after the import is to take a snapshot of the VM so that you can return back to this fresh state anytime you feel like you need to start all over again. The credentials of the VM are the following

```
Username: seed
Password: IK2206
```

We need to use OpenSSL package in this project. The package includes the header files, libraries, and commands. The package was not installed in the 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.

3 Project Tasks

In this project, you need to implement a simple VPN for Linux. We call it miniVPN. This chapter explains the tasks that you have to carry out in order to complete your own miniVPN implementation.
You will be dealing with tunnel establishment for various scenarios in the first three tasks. Four to six are software development tasks where you are expected to code your own VPN implementation that shall secure one of tunneling scenarios of your preference using TLS/SSL. Task seven involves no coding, you are expected to install and configure an already existing solution for IPSec VPN. Last task requires you to compare and contrast between IPSec and TLS/SSL based VPNs.

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 devices 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 network 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.

Following is an excellent tutorial on how to use TUN/TAP to create a tunnel between two machines, which is something that you have to master. `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. [1]

For your convenience we have modified that `simpletun` program and uploaded in Canvas. You can simply download this C program and run the following command to compile it. We will use `simpletun` to create tunnels in this project:

```
$ 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. Refer to section 4.1 of this document for VM settings.

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

[1] While compiling 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.

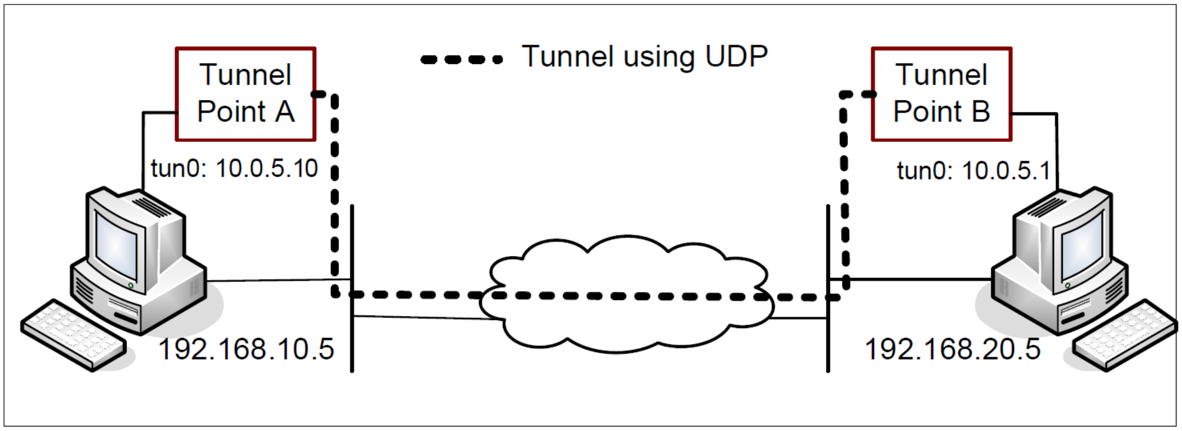

Figure 1: Host-to-Host Tunnel

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 VM now has multiple network interfaces, one is its own Ethernet 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 launch another terminal 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 10.0.5.10/24 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 launch another terminal for the following 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
```

Note that both ends of the tunnel are in the same subnet as if they were connected directly with a cable, i.e. pseudowire. This is the most common way of addressing tunnel interfaces. The only thing it can't do with current configuration, compared to a direct physical link, is passing L2 frames.

4. **Using the tunnel:** The tunnel should now be up. Now we can access 10.0.5.1 from 192.168.10.5 (and similarly access 10.0.5.10 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 10.0.5.10
$ ssh 10.0.5.10
```

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.

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

In the previous task you managed to establish the communication between two hosts through a tunnel. Imagine a scenario where a host needs to communicate with a network full of hosts at another location. It would be unfeasible to establish a tunnel for each host. Instead we place the network behind a VPN gateway and configure a tunnel between the source host and the gateway.

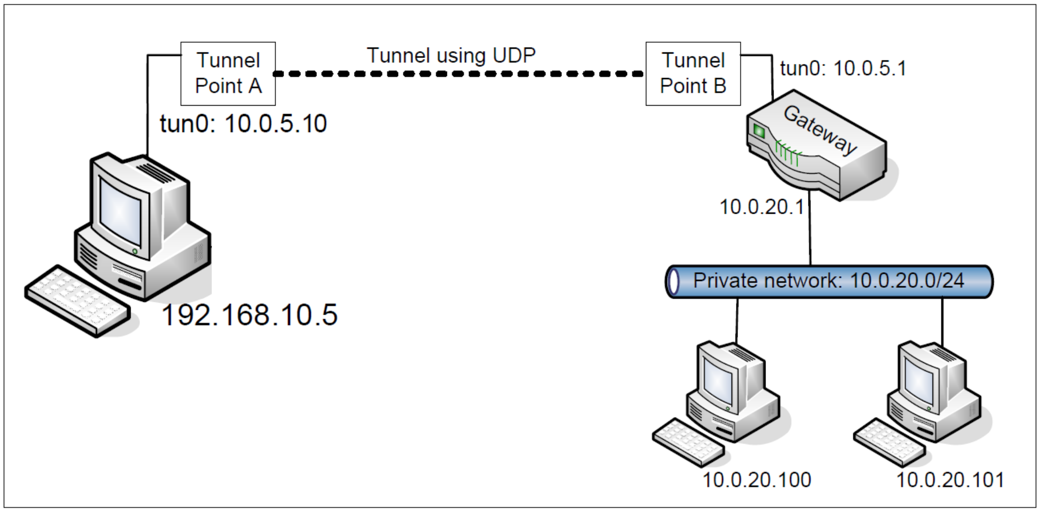

Figure 2: Host to Gateway

In this task, you need to create a tunnel between a VM and another VM that has gateway role, allowing the VM to access the private network that is behind the gateway, through the tunnel. 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. Refer to section 4.3 of this document for VM settings.

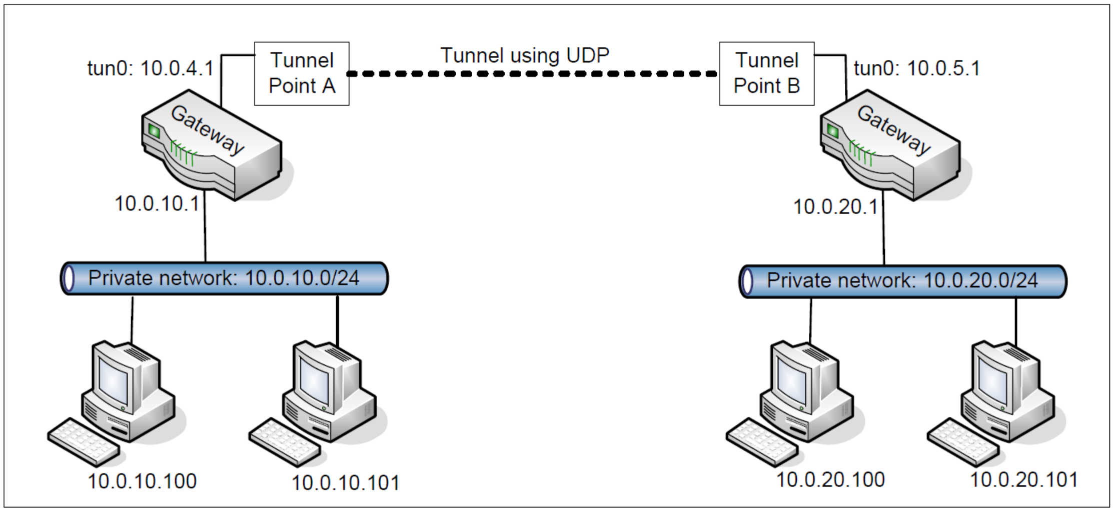

Figure 3: Gateway to Gateway

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 3.

3.4 Task 4: Create a secure 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 secure 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 project, 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 project, we only need to support the Message Authentication Code (MAC) method. The AES encryption algorithm and the HMAC-SHA256 algorithm 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. At the same time 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 project, `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 a TLS/SSL connection between the client and the server, in which case, the verification of certificates will be automatically carried out by the/TLS/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 the users 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 if the provided user name and password matches with the information in the server's user database.

In this project, 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 functions at the client side to allow the client to do the following:

- Change the session key on the client side, and inform the server to make the similar change.
- Change the IV on the client side, 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 corresponding resources.

You are encouraged to implement other features for your `MiniVPN` if time permits. However, whatever features you add to your implementation you need to ensure that security is not compromised.

3.6 Task 6: Supporting Multiple VPN Tunnels (Optional task)

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 came from. Without this information, the server cannot know which decryption key (and IV) should be used to decrypt the packet; using an incorrect key is going to cause the packet to be dropped, because the HMAC will not match. Investigate the IPSec protocol and think about how IPSec can support multiple tunnels. Perhaps you could use similar ideas for your solution.

3.7 Task 7: IPSec VPN Configuration

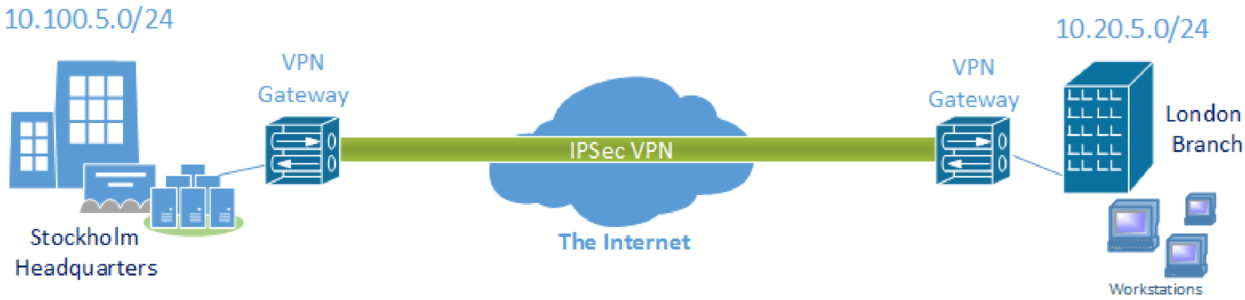

Figure 4: IPSec gateway to gateway VPN

ACME is a finance company of which the headquarters(HQ) is located in Stockholm. The computers of the recently established branch in London require access to the servers and services that reside at the data center in HQ in a secure fashion. You have been hired by ACME to accomplish this task. Following are the requirements outlined.

- Any layer 3 and above communication between HQ data center and London workstation subnets shall be secured with IPSec VPN.
- Your choice of cryptographic algorithms shall comply with the Strong Cryptography [2] definition of PCI-DSS.

For this task you install and configure StrongSwan; an Open Source IPsec-based VPN solution for Linux and other UNIX based operating systems, as the VPN gateways.

3.8 Task 8: IPSec vs. SSL VPN

In this task you are expected to investigate how IPSec and TLS/SSL VPN operates and handles the traffic, their differences and behavior under traffic load. Youu are expected to

[2] See Strong Cryptography in `https://www.pcisecuritystandards.org/pci_security/glossary`

- Perform packet captures for both of the scenarios, elaborate on packet and protocol structure (i.e. locate the header(s), payload, encapsulating protocol).
- Perform stress testing via generating bulk traffic using iperf. Investigate how CPU utilization and latency vary with traffic load.
- For SSL VPN only, extract the private key, then try to decrypt traffic from Wireshark [3].

Using your findings, discuss on pros and cons of both IPSec and SSL based VPNs.

4 Guidelines

4.1 Internetwork between VMs

Figure 1 represents a real-world situation where endpoints are geographically disparate and there exists an intermediary network (such as the Internet) that allows IPs of different subnets communicate. In order to fully represent that network environment we need a router in between the VMs and the easiest way is to configure your host machine as a router. So for task 3.1, create two VirtualBox host-only adapters (File → Preferences → Network → Host-only network) and assign the IPs 192.168.10.1/24 to one and 192.168.20.1/24 to the other. Now in the network settings of your host machine, you will see the two adapters created and assigned the IP addresses. Next step is to configure your host OS to enable IP forwarding so that it can route between its network adapters.

For Windows users: Go to Start and search for cmd. Right-click on ”Command Prompt” in the search result then select ”Run as administrator”. At the command prompt type regedit. Navigate to the `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters\IPEnableRouter` setting, right click and select Modify. Change 0 to 1 and exit the editor. Next, run services.msc in the command prompt. Go into the properties of *Routing and Remote Access* service and change the startup type to Automatic. Click apply and the Start button will become available. Start the service.

For Linux and Mac users: Open up a terminal and run the following commands.

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

Now that your host OS acts as a router between VirtualBox adapters, what is left is to change the network adapters of the VMs from NAT to Host-only and then assign one adapter per VM. In the VM to which you assigned the adapter with IP 192.168.10.1/24, assign the IP address 192.168.10.5 to eth0 then set 192.168.10.1 as the default gateway. Do the same for the other VM but for 192.168.20. By the end of your configuration two VMs should be able to ping each other.

4.2 NAT adapter and port forwarding

The provided VM comes with a NAT adapter by default [4]. 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.

[3] See `https://support.citrix.com/article/CTX116557`

[4] 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.

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.

4.3 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 commands.

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

We also need to configure the routing table of the gateway, so it can behave correctly. Details of the configuration are left to you. 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.

4.4 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 project, 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".
```

4.5 Create a secure TCP connection using OpenSSL

In this project, you 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 tutorial from IBM in `http://www.ibm.com/developerworks/linux/library/l-openssl.html` is very helpful for you to get going with OpenSSL programming.

In addition we provide a full blown tutorial with example code in *Introduction to OpenSSL programming* section in Canvas. To make the openssl-example program work, you need to do the following:

- Untar openssl-examples-20020110.tar.gz
- Run `"./configure"` to generated the Makefile.
- Open the generated `Makefile`, find the following line (about the 3rd 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 (directly under VPN Project module in Canvas) 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.

4.6 StrongSwan for IPSec VPN

StrongSwan is extremely well-documented that you would get going in no time. Installation is the following single line.

```
$ sudo apt-get install strongswan
```

Following is a scenario that is very similar to ours where working configurations are provided.
https://www.strongswan.org/testresults4.html

Follow section 4.3 for setting the VirtualBox environment. The only difference is the ports that you have to forward at the NAT interfaces of the gateways. For IPSec VPN to operate properly behind NAT, you have to forward UDP port 500 for IKE and UDP port 4500 for NAT-Traversal where ESP packets are encapsulated within UDP packets.

4.7 Miscellaneous 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 such as http://www.tldp.org/LDP/lpg/node7.html.

5 Submission and Demonstration

You are expected to perform a demonstration of your work. Start your demonstration with describing your design and implementation and also describe how you tested the functionality and security of your system. You should also submit your work in Canvas. Final deliverable for the submission consists of

- Your source code
- Your presentation slides.

archive them in a single ZIP file and submit in Canvas. Take the following into consideration when you prepare for demonstration:

- The total time of the demo will be 20 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. 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.
- Do turn off the messages your system prints out for debugging purposes. Those messages should not appear in a demonstration.