



SAN JOSE STATE UNIVERSITY

PROJECT REPORT

On

Virtual Private Network (VPN) Lab

Submitted to
Dr. Juzi Zhao
(Instructor – CMPE/EE 209)

May 10, 2018

Team Members:

Parvathi Chandrasekhar – 012474995

Varsha Kohirkar – 012031643

Sudha Ramaswamy Vijayakumar – 012486994

Group no: 12

Contents:

S. No.	Topics	Pg. No.
1.	Abstract	3
2.	Motivation and related work	5
3.	Concise statement of goals	6
4.	Description of model/ approach used	7
5.	Detailed description/ Implementation	8
6.	Conclusion	22
7.	Future scope	23
8.	Challenges faced during the project	23
9.	Appendix	24
10.	References	42

Abstract :

The objective of this project is to implement a VPN. Many recent technological advances have not only meant a shift toward industrial and retail globalization but also an increase in customer expectation and knowledge. All of this has led many companies to look to technology to provide an edge with which and on which to survive and excel. With facilities around the world and vendors, partners and staff spread even wider, the need for reliable and secure communications continues to increase.

According to Cole, Fossen, Northcutt and Pomeranz the definition of a VPN is “ a restricted use, logical computer network that is constructed from the system resources of a relatively public, physical network (such as the Internet), often by using encryption and often by tunneling links of the virtual network across the real network”. These virtual networks can be anything from a small remote office to a vendor or even a mobile user.

In recent years, various Virtual Private Network (VPN) technologies have been widely used to provide secure site-to-site connectivity and remote access. There are many reasons for such overwhelming adoption and business success; two major factors are total ownership cost savings and productivity enhancements. The total ownership cost can be considered as the initial deployment cost plus the cost of user training, support, and facility maintenance over time. Productivity enhancements can be measured in terms of tool effectiveness, user time savings, usability improvements, and user satisfaction.

A Virtual Private Network (VPN) is used to create 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. SSL VPN (Secure Sockets Layer virtual private network) is a form of VPN that can be used with a standard Web browser. In contrast to the traditional Internet Protocol Security (IPsec) VPN, SSL VPN does not require the installation of specialized client software on the end user's computer. It's used to give remote users with access to Web applications, client/server applications and internal network connections. 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 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, we have implemented a simple SSL VPN for Ubuntu. This includes creating a host-host tunnel, host-gateway tunnel as well as a gateway-gateway tunnel using TUN/TAP interfaces. The next step comprises of creating the VPN by encrypting the tunnel. Encryption includes confidentiality and integrity. Encryption is taken care of by AES Encryption Algorithm and confidentiality by HMAC-SHA256. Symmetric key algorithm will be used for both, same key used for both encryption and MAC. VPN server will be authenticated using public key certificates.

Motivation and related work:

Introduction:

Secure Sockets Layer (SSL) VPN is an emerging technology that provides remote-access VPN capability, using the SSL function that is already built into a modern web browser. SSL VPN allows users from any Internet-enabled location to launch a web browser to establish remote-access VPN connections, thus promising productivity enhancements and improved availability, as well as further IT cost reduction for VPN client software and support.

Additional VPN background information is widely available. Security mechanisms that can be used for risk mitigation must be taken into consideration.

Why SSL VPN is preferred?

SSL VPN has some unique features when compared with other existing VPN technologies. Most noticeably, SSL VPN uses SSL protocol and its successor, Transport Layer Security (TLS), to provide a secure connection between remote users and internal network resources. Today, this SSL/TLS function exists ubiquitously in modern web browsers.

While discussing the pros and cons of SSL VPN, we mainly compare it with the other predominant VPN technology, which is IPSEC (IP Security) VPN. IPSEC is an encryption protocol that works at the network layer. Unlike IPSEC VPN, SSL VPN doesn't require installation and configuration of client software at the user end. As a result, SSL VPN is also known as "clientless VPN" or "Web VPN".

Also, only internet browser to use SSL VPN is just needed. This in turn provides flexibility to use SSL VPN from any platform - Mac OS X, Windows, UNIX or any device like PC, Web-enabled phones, PDAs, etc. SSL VPN solutions provide granular access control for the application. One can define which user groups have what level of access on which all applications. SSL uses TCP port 443, which is normally already opened on the firewall. It also helps remote users when they are sitting behind other company's firewall. IPSEC uses specific UDP ports; If not in use, these ports are blocked by the firewall.

Another SSL VPN advantage over IPsec VPN is its ease of use for end users. Different IPsec VPN vendors may have different implementation and configuration requirements. SSL VPN, on the other hand, requires only a modern web browser. End users may even choose their favorite web browsers without being restricted by the operating system.

SSL VPN is advantageous for end users in the area of outbound connection security. In most environments, outbound Secure HTTP (HTTPS) traffic, which is also based on SSL, is not blocked. This means that even if a particular local environment does not permit outbound IPsec VPN sessions (such restriction is not unusual), SSL VPN is likely free of such restriction.

There is a difference between a full VPN tunnel and an SSL-enabled proxy server. The latter is an application gateway that supports a certain type of applications. A complete SSL VPN, on the other hand, is a VPN that provides all VPN characteristics and local LAN user experience (in terms of network access). If application access requirements are modest, SSL VPN does not require additional client software to be installed on the endpoint device. For broader application access, a dynamically downloadable tunneling client is typically delivered when needed to the client machine to support such full SSL VPN capabilities.

Concise statement of goals:

The learning objective of this lab is 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, we have implemented a simple SSL VPN.

In a nutshell, Virtual Private Network (VPN) is a technology that allows creating a private or secure network over the public network, such as Internet. This is achieved by establishing a secure tunnel between the user's machine and the enterprise network after authenticating the user. This enables the enterprises to provide access to the internal network for mobile employees without compromising on security. Secure VPN are one type of VPNs that use tunneling protocols with cryptographic capabilities. SSL is one such protocol that is used to provide confidentiality and authenticity while communicating over Internet. As the name indicates, SSL VPN uses SSL protocol to secure the VPN tunnels. Security awareness is an integral component of an organization's overall security effort.

SSL VPN user security awareness information on why VPN in general and SSL VPN including aspects of present security risks. Use at public terminals that do not meet standard security

policies and standards must be avoided and security precautions, such as terminating VPN sessions and clearing documents/information before leaving a public computer must be implemented. Examine the server certificate for URL's to guard against man-in-the-middle attacks. If SSL VPN is configured for non-web traffic, Viruses and Worms may infect the company's internal network from an insecure public terminal.

Description of model/ approach used :

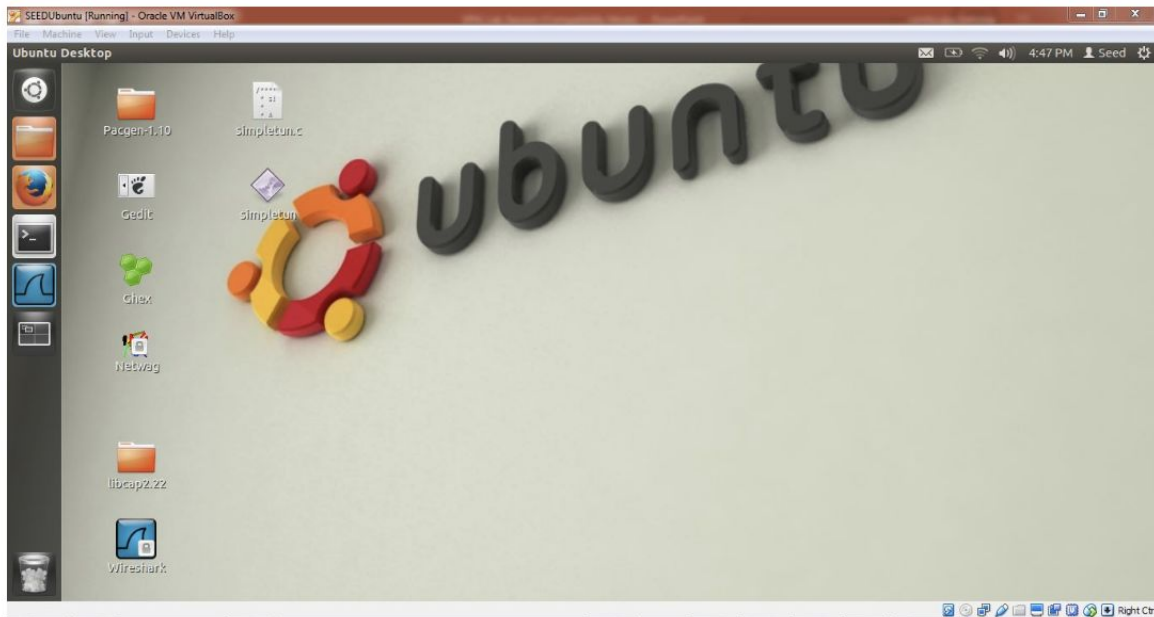
Here, we have incorporated several tasks in order to implement simple SSL VPN for Ubuntu. This includes creating a host-host tunnel, host-gateway tunnel as well as a gateway-gateway tunnel using TUN/TAP interfaces. The next step comprises of creating the VPN by encrypting the tunnel. Encryption includes confidentiality and integrity. Encryption is taken care of by AES Encryption Algorithm and confidentiality by HMAC-SHA256. Symmetric key algorithm will be used for both, same key used for both encryption and MAC. VPN server will be authenticated using public key certificates.

Introduction to OpenSSL package :

We need to use OpenSSL package in this lab. This package already includes the header files, libraries, and commands and can be easily installed using the following command

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

After downloading the source package, .tar.gz file is unpacked, and used the standard step ("*./config*", "*make*", "*make install*") to build and install the OpenSSL package.



The above figure shows Seed Ubuntu Installation with OpenSSL Library.

Detailed description/ Implementation:

Task 1: Host to Host Tunnel using Tun Interface.

TUN and TAP are virtual network kernel drivers which 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 whereas 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.

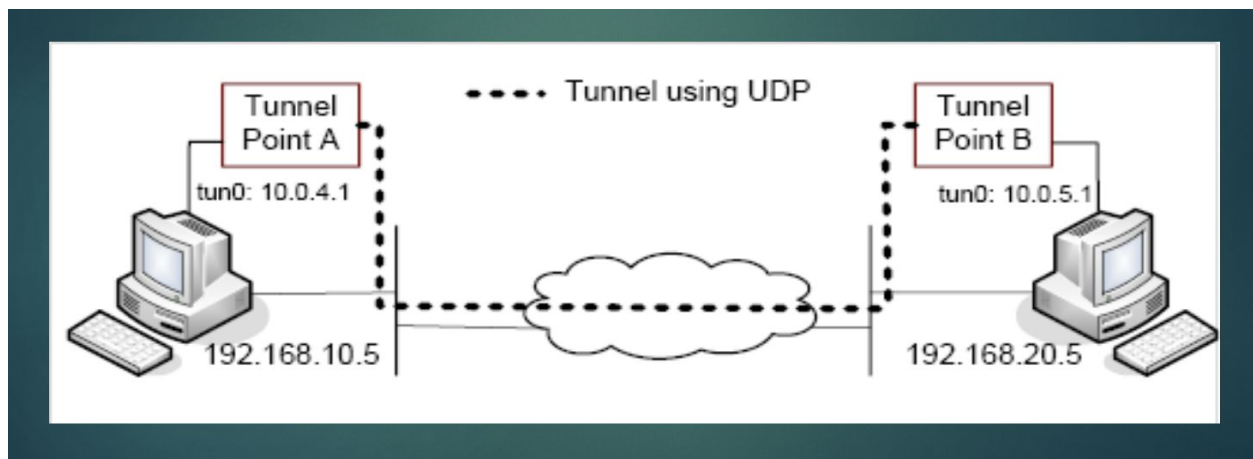
Packets sent by an operating system via a TUN/TAP network interface are delivered to the user-space program. On the other hand, packets sent by the program via a TUN/TAP network interface are injected into the operating system network stack; to the operating system, it appears that the packets come from 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 and the IP packets that the program sends to the interface will be piped into the computer through this virtual network interface. `read()` and `write()` system calls can be used to receive packets from or send packets to the virtual interface.

We have established Host to Host tunnel using Tun interface.

Tun interface is made active using command `openvpn` with the following command:

`-mktun -dev tun2`



The above figure shows the UDP connection is established between two Seed Ubuntu.

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.

a. 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.

b. Tunnel Point A:

We use Tunnel Point A as the server side of the tunnel. Point A is on machine 192.168.10.5. 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, we 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).

We need to find another window to configure the tun0 interface since the above command will block and wait for connections .

Here, 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
```

c.Tunnel Point B:

We have used Tunnel Point B as the client side of the tunnel. Point B is on machine 192.168.20.5. 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
```

d.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
```

e.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 using the following commands:

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
```

Gateway Setup:

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:

IP forwarding can be enabled using the following command:

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

Routing rules:

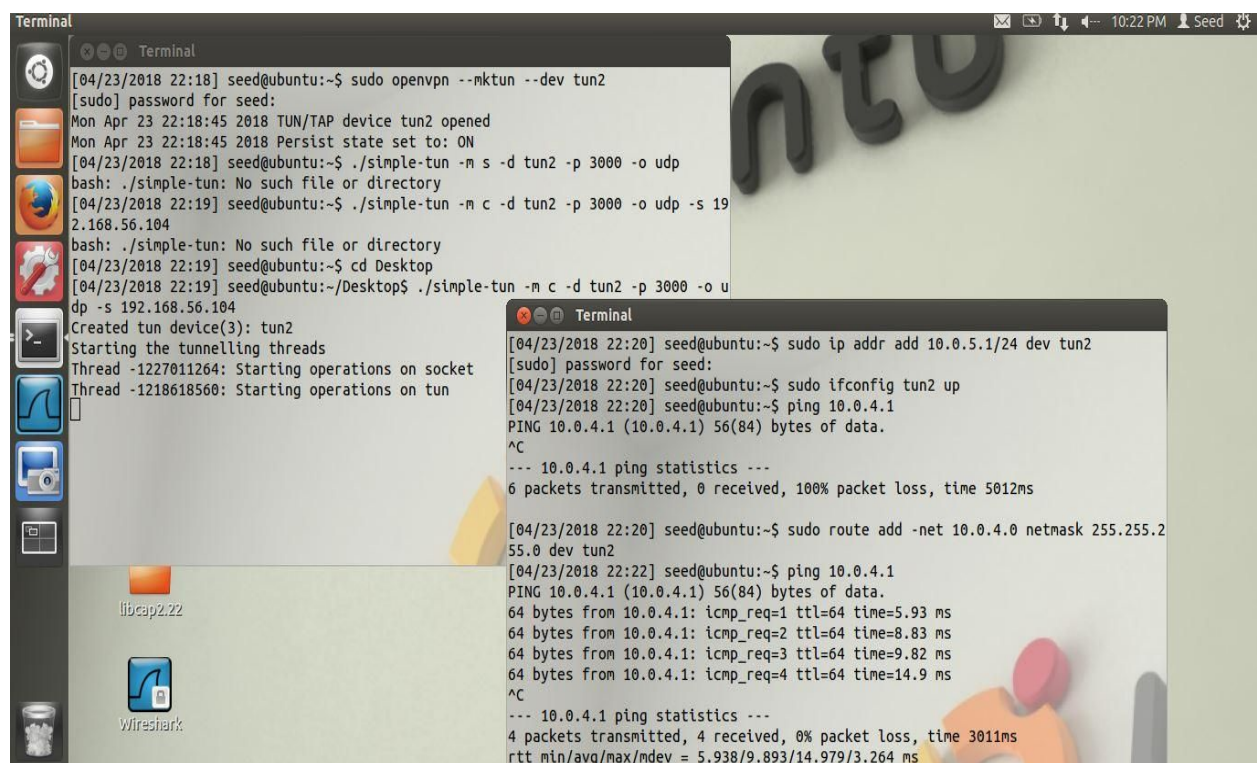
We also need to configure the routing table of the gateway, so it can behave correctly.

Example:

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

Host Setup:

Configure the routing table. 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.

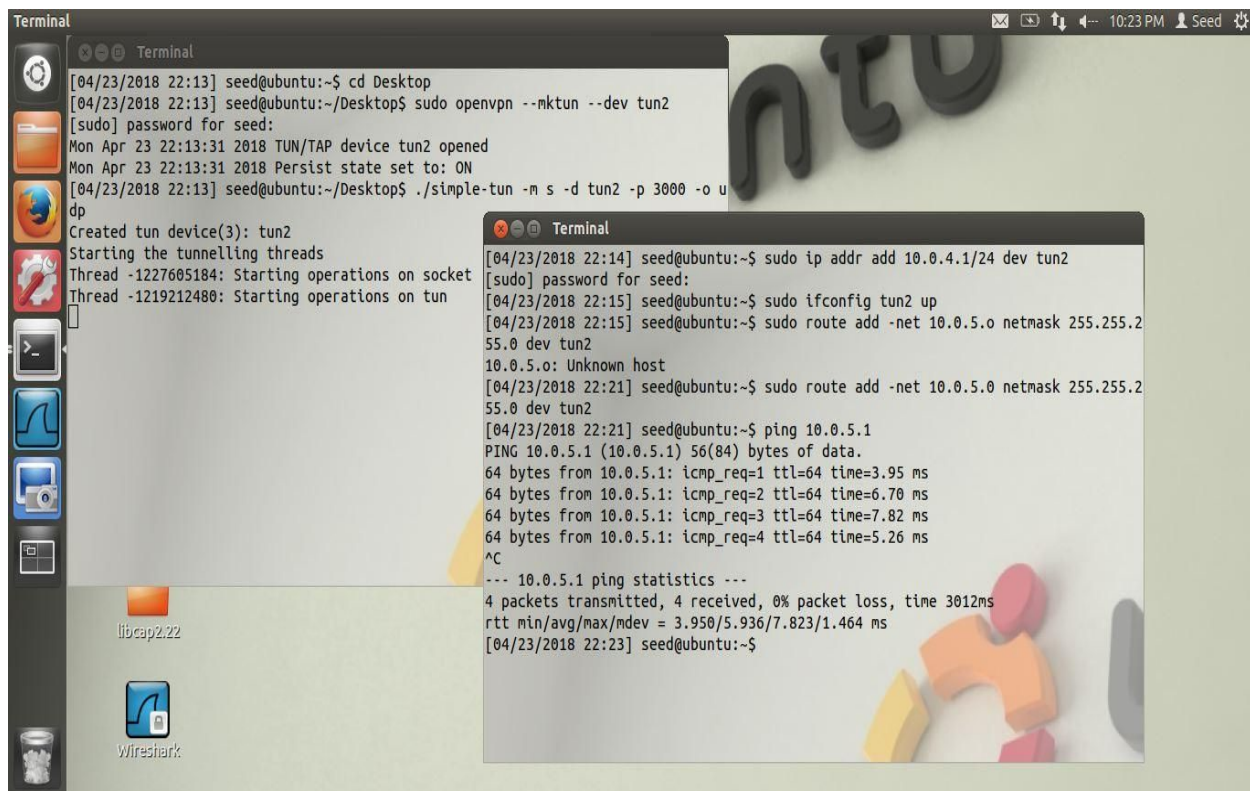


```
Terminal
[04/23/2018 22:18] seed@ubuntu:~$ sudo openvpn --mktun --dev tun2
[sudo] password for seed:
Mon Apr 23 22:18:45 2018 TUN/TAP device tun2 opened
Mon Apr 23 22:18:45 2018 Persist state set to: ON
[04/23/2018 22:18] seed@ubuntu:~$ ./simple-tun -m s -d tun2 -p 3000 -o udp
bash: ./simple-tun: No such file or directory
[04/23/2018 22:19] seed@ubuntu:~$ ./simple-tun -m c -d tun2 -p 3000 -o udp -s 192.168.56.104
bash: ./simple-tun: No such file or directory
[04/23/2018 22:19] seed@ubuntu:~$ cd Desktop
[04/23/2018 22:19] seed@ubuntu:~/Desktop$ ./simple-tun -m c -d tun2 -p 3000 -o udp -s 192.168.56.104
dp -s 192.168.56.104
Created tun device(3): tun2
Starting the tunnelling threads
Thread -1227011264: Starting operations on socket
Thread -1218618560: Starting operations on tun

Terminal
[04/23/2018 22:20] seed@ubuntu:~$ sudo ip addr add 10.0.5.1/24 dev tun2
[sudo] password for seed:
[04/23/2018 22:20] seed@ubuntu:~$ sudo ifconfig tun2 up
[04/23/2018 22:20] seed@ubuntu:~$ ping 10.0.4.1
PING 10.0.4.1 (10.0.4.1) 56(84) bytes of data.
^C
--- 10.0.4.1 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 5012ms

[04/23/2018 22:20] seed@ubuntu:~$ sudo route add -net 10.0.4.0 netmask 255.255.255.0 dev tun2
[04/23/2018 22:22] seed@ubuntu:~$ ping 10.0.4.1
PING 10.0.4.1 (10.0.4.1) 56(84) bytes of data.
64 bytes from 10.0.4.1: icmp_req=1 ttl=64 time=5.93 ms
64 bytes from 10.0.4.1: icmp_req=2 ttl=64 time=8.83 ms
64 bytes from 10.0.4.1: icmp_req=3 ttl=64 time=9.82 ms
64 bytes from 10.0.4.1: icmp_req=4 ttl=64 time=14.9 ms
^C
--- 10.0.4.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3011ms
rtt min/avg/max/mdev = 5.938/9.893/14.979/3.264 ms
```

The above figure shows Client contacting server.



```
[04/23/2018 22:13] seed@ubuntu:~$ cd Desktop
[04/23/2018 22:13] seed@ubuntu:~/Desktop$ sudo openvpn --mktun --dev tun2
[sudo] password for seed:
Mon Apr 23 22:13:31 2018 TUN/TAP device tun2 opened
Mon Apr 23 22:13:31 2018 Persist state set to: ON
[04/23/2018 22:13] seed@ubuntu:~/Desktop$ ./simple-tun -m s -d tun2 -p 3000 -o u
dp
Created tun device(3): tun2
Starting the tunnelling threads
Thread -1227605184: Starting operations on socket
Thread -1219212480: Starting operations on tun
[04/23/2018 22:14] seed@ubuntu:~$ sudo ip addr add 10.0.4.1/24 dev tun2
[sudo] password for seed:
[04/23/2018 22:15] seed@ubuntu:~$ sudo ifconfig tun2 up
[04/23/2018 22:15] seed@ubuntu:~$ sudo route add -net 10.0.5.0 netmask 255.255.2
55.0 dev tun2
10.0.5.0: Unknown host
[04/23/2018 22:21] seed@ubuntu:~$ sudo route add -net 10.0.5.0 netmask 255.255.2
55.0 dev tun2
[04/23/2018 22:21] seed@ubuntu:~$ ping 10.0.5.1
PING 10.0.5.1 (10.0.5.1) 56(84) bytes of data.
64 bytes from 10.0.5.1: icmp_req=1 ttl=64 time=3.95 ms
64 bytes from 10.0.5.1: icmp_req=2 ttl=64 time=6.70 ms
64 bytes from 10.0.5.1: icmp_req=3 ttl=64 time=7.82 ms
64 bytes from 10.0.5.1: icmp_req=4 ttl=64 time=5.26 ms
^C
--- 10.0.5.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3012ms
rtt min/avg/max/mdev = 3.950/5.936/7.823/1.464 ms
[04/23/2018 22:23] seed@ubuntu:~$
```

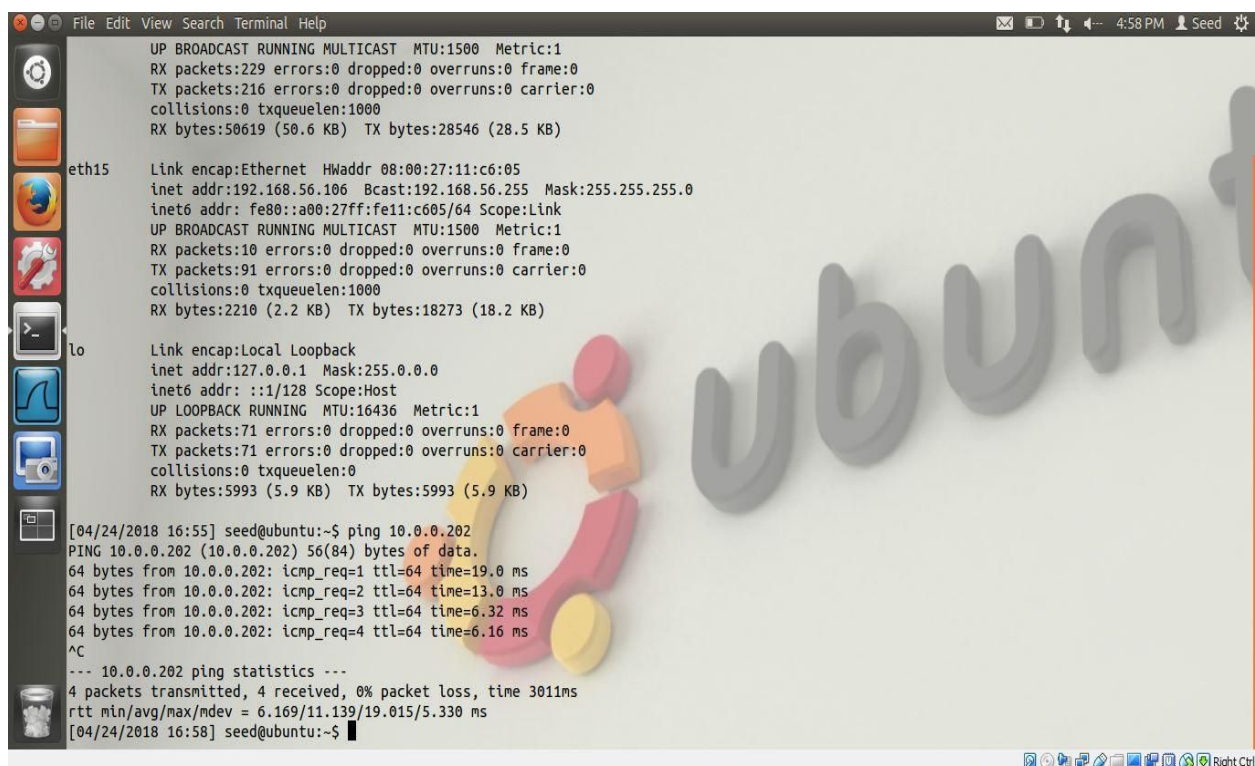
The above figure shows Server contacting client.

Use of UDP in the Tunnel:

- If the VPN uses TCP, then TCP connections will use IP packets sent through the VPN, so we end up paying the TCP overhead twice. Also, TCP is heavyweight and requires three packets to set up a connection before the actual data are sent.
- However, an UDP-based VPN has the potential for slightly better performance because of lower overhead especially in case of low bandwidth and it is lightweight, fast and efficient. There is no acknowledgement of handshake. It does error-checking, fast and streaming RDP connections can be made but there is no recovery option.

Task 2: Create a Host-to-Gateway Tunnel

Once the tunnel is set up on the two VMs within a single host machine, we should set up a similar tunnel on two VMs on two different host machines. Port forwarding technique can be used for this purpose to access certain port of VM to outside. In this task, a tunnel between a computer and a gateway, allowing the computer to access the private network that is connected to the gateway must be created and we need two physical computers to demonstrate this. On one computer, several VMs within the computer to set up the gateway and the private network can run while in the other computer, use a VM to communicate to the hosts on the private network.



```
UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
RX packets:229 errors:0 dropped:0 overruns:0 frame:0
TX packets:216 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:50619 (50.6 KB)  TX bytes:28546 (28.5 KB)

eth15  Link encap:Ethernet  HWaddr 08:00:27:11:c6:05
        inet addr:192.168.56.106  Bcast:192.168.56.255  Mask:255.255.255.0
        inet6 addr: fe80::a00:27ff:fe11:c605/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:10 errors:0 dropped:0 overruns:0 frame:0
        TX packets:91 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:2210 (2.2 KB)  TX bytes:18273 (18.2 KB)

lo      Link encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING  MTU:16436  Metric:1
        RX packets:71 errors:0 dropped:0 overruns:0 frame:0
        TX packets:71 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:5993 (5.9 KB)  TX bytes:5993 (5.9 KB)

[04/24/2018 16:55] seed@ubuntu:~$ ping 10.0.0.202
PING 10.0.0.202 (10.0.0.202) 56(84) bytes of data:
64 bytes from 10.0.0.202: icmp_req=1 ttl=64 time=19.0 ms
64 bytes from 10.0.0.202: icmp_req=2 ttl=64 time=13.0 ms
64 bytes from 10.0.0.202: icmp_req=3 ttl=64 time=6.32 ms
64 bytes from 10.0.0.202: icmp_req=4 ttl=64 time=6.16 ms
^C
--- 10.0.0.202 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3011ms
rtt min/avg/max/mdev = 6.169/11.139/19.015/5.330 ms
[04/24/2018 16:58] seed@ubuntu:~$
```

The above figure shows Host to Host gateway tunnel setup.


```
Terminal
eth13 Link encap:Ethernet HWaddr 08:00:27:eb:2e:22
      inet addr:10.0.0.202 Bcast:10.0.0.255 Mask:255.255.255.0
      inet6 addr: 2601:646:8200:f49f::2850/64 Scope:Global
      inet6 addr: 2601:646:8200:f49f:a00:27ff:feeb:2e22/64 Scope:Global
      inet6 addr: fe80::a00:27ff:feeb:2e22/64 Scope:Link
      inet6 addr: 2601:646:8200:f49f:489c:e072:1e4:8b56/64 Scope:Global
      UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
      RX packets:1205 errors:0 dropped:0 overruns:0 frame:0
      TX packets:269 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:158057 (158.0 KB) TX bytes:49729 (49.7 KB)

eth14 Link encap:Ethernet HWaddr 08:00:27:b2:af:10
      inet addr:192.168.56.104 Bcast:192.168.56.255 Mask:255.255.255.0
      inet6 addr: fe80::a00:27ff:feb2:af10/64 Scope:Link
      UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
      RX packets:25 errors:0 dropped:0 overruns:0 frame:0
      TX packets:86 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:7455 (7.4 KB) TX bytes:17042 (17.0 KB)

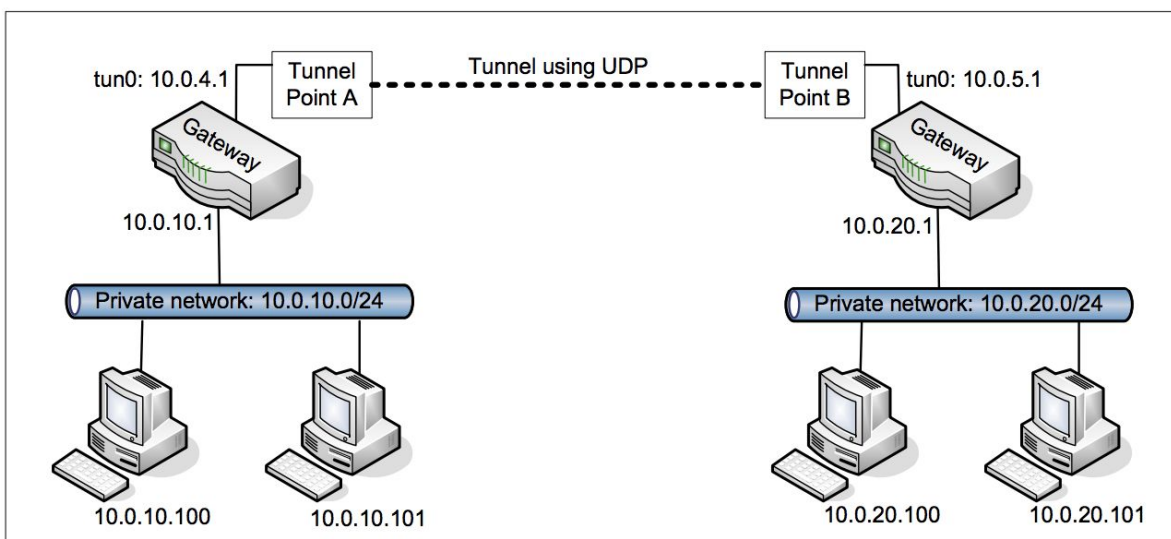
lo Link encap:Local Loopback
   inet addr:127.0.0.1 Mask:255.0.0.0
   inet6 addr: ::1/128 Scope:Host
   UP LOOPBACK RUNNING MTU:16436 Metric:1
   RX packets:132 errors:0 dropped:0 overruns:0 frame:0
   TX packets:132 errors:0 dropped:0 overruns:0 carrier:0
   collisions:0 txqueuelen:0
   RX bytes:12367 (12.3 KB) TX bytes:12367 (12.3 KB)

[04/24/2018 16:53] seed@ubuntu:~$ sudo sysctl net.ipv4.ip_forward=1
[sudo] password for seed:
net.ipv4.ip_forward = 1
[04/24/2018 16:56] seed@ubuntu:~$ sudo route add -net 10.0.0.0 netmask 255.255.255.0 gw 10.0.0.202
[04/24/2018 16:58] seed@ubuntu:~$
```

The above figure shows Gateway Ip address.

Task 3: Create a Gateway-to-Gateway Tunnel

We need 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 shown in the below figure.




```
Terminal
RX bytes:158057 (158.0 KB) TX bytes:49729 (49.7 KB)

eth14  Link encap:Ethernet  HWaddr 08:00:27:b2:af:10
        inet addr:192.168.56.104 Bcast:192.168.56.255 Mask:255.255.255.0
        inet6 addr: fe80::a00:27ff:feb2:af10/64 Scope:Link
        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
        RX packets:25 errors:0 dropped:0 overruns:0 frame:0
        TX packets:86 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:7455 (7.4 KB) TX bytes:17042 (17.0 KB)

lo      Link encap:Local Loopback
        inet addr:127.0.0.1 Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING  MTU:16384  Metric:1
        RX packets:132 errors:0 dropped:0 overruns:0 frame:0
        TX packets:132 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:12367 (12.3 KB) TX bytes:12367 (12.3 KB)

[04/24/2018 16:53] seed@ubuntu:~$ sudo sysctl net.ipv4.ip_forward=1
[sudo] password for seed:
net.ipv4.ip_forward = 1
[04/24/2018 16:56] seed@ubuntu:~$ sudo route add -net 10.0.0.0 netmask 255.255.255.0 gw 10.0.0.202
[04/24/2018 16:58] seed@ubuntu:~$ ping 10.0.0.47
PING 10.0.0.47 (10.0.0.47) 56(84) bytes of data.
 64 bytes from 10.0.0.47: icmp_req=1 ttl=64 time=23.6 ms
 64 bytes from 10.0.0.47: icmp_req=2 ttl=64 time=9.93 ms
 64 bytes from 10.0.0.47: icmp_req=3 ttl=64 time=8.85 ms
 64 bytes from 10.0.0.47: icmp_req=4 ttl=64 time=8.48 ms
^C
--- 10.0.0.47 ping statistics ---
 4 packets transmitted, 4 received, 0% packet loss, time 3009ms
 rtt min/avg/max/mdev = 8.488/12.719/23.602/6.306 ms
[04/24/2018 17:26] seed@ubuntu:~$
```

The above figure shows establishment of Gateway to gateway tunnel.

Task 4: Create a Virtual Private Network (VPN)

To get a VPN, we need to secure the tunnel which we created between two machines.

To secure the network 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. We will establish the TCP connection between client and server and encrypt the data passing through with AES encryption. We have used 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 are both implemented in the OpenSSL library. There are plenty of online documents explaining how to use the Open SSL'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). The IV should be randomly generated for each VPN tunnel.

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.

Create Certificate :

In order to use OpenSSL to create certificates, we need 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 configuration file can be obtained from */usr/lib/ssl/openssl.cnf*.

After copying this file into the current directory, you need to create several sub-directories 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 the configuration file *openssl.cnf* is set up, one can create certificates for the three parties involved, the Certificate Authority (CA), the server, and the client.

Certificate Authority (CA):

We can use CA to issue certificates for servers and users. We have created a self-signed certificate for the CA. This means that this CA is totally trusted, and its certificate will serve as the root certificate.

The following command to generate the self-signed certificate for the CA can be used :

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

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. Then, CA can be made to issue a public-key certificate for the server.

Server:

A public/private key pair for the server must be created. The server should run the following command to generate an RSA key pair (both private and public keys).

The keys will be stored in the file *server.key*:

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

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

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

Client:

The similar steps can be followed as above 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:

First of all, Self signed certificate is generated for Certificate Authority(CA) at both client and server This command will generate CA private key and CA public key certificate. At server side,private-public key pair is generated using RSA algorithm so that CA can issue server a trusted public-key certificate. For this Certificate Signing Request is generated and sent to CA from server to issue the certificate. Same steps are replicated at client side to generate a certificate.

The CSR file needs to have the CA's signature to form a certificate.

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
```

Creating secure connection using OpenSSL:

Preparation work:

Untar the package. apt-get source openssl . Run "./configure" to generated the Makefile.

Directory of headers and libraries:

```
/usr/local/ssl/include
/usr/local/ssl/lib
```

Open the generated Makefile. Add dynamic library for the compilation purpose.

LD=-L/usr/local/ssl/lib -lssl -lcrypto. Add -ldl to the end of this line.

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. If the certificates have already expired, then the authentication will fail and in that case we need to replace the certificates with new one.

OpenSSL is used to 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.

Secure connections require a handshake after the connection is established. the server sends a certificate to the client the client then verifies against a set of trust certificates It also checks the certificate to make sure that it has not expired a trust certificate store be loaded prior to establishing the connection The client will send a certificate to the server only if the server requests one.

```
Terminal
[04/24/2018 19:00] seed@ubuntu:~/Desktop/dir$ sudo openssl ca -in client.csr -out client.crt -cert ca.crt -keyfile ca.key -config openssl.cnf
Using configuration from openssl.cnf
Enter pass phrase for ca.key:
Check that the request matches the signature
Signature ok
Certificate Details:
  Serial Number: 286 (0x11e)
  Validity
    Not Before: Apr 25 02:00:57 2018 GMT
    Not After : Apr 25 02:00:57 2019 GMT
  Subject:
    countryName           = US
    stateOrProvinceName   = CA
    organizationName      = SJSU
    organizationalUnitName = 1
    commonName            = Par
    emailAddress          = parvathi.chandrasekhar@sjsu.edu
  X509v3 extensions:
    X509v3 Basic Constraints:
      CA:FALSE
    Netscape Comment:
      OpenSSL Generated Certificate
    X509v3 Subject Key Identifier:
      78:D0:A9:70:29:C7:92:09:C6:33:11:7D:AC:87:9B:BF:F5:98:CD:53
    X509v3 Authority Key Identifier:
      keyId:C7:00:41:3E:D0:C1:A6:AA:39:FC:EA:C3:CA:39:AF:BE:A7:42:69:E4

Certificate is to be certified until Apr 25 02:00:57 2019 GMT (365 days)
Sign the certificate? [y/n]:y

1 out of 1 certificate requests certified, commit? [y/n]:y
Write out database with 1 new entries
Data Base Updated
[04/24/2018 19:01] seed@ubuntu:~/Desktop/dir$
```

The above figure shows Certificate generation.

```
[04/24/2018 19:01] seed@ubuntu:~$ openssl ca -in client.csr -out client.crt -cert ca.crt -keyfile ca.key -config openssl.cnf
Using configuration from openssl.cnf
Enter pass phrase for ca.key:
Check that the request matches the signature
Signature ok
Certificate Details:
  Serial Number: 286 (0x11e)
  Validity
    Not Before: Apr 25 02:01:57 2018 GMT
    Not After : Apr 25 02:01:57 2019 GMT
  Subject:
    countryName           = US
    stateOrProvinceName   = CA
    organizationName      = SJSU
    organizationalUnitName = 1
    commonName            = varsha
    emailAddress          = varsha.kohirkar@gmail.com
  X509v3 extensions:
    X509v3 Basic Constraints:
      CA:FALSE
    Netscape Comment:
      OpenSSL Generated Certificate
    X509v3 Subject Key Identifier:
      08:E9:39:EF:43:59:92:C9:AD:8A:BD:CA:B2:5F:E4:73:E1:7A:EF:F0
    X509v3 Authority Key Identifier:
      keyId:8F:4F:E9:91:BD:8F:2B:8A:A3:0B:79:9D:1A:59:00:BE:AC:42:BC:47

Certificate is to be certified until Apr 25 02:01:57 2019 GMT (365 days)
Sign the certificate? [y/n]:y

1 out of 1 certificate requests certified, commit? [y/n]:y
Write out database with 1 new entries
Data Base Updated
[04/24/2018 19:02] seed@ubuntu:~$
```

The above figure shows Client certificate generation.

```
Terminal
[04/24/2018 18:48] seed@ubuntu:~/Desktop/dir$ sudo openssl ca -in server.csr -out server.crt -cert ca.crt -keyfile ca.key -config openssl.cnf
Using configuration from openssl.cnf
Enter pass phrase for ca.key:
Check that the request matches the signature
Signature ok
Certificate Details:
  Serial Number: 286 (0x11e)
  Validity
    Not Before: Apr 25 01:54:42 2018 GMT
    Not After : Apr 25 01:54:42 2019 GMT
  Subject:
    countryName           = US
    stateOrProvinceName   = CA
    organizationName       = SJSU
    organizationalUnitName = 1
    commonName             = Par
    emailAddress           = parvathi.chandrasekhar@sjsu.edu
  X509v3 extensions:
    X509v3 Basic Constraints:
      CA:FALSE
    Netscape Comment:
      OpenSSL Generated Certificate
    X509v3 Subject Key Identifier:
      DC:66:F1:12:23:36:F9:60:FE:B4:74:97:EA:E9:08:93:29:99:51:DC
    X509v3 Authority Key Identifier:
      keyId:1B:BC:25:FB:C8:1E:AD:B5:9B:80:48:86:CE:6A:52:97:39:13:DA:34

Certificate is to be certified until Apr 25 01:54:42 2019 GMT (365 days)
Sign the certificate? [y/n]:y

1 out of 1 certificate requests certified, commit? [y/n]:y
Write out database with 1 new entries
Data Base Updated
[04/24/2018 18:54] seed@ubuntu:~/Desktop/dir$ cd --
```

The above figure shows Server certificate generation.

```
[04/30/2018 11:01] seed@ubuntu:~/Downloads/demo_openssl_api/demo_openssl_api/demo_openssl_api$ gcc -o clientt clientt.cpp -lssl -lcrypto
[04/30/2018 11:02] seed@ubuntu:~/Downloads/demo_openssl_api/demo_openssl_api/demo_openssl_api$ ./clientt
initial setup done

verified cacert location
Enter PEM pass phrase:
successfully connected to server
SSL connection using AES256-GCM-SHA384
Server certificate:
  subject: /C=US/ST=CA/O=SJSU/OU=1/CN=varsha/emailAddress=varsha.kohirkar@gmail.com
  issuer: /C=US/ST=CA/L=SJ/O=SJSU/OU=1/CN=varsha/emailAddress=varsha.kohirkar@gmail.com
Got 11 chars: 'I hear you.'
```

The above figure shows SSL connection using AES and SHA384,Server certificate verified.

Conclusion:

We have successfully implemented a simple SSL VPN and we have described the approach followed in detail. As already stated above, SSL VPNs were born from the user community's want for VPN access from anywhere at any time. SSL VPNs use the industry accepted standards of encryption and key exchange such as 3DES, MD5 and SHA. SSL VPNs provide access to web based applications and not the internal network. The functionality that SSL remote access VPNs cannot be outweighed and vendors are confident that they too will stand the test of time. As the number of legacy applications continues to decline and the number of web-based applications continues to increase SSL will slowly become the primary solution for VPN access. However, the layer 3 encryption methodology of IPsec will always be considered

the more secure and versatile. Many vendors such as Cisco, Checkpoint and Nortel are beginning to offer a solution that utilizes a combination of the two. They are suggesting that the best solution offers an IPSec VPN to those users that require low-level access and from those computers over which the company has control. They are then using an SSL VPN solution to provide simple email and extranet access to staff, customers and vendors that require only web-based access. In this way costs are kept low as support calls remain low and access is granted to all users how and when they want.

Future scope:

Supporting Multiple VPN Tunnels:

Future scope is to make VPN server, allow 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. The server needs to figure out from which VPN tunnel the incoming packet comes from, to ensure proper decryption.

Challenges faced during the project:

- Third party searching tools (like Google desktop) cache the web pages served by SSL VPN gateway. This poses a serious security concern while using SSL VPN, which erases temporary data after the completion of each SSL VPN session.
- The primary issue with SSL VPNs is that it only really provides VPN access to web-enabled applications.
- Another issue is that multiple key exchanges may be required during one session. This slows performance of the web server due to the load of performing constant SSL.
- The fact that SSL VPN provides browser-based access to applications means that Internet access is always required. If a mobile user does not have access to the Internet then he cannot work offline. Unlike IPSec where the applications were loaded locally SSL VPN applications are not local. This may mean a lack of productivity in the event of no access. Despite all of the issues, SSL remote access VPNs ultimately provide what the end users crave and that is access from anywhere. It is for this benefit alone that many believe that it will become the dominant technology.

Appendix:

//SERVER CODE//

```
/* serv.cpp - Minimal ssleay server for Unix
   30.9.1996, Sampo Kellomaki <sampo@iki.fi> */

/* mangled to work with SSLeay-0.9.0b and OpenSSL 0.9.2b
   Simplified to be even more minimal
   12/98 - 4/99 Wade Scholine <wades@mail.cybg.com> */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <memory.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <openssl/rsa.h>          /* SSLeay stuff */
#include <openssl/crypto.h>
#include <openssl/x509.h>
#include <openssl/pem.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

/* define HOME to be dir for key and cert files... */
#define HOME "."
/* Make these what you want for cert & key files */
#define CERTF HOME "server.crt"
#define KEYF HOME "server.key"
#define CACERT HOME "ca.crt"

#define CHK_NULL(x) if ((x)==NULL) exit (1)
#define CHK_ERR(err,s) if ((err)==-1) { perror(s); exit(1); }
#define CHK_SSL(err) if ((err)==-1) { ERR_print_errors_fp(stderr);
exit(2); }
```



```

int main ()
{
    int err;
    int listen_sd;
    int sd;
    struct sockaddr_in sa_serv;
    struct sockaddr_in sa_cli;
    size_t client_len;
    SSL_CTX* ctx;
    SSL*      ssl;
    X509*      client_cert;
    char*      str;
    char      buf [4096];
    const SSL_METHOD *meth;

    /* SSL preliminaries. We keep the certificate and key with the context.
    */

    SSL_load_error_strings();
    SSL_load_error_strings();
    meth = SSLv23_server_method();
    ctx = SSL_CTX_new (meth);
    if (!ctx) {
        ERR_print_errors_fp(stderr);
        exit(2);
    }

    //SSL_CTX_set_verify(ctx,SSL_VERIFY_PEER,NULL); /* whether verify the
certificate */
    SSL_CTX_load_verify_locations(ctx,CACERT,NULL);
    SSL_CTX_set_client_CA_list(ctx, SSL_load_client_CA_file(CACERT));
    if (SSL_CTX_use_certificate_file(ctx, CERTF, SSL_FILETYPE_PEM) <= 0) {
        ERR_print_errors_fp(stderr);
        exit(3);
    }
    if (SSL_CTX_use_PrivateKey_file(ctx, KEYF, SSL_FILETYPE_PEM) <= 0) {
        ERR_print_errors_fp(stderr);
        exit(4);
    }

    if (!SSL_CTX_check_private_key(ctx)) {
        fprintf(stderr,"Private key does not match the certificate public
key\n");
        exit(5);
    }

```

```

}

/* ----- */
/* Prepare TCP socket for receiving connections */

listen_sd = socket (AF_INET, SOCK_STREAM, 0);   CHK_ERR(listen_sd,
"socket");

memset (&sa_serv, '\0', sizeof(sa_serv));
sa_serv.sin_family      = AF_INET;
sa_serv.sin_addr.s_addr = INADDR_ANY;
sa_serv.sin_port        = htons (8888);        /* Server Port number
*/

err = bind(listen_sd, (struct sockaddr*) &sa_serv,
            sizeof (sa_serv));                  CHK_ERR(err, "bind");

/* Receive a TCP connection. */

err = listen (listen_sd, 5);                    CHK_ERR(err, "listen");
printf("started listening");
client_len = sizeof(sa_cli);
sd = accept (listen_sd, (struct sockaddr*) &sa_cli, &client_len);
CHK_ERR(sd, "accept");
//close (listen_sd);

printf ("\nConnection from %lx, port %x\n",
        sa_cli.sin_addr.s_addr, sa_cli.sin_port);

/* ----- */
/* TCP connection is ready. Do server side SSL. */

ssl = SSL_new (ctx);
CHK_NULL(ssl);
SSL_set_fd (ssl, sd);
err = SSL_accept (ssl);
CHK_SSL(err);

/* Get the cipher - opt */

printf ("SSL connection using %s\n", SSL_get_cipher (ssl));

/* Get client's certificate (note: beware of dynamic allocation) - opt
*/
//SSL_CTX_set_verify(ctx,SSL_VERIFY_PEER,NULL);

```

```

client_cert = SSL_get_peer_certificate (ssl);
printf("client certificate is%s",client_cert);
if (client_cert != NULL) {
    printf ("Client certificate:\n");

    str = X509_NAME_oneline (X509_get_subject_name (client_cert), 0, 0);
    CHK_NULL(str);
    printf ("\t subject: %s\n", str);
    OPENSSL_free (str);

    str = X509_NAME_oneline (X509_get_issuer_name (client_cert), 0, 0);
    CHK_NULL(str);
    printf ("\t issuer: %s\n", str);
    OPENSSL_free (str);

    /* We could do all sorts of certificate verification stuff here before
       deallocating the certificate. */

    X509_free (client_cert);
} else
    printf ("Client does not have certificate.\n");

/* DATA EXCHANGE - Receive message and send reply. */

err = SSL_read (ssl, buf, sizeof(buf) - 1);
CHK_SSL(err);
buf[err] = '\0';
printf ("Got %d chars:'%s'\n", err, buf);

err = SSL_write (ssl, "I hear you.", strlen("I hear you.));
CHK_SSL(err);

/* Clean up. */

close (sd);
SSL_free (ssl);
SSL_CTX_free (ctx);

return 0;
}
/* EOF - serv.cpp */

```

//CLIENT CODE//

```
/* cli.cpp - Minimal ssleay client for Unix
   30.9.1996, Sampo Kellomaki <sampo@iki.fi> */

/* mangled to work with SSLeay-0.9.0b and OpenSSL 0.9.2b
   Simplified to be even more minimal
   12/98 - 4/99 Wade Scholine <wades@mail.cybg.com> */

#include <unistd.h>
#include <stdio.h>
#include <memory.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <openssl/crypto.h>
#include <openssl/x509.h>
#include <openssl/pem.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

#define CERTF "client.crt"
#define KEYF "client.key"
#define CACERT "ca.crt"

#define CHK_NULL(x) if ((x)==NULL) exit (1)
#define CHK_ERR(err,s) if ((err)==-1) { perror(s); exit(1); }
#define CHK_SSL(err) if ((err)==-1) { ERR_print_errors_fp(stderr);
exit(2); }

int main ()
{
    int err;
    int sd;
```

```

struct sockaddr_in sa;
SSL_CTX* ctx;
SSL*      ssl;
X509*     server_cert;
char*     str;
char      buf [4096];
const SSL_METHOD *meth;

SSL_load_error_strings();
meth = SSLv23_client_method();
SSL_CTX_load_error_strings();
ctx = SSL_CTX_new (meth);
CHK_NULL(ctx);
CHK_SSL(err);
    printf("initial setup done\n");

//SSL_CTX_set_verify(ctx,SSL_VERIFY_PEER,NULL);
SSL_CTX_load_verify_locations(ctx,CACERT,NULL);
    printf("\nverified cacert location\n");
if (SSL_CTX_use_certificate_file(ctx, CERTF, SSL_FILETYPE_PEM) <= 0) {
    printf("SSL_CTX_use_certificate_file failed");
    ERR_print_errors_fp(stderr);
    exit(-2);
}

if (SSL_CTX_use_PrivateKey_file(ctx, KEYF, SSL_FILETYPE_PEM) <= 0) {
    printf("SSL_CTX_use_PrivateKey_file failed");
    ERR_print_errors_fp(stderr);
    exit(-3);
}

if (!SSL_CTX_check_private_key(ctx)) {
    printf("Private key does not match the certificate public keyn");
    exit(-4);
}

/* ----- */
/* Create a socket and connect to server using normal socket calls. */

sd = socket (AF_INET, SOCK_STREAM, 0);          CHK_ERR(sd, "socket");

memset (&sa, '\0', sizeof(sa));
sa.sin_family      = AF_INET;
sa.sin_addr.s_addr = inet_addr ("10.0.0.47");  /* Server IP */
sa.sin_port        = htons      (8888);        /* Server Port number */

```

```

err = connect(sd, (struct sockaddr*) &sa, sizeof(sa));
CHK_ERR(err, "connect");

printf("successfully connected to server\n");
/* ----- */
/* Now we have TCP connction. Start SSL negotiation. */

ssl = SSL_new (ctx);
CHK_NULL(ssl);
SSL_set_fd (ssl, sd);
err = SSL_connect (ssl);
CHK_SSL(err);

/* Following two steps are optional and not required for
   data exchange to be successful. */

/* Get the cipher - opt */

printf ("SSL connection using %s\n", SSL_get_cipher (ssl));

/* Get server's certificate (note: beware of dynamic allocation) - opt
*/
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
server_cert = SSL_get_peer_certificate (ssl);
CHK_NULL(server_cert);
printf ("Server certificate:\n");

str = X509_NAME_oneline (X509_get_subject_name (server_cert), 0, 0);
CHK_NULL(str);
printf ("\t subject: %s\n", str);
OPENSSL_free (str);

str = X509_NAME_oneline (X509_get_issuer_name (server_cert), 0, 0);
CHK_NULL(str);
printf ("\t issuer: %s\n", str);
OPENSSL_free (str);

/* We could do all sorts of certificate verification stuff here before
   deallocating the certificate. */

X509_free (server_cert);

/* ----- */
/* DATA EXCHANGE - Send a message and receive a reply. */

```

```

    err = SSL_write (ssl, "Hello World!", strlen("Hello World!"));
CHK_SSL(err);

    err = SSL_read (ssl, buf, sizeof(buf) - 1);
CHK_SSL(err);
    buf[err] = '\0';
    printf ("Got %d chars: '%s'\n", err, buf);
    SSL_shutdown (ssl); /* send SSL/TLS close_notify */

/* Clean up. */

    close (sd);
    SSL_free (ssl);
    SSL_CTX_free (ctx);

    return 0;
}
/* EOF - cli.cpp */

```

//SIMPLETUN PROGRAM//

```
/*
 *
 *  * simpletun.c
 *
 *
 *
 *  * A simplistic, simple-minded, naive tunnelling program using tun/tap
 *
 *  * interfaces and TCP. Handles (badly) IPv4 for tun, ARP and IPv4 for
 *
 *  * tap. DO NOT USE THIS PROGRAM FOR SERIOUS PURPOSES.
 *
 *
 *  * You have been warned.
 *
 *  * (C) 2009 Davide Brini.
 *
 *  * DISCLAIMER AND WARNING: this is all work in progress. The code is
 *
 *  * ugly, the algorithms are naive, error checking and input validation
 *
 *  * are very basic, and of course there can be bugs. If that's not enough,
 *
 *  * the program has not been thoroughly tested, so it might even fail at
 *
 *  * the few simple things it should be supposed to do right.
 *
 *  * Needless to say, I take no responsibility whatsoever for what the
 *
 *  * program might do. The program has been written mostly for learning
 *
 *  * purposes, and can be used in the hope that is useful, but everything
 *
 *  * is to be taken "as is" and without any kind of warranty, implicit or
 *
 *  * explicit. See the file LICENSE for further details.
 *
 */
```



```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <linux/if.h>
#include <linux/if_tun.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <arpa/inet.h>
#include <sys/select.h>
#include <sys/time.h>
#include <errno.h>
#include <stdarg.h>

/* buffer for reading from tun/tap interface, must be >= 1500 */
#define BUFSIZE 2000
#define CLIENT 0
#define SERVER 1
#define PORT 55555

/* some common lengths */
#define IP_HDR_LEN 20
#define ETH_HDR_LEN 14
#define ARP_PKT_LEN 28

int debug;
char *progname;

/*****
 *
 * tun_alloc: allocates or reconnects to a tun/tap device. The caller
 *
 *          needs to reserve enough space in *dev.
 *
 *****/

/
int tun_alloc(char *dev, int flags) {

    struct ifreq ifr;
    int fd, err;

```

```

if( (fd = open("/dev/net/tun", O_RDWR)) < 0 ) {
    perror("Opening /dev/net/tun");
    return fd;
}

memset(&ifr, 0, sizeof(ifr));

ifr.ifr_flags = flags;

if (*dev) {
    strncpy(ifr.ifr_name, dev, IFNAMSIZ);
}

if( (err = ioctl(fd, TUNSETIFF, (void *)&ifr)) < 0 ) {
    perror("ioctl(TUNSETIFF)");
    close(fd);
    return err;
}

strcpy(dev, ifr.ifr_name);

return fd;
}

/*****
 *
 * cread: read routine that checks for errors and exits if an error is
 *
 *         returned.
 *
 *****/

/
int cread(int fd, char *buf, int n){

    int nread;

    if((nread=read(fd, buf, n))<0){
        perror("Reading data");
        exit(1);
    }
    return nread;
}

/*****/

```

```

*
* cwrite: write routine that checks for errors and exits if an error is
*
*         returned.
*
*****
/
int cwrite(int fd, char *buf, int n){

    int nwrite;

    if((nwrite=write(fd, buf, n))<0){
        perror("Writing data");
        exit(1);
    }
    return nwrite;
}

/*****
*
* read_n: ensures we read exactly n bytes, and puts those into "buf".
*
*         (unless EOF, of course)
*
*****
/
int read_n(int fd, char *buf, int n) {

    int nread, left = n;

    while(left > 0) {
        if ((nread = cread(fd, buf, left))==0){
            return 0 ;
        }else {
            left -= nread;
            buf += nread;
        }
    }
    return n;
}

/*****
*

```

```

* do_debug: prints debugging stuff (doh!)
*

*****
/
void do_debug(char *msg, ...){

    va_list argp;

    if(debug){
        va_start(argp, msg);
        fprintf(stderr, msg, argp);
        va_end(argp);
    }
}

/*****
*
* my_err: prints custom error messages on stderr.
*

*****
/
void my_err(char *msg, ...) {

    va_list argp;

    va_start(argp, msg);
    fprintf(stderr, msg, argp);
    va_end(argp);
}

/*****
*
* usage: prints usage and exits.
*

*****
/
void usage(void) {
    fprintf(stderr, "Usage:\n");
    fprintf(stderr, "%s -i <iface> [-s|-c <serverIP>] [-p <port>]
[-u|-a] [-d]\n", progname);
    fprintf(stderr, "%s -h\n", progname);
    fprintf(stderr, "\n");
}

```

```

    fprintf(stderr, "-i <ifacename>: Name of interface to use
(mandatory)\n");
    fprintf(stderr, "-s|-c <serverIP>: run in server mode (-s), or specify
server address (-c <serverIP>) (mandatory)\n");
    fprintf(stderr, "-p <port>: port to listen on (if run in server mode) or
to connect to (in client mode), default 55555\n");
    fprintf(stderr, "-u|-a: use TUN (-u, default) or TAP (-a)\n");
    fprintf(stderr, "-d: outputs debug information while running\n");
    fprintf(stderr, "-h: prints this help text\n");
    exit(1);
}

```

```

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

    int tap_fd, option;
    int flags = IFF_TUN;
    char if_name[IFNAMSIZ] = "";
    int header_len = IP_HDR_LEN;
    int maxfd;
    uint16_t nread, nwrite, plength;
    // uint16_t total_len, ethertype;
    char buffer[BUFSIZE];
    struct sockaddr_in local, remote;
    char remote_ip[16] = "";
    unsigned short int port = PORT;
    int sock_fd, net_fd, optval = 1;
    socklen_t remotelen;
    int cliserv = -1;    /* must be specified on cmd line */
    unsigned long int tap2net = 0, net2tap = 0;

    progname = argv[0];

    /* Check command line options */
    while((option = getopt(argc, argv, "i:sc:p:uahd")) > 0){
        switch(option) {
            case 'd':
                debug = 1;
                break;
            case 'h':
                usage();
                break;
            case 'i':
                strncpy(if_name, optarg, IFNAMSIZ-1);
                break;
            case 's':

```

```

        cliserv = SERVER;
        break;
    case 'c':
        cliserv = CLIENT;
        strncpy(remote_ip,optarg,15);
        break;
    case 'p':
        port = atoi(optarg);
        break;
    case 'u':
        flags = IFF_TUN;
        break;
    case 'a':
        flags = IFF_TAP;
        header_len = ETH_HDR_LEN;
        break;
    default:
        my_err("Unknown option %c\n", option);
        usage();
    }
}

argv += optind;
argc -= optind;

if(argc > 0){
    my_err("Too many options!\n");
    usage();
}

if(*if_name == '\0'){
    my_err("Must specify interface name!\n");
    usage();
}else if(cliserv < 0){
    my_err("Must specify client or server mode!\n");
    usage();
}else if((cliserv == CLIENT)&&(*remote_ip == '\0')){
    my_err("Must specify server address!\n");
    usage();
}

/* initialize tun/tap interface */
if ( (tap_fd = tun_alloc(if_name, flags | IFF_NO_PI)) < 0 ) {
    my_err("Error connecting to tun/tap interface %s!\n", if_name);
    exit(1);
}

```

```

}

do_debug("Successfully connected to interface %s\n", if_name);

if ( (sock_fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket()");
    exit(1);
}

if(cliserv==CLIENT){
    /* Client, try to connect to server */

    /* assign the destination address */
    memset(&remote, 0, sizeof(remote));
    remote.sin_family = AF_INET;
    remote.sin_addr.s_addr = inet_addr(remote_ip);
    remote.sin_port = htons(port);

    /* connection request */
    if (connect(sock_fd, (struct sockaddr*) &remote, sizeof(remote)) < 0){
        perror("connect()");
        exit(1);
    }

    net_fd = sock_fd;
    do_debug("CLIENT: Connected to server %s\n",
inet_ntoa(remote.sin_addr));

} else {
    /* Server, wait for connections */

    /* avoid EADDRINUSE error on bind() */
    if(setsockopt(sock_fd, SOL_SOCKET, SO_REUSEADDR, (char *)&optval,
sizeof(optval)) < 0){
        perror("setsockopt()");
        exit(1);
    }

    memset(&local, 0, sizeof(local));
    local.sin_family = AF_INET;
    local.sin_addr.s_addr = htonl(INADDR_ANY);
    local.sin_port = htons(port);
    if (bind(sock_fd, (struct sockaddr*) &local, sizeof(local)) < 0){
        perror("bind()");
        exit(1);
    }
}

```

```

    }

    if (listen(sock_fd, 5) < 0){
        perror("listen()");
        exit(1);
    }

    /* wait for connection request */
    remotelen = sizeof(remote);
    memset(&remote, 0, remotelen);
    if ((net_fd = accept(sock_fd, (struct sockaddr*)&remote, &remotelen))
< 0){
        perror("accept()");
        exit(1);
    }

    do_debug("SERVER: Client connected from %s\n",
inet_ntoa(remote.sin_addr));
    }

    /* use select() to handle two descriptors at once */
    maxfd = (tap_fd > net_fd)?tap_fd:net_fd;

    while(1) {
        int ret;
        fd_set rd_set;

        FD_ZERO(&rd_set);
        FD_SET(tap_fd, &rd_set); FD_SET(net_fd, &rd_set);

        ret = select(maxfd + 1, &rd_set, NULL, NULL, NULL);

        if (ret < 0 && errno == EINTR){
            continue;
        }

        if (ret < 0) {
            perror("select()");
            exit(1);
        }

        if(FD_ISSET(tap_fd, &rd_set)){
            /* data from tun/tap: just read it and write it to the network */

            nread = cread(tap_fd, buffer, BUFSIZE);

```



```

        tap2net++;
        do_debug("TAP2NET %lu: Read %d bytes from the tap interface\n",
tap2net, nread);

        /* write length + packet */
        plength = htons(nread);
        nwrite = cwrite(net_fd, (char *)&plength, sizeof(plength));
        nwrite = cwrite(net_fd, buffer, nread);

        do_debug("TAP2NET %lu: Written %d bytes to the network\n", tap2net,
nwrite);
    }

    if(FD_ISSET(net_fd, &rd_set)){
        /* data from the network: read it, and write it to the tun/tap
interface.
        * We need to read the length first, and then the packet */

        /* Read length */
        nread = read_n(net_fd, (char *)&plength, sizeof(plength));
        if(nread == 0) {
            /* ctrl-c at the other end */
            break;
        }

        net2tap++;

        /* read packet */
        nread = read_n(net_fd, buffer, ntohs(plength));
        do_debug("NET2TAP %lu: Read %d bytes from the network\n", net2tap,
nread);

        /* now buffer[] contains a full packet or frame, write it into the
tun/tap interface */
        nwrite = cwrite(tap_fd, buffer, nread);
        do_debug("NET2TAP %lu: Written %d bytes to the tap interface\n",
net2tap, nwrite);
    }
}

return(0);
}

```

References:

http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Networking/VPN/VPN.pdf
<https://en.wikipedia.org/wiki/SSL>
<http://waldner.netsons.org/d2-tuntap.php>
<http://www.mjmwired.net/kernel/Documentation/networking/tuntap.txt>
<http://waldner.netsons.org/d2-tuntap.php> <http://sites.inka.de/~W1011/devel/tcp-tcp.html>
<http://waldner.netsons.org/d3-ssh-tuntap.php> <http://www.madboa.com/geek/openssl/>
<http://www.securityfocus.com/infocus/1466>
<http://www.ibm.com/developerworks/linux/library/l-openssl.html>
<http://www.securityfocus.com/infocus/1388> <http://www.securityfocus.com/infocus/1462>
<http://www.madboa.com/geek/openssl/>