Computer Science 530 - Lab Assignment #9 - Confidential Communication with Tunnels, Encryption, and VPNs -- Fall 2021

Due: Friday December 2, 2022, 4:30 p.m.

(Lab will be accepted up to one week late with no penalty)

Overview

Infrastructure for Lab

Location of files

There is only one appliance (fedora30 Fall 20) that you will use in this lab. You will create 5 instances (node0, node1 node2 node3 and node4) using the populate script. You have already downloaded and loaded the ova file for this appliance, but I have also made it available in the CSci530 google drive in the folder for Lab 9 here.

Please note that you may need to login to google drive with your USC account in order to access these files.

You will note that there is a directoy with scripts (or BATCH files) for this lab. There is a directory for windows machines, and another for Linux and apple systems. Download the scripts from the directory that is relevant to your machine. The scripts in these directories are used to clone the virtual machines (populate), start them (poweron), configure the network between them (construct network, set internal settings for the guest machine (guestOS-internal-setting), power them off, and get rid of them when you are done with the lab (destroy).

You will run these scripts at the appropriate time for the experment nftables below.

Some notes on this instance of fedora Linux

We have already loaded most of the programs you will need for this lab into the virtual appliance. When you start the virtual machine you will be asked to login. The password for both the root and students accounts is "c$l@bLinuX". The third character is the letter "l" as in lab.

Confidential communication with tunnels, encryption, VPNs

Synopsis

This exercise implements several communication channels that are tunneled, encrypted, or both. Four software products are used:

- IP-in-IP
- ssh port forwarding
- stunnel
- OpenVPN

The unencrypted IP-in-IP is included for tutorial value, despite lack of security value. Given such a tunnel, encryption can be added to it. OpenVPN does that. The other two, ssh and stunnel, do not construct tunnels by strict definition. However, they do properly represent the class of wrapper products that can secure a communication channel between two points by introducing encryption at one and decryption at the other.

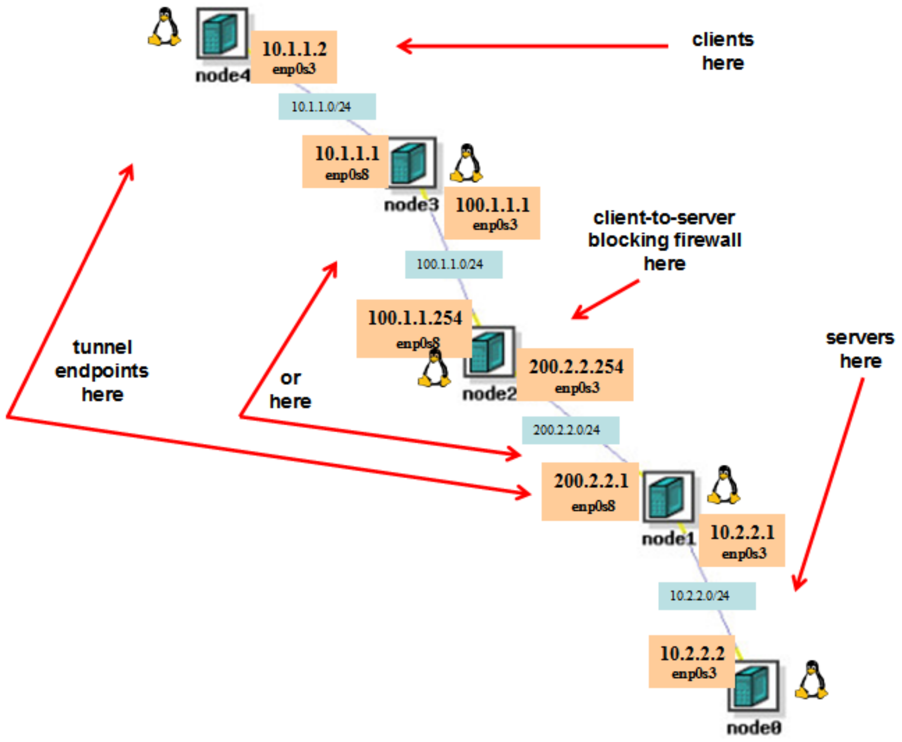
Background and recommended reading materials

- RFCs IP in IP Tunneling and IP Encapsulation within IP
- home page for stunnel project
- home page for OpenVPN

Project specification

1. Setting up the topology and tools

This exercise uses this network topology:



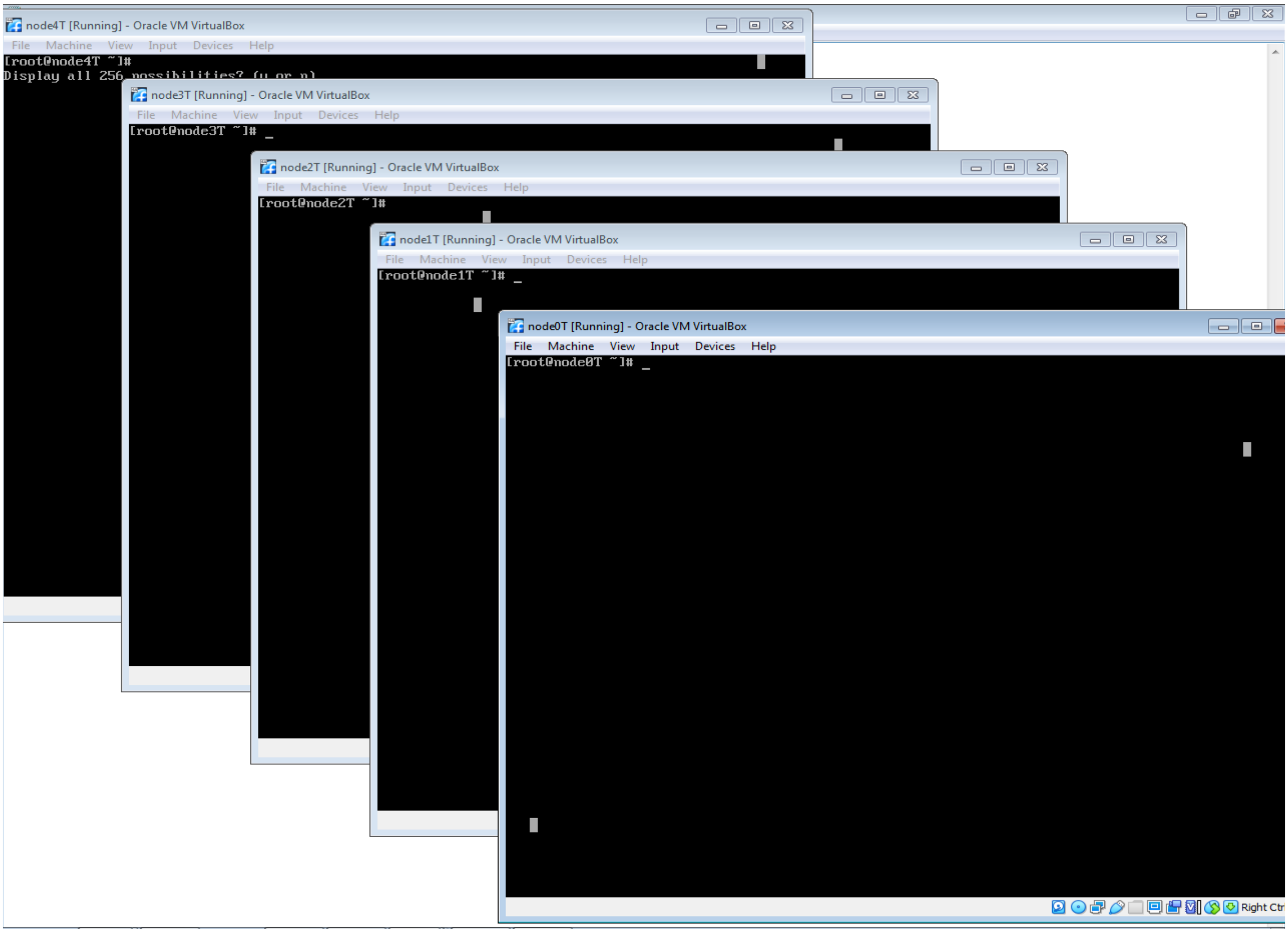
This is a 4-subnet internetwork. (Each subnet consists of an adjacent pair of hosts. Nodes 1, 2, and 3 serve as routers that join pairs of adjacent subnets.)

Use provided scripts to start your experiment.

Setting up this experiment
Scripts are used. A set of them is provided for each experiment. Please see the "Virtual Machine Usage" section of [this document](). A summary of the scripts' use and intended execution order follows. "vmconfigure-populate" is first, creates machines "vmconfigure-construct-network" is next, if it exists "vmconfigure-guestOS-internal-settings" is next, it makes the internal settings after powering the machines on; so if this one exists "vmconfigure-poweron" is not needed "vmconfigure-poweron" is next, if there is no "vmconfigure-guestOS-internal-settings" Now you can use your VM(s). When you are finished: "vmconfigure-poweroff" "vmconfigure-destroy" if you want to delete the machines, but not if you plan to come back to them later If you do come back to them later, "vmconfigure-guestOS-internal-settings" should be repeated, if present "vmconfigure-poweron" if there is not "vmconfigure-guestOS-internal-settings" present

Log in to each VM as root. Set up your monitor screen to contain 5 terminal windows, one to each of the experiment nodes arranged as in the graphic below.

Your screen will look something like the following, with the windows stacked in the same order as in the above figure, and the title bars accessibly arranged for single-click selection of the window of any desired node (they may also be selectable by using keystrokes, possibly alt-Tab, or task bar icons).



<u>Working servers for experimentation</u>

To use tunnels we need a tunnel destination, a server to which to send data. First as a reference case we can interact with that server normally, untunneled. Then we can run the same interaction through the various tunnel alternatives we will create, optionally obstructing the untunneled data path to verify the tunnel's functionality.

Let's use a couple of servers, and let's choose node0 as their placement location. We'll interact with them from node4 at the opposite end of the internetwork. node0 will make available
 an echo server running on udp port 7
 an echo server running on tcp port 7
 a apache/httpd web server running on tcp port 80

Run the apache server on node0 as follows:

systemctl start httpd

In order to start the echo servers, edit two files. They are /etc/xinetd.d/echo-dgram and /etc/xinetd.d/echo-stream. In each, change the line that reads "disable = yes" to "disable = no" instead. If you are not comfortable using the vi editor, you can accomplish the same thing programatically instead by typing carefully:

sed -i '/disable/s/yes/no/' /etc/xinetd.d/echo-dgram
sed -i '/disable/s/yes/no/' /etc/xinetd.d/echo-stream

after making the changes, make the echo servers run:

systemctl restart xinetd

You should Verify the servers' presence. On node0:

netstat -pantu | grep -E "httpd|xinetd"

You should see something like:

```
[root@node0T ~]# netstat -pantu | grep -E "xinetd|httpd"
tcp6       0      0 :::80                   :::*                    LISTEN      877/httpd
tcp6       0      0 :::7                    :::*                    LISTEN      1097/xinetd
udp6       0      0 :::7                    :::*                                1097/xinetd
```

What about suitable clients for these servers? A suitable node4 echo client to run against the echo servers is netcat (nc). A suitable web client to run against apache is lynx. Let's use these clients to test the servers. On node4:

nc -u node0 7

It will silently wait for you to type something on the next line. Type the following then press enter:

The cow jumped over the moon.

It will appear a second time. The first time you see it on the screen it came from you, while you typed it in; the second time you see it, it came (echoed) from the remote machine (node0). Stop it with a ctrl-C keystroke.

Do it again, but use the remote machines tcp echo service as opposed to its udp echo service. (The remote node0 is running both, they are distinct, and their port 7's are distinct. The udp protocol has its own port numbers as does tcp, and the two protocols' numbers don't overlap.) On node4:

nc -t node0 7

It will silently wait for you to type something on the next line. Type the following then press enter:

The cow jumped over the moon.

It will appear a second time. The first time you see it on the screen it came from you, while you typed it in; the second time you see it, it came (echoed) from the remote machine (node0). Stop it with a ctrl-C keystroke.

(The convenience of using "node0" here instead of "10.2.2.2" works because they are mapped together for you in each local /etc/hosts file. Use the IPs themselves if you prefer but the names are more recognizable.) "The cow jumped over the moon." has appeared on your screen, having been bounced back to you by the server at the other end. Let's sniff this e*n route*, at node2. On node2:

tcpdump -xXnnti enp0s8 [of its 2 interfaces, we are sniffing on node2's "upper" one in terms of the above topology map]

Now go back to node4 and repeat the above client "echo..." commands. When they run, note the activity on node2's screen showing the passage of traffic back and forth. Scrutinize it till you locate within it on node2's screen the "...cow..." phrase. It's fully legible, not encrypted.

Similarly let's test the web server. It will serve a default web page named index.html if there is one. There isn't, so let's create one and make it distinguishing. On node0:

echo '<h1>Hi, you have reached node0.</h1>' > /var/www/html/index.html

Test it by browsing this page from node4 using the lynx character-mode web browser. (Browsers have two halves-- the server interaction half, and the local display half. Lynx is no different from any other browser in terms of server interaction, which we care about; it lacks the local-display half of familiar GUI browsers, but we don't care about that. Minimalist, non-GUI display is good enough for our network diagnostic purposes.) On node4 browse node0 (10.2.2.2):

lynx http://node0:80

Make sure the "Hi, you have reached node0." from node0's default web page reaches your screen. (lynx tutorial: q followed by y will quit; ctrl-r will refresh.)

These interactions are working normally, the old fashioned way. That is, by using standard routing. Your provided scripts have placed routes (in the machines' routing tables) to enable all our nodes up and down the line to reach each other. In particular there are no tunnels involved.

We're going to build some. We'll test them by running these interactions through them, instead of through standard routing as just seen. So we will want to defeat the operation of standard routing for these purposes and the tunnels' acid test. node2 in the middle is a good place to put an obstructive firewall rule in the iptables FORWARDing chain. We could make a rule that blocks the traffic by its port numbers or by its source and/or destination IP addresses. Let's do the latter. On node2:

iptables -A FORWARD -s node4 -d node0 -j DROP

It says, "Disallow IP from node4 to node0." That's sufficient. Go back to node4 and repeat the two client echo|nc commands and the one lynx command. Now, no data comes back. The original data never reach node0 because it can't get through node2. If you wanted to remove the obstruction-- but for purposes of our exercise please leave it in place-- you would run the above iptables commands with -A replaced by -D. (Examining/verifying the firewall ruleset in effect on node2 can be done with "iptables -nL". It will show the obstruction, referencing the nodes in IP terms as "10.1.1.2" and "10.2.2.2".)

2. IP-over-IP

IP "over" (or "in") IP takes advantage of the fact that what an IP packet carries is data, data has many varieties and, among them, IP packets are themselves data. So there is nothing to stop an IP packet from carrying another IP packet. That's what IP-over-IP does. It is formalized as a standard protocol, assigned protocol number 4. An IP packet that's carrying another IP packet as its data declares so by bearing the number 4 in the "protocol" field of its header.

```
IP Header Format

 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

4 for IP
(6 for TCP
17 for UDP
50 for ESP, etc)

We will put IP-over-IP tunnel endpoints on nodes 1 and 3. That is, we'll create new tunnel interfaces on them (with the "ifconfig" command). Then with IP routing, which corresponds destinations with interfaces, we will tell node1 that the remote 10.1.1.0/24 network is reachable as a destination through its tunnel interface (with the "route" command). Similarly we'll tell node 3 that the remote 10.2.2.0/24 network is reachable through *its* tunnel interface. For each of the 2 nodes a script that does this is provided.

In your node1 terminal window, obtain and execute its script (examine it briefly if you wish):

cd
cp /home/public/tunl-ipip-node1 . (observe the final dot, shorthand for "current directory," an important part of this syntax)
chmod +x tunl-ipip-node1
./tunl-ipip-node1 (when it displays a routing table press enter to see a second one; they are "before" and "after" tunnel construction)
[if you get a "SIOCDELRT:" error message you can ignore it]

and similarly in your node3 terminal window:

cd
cp /home/public/tunl-ipip-node3 .
chmod +x tunl-ipip-node3
./tunl-ipip-node3 (when it displays a routing table press enter to see a second one; they are "before" and "after" tunnel construction)
[if you get a "SIOCDELRT:" error message you can ignore it]

The tunnels are now in place. View them. On both node1 and node3:

ifconfig tunl0

Go back to node4 and repeat the two client echo|nc commands and the one lynx command. It's working again, despite the node2 firewall. That's because the data comes to node2 inside the tunnel, to which its firewall does not apply. Try this too, from node4:

ping -c 1 node0

To see explicit evidence of the tunnel let's use tcpdump to sniff some of the interfaces the data must traverse to travel over-and-back. There are 8 of them (look at the diagram, where each address belongs to an interface). We'll selectively look at several of them. As an example, in your node3 terminal window start the tcpdump command on the interior interface, the one addressed as 10.1.1.1, on interface enp0s8. In the node3 terminal window, initiate tcpdump:

tcpdump -nnti enp0s8

Now, back in the node4 terminal window again ping:

ping -c 1 node0

A packet passes through, addressed between the node4 and node0 ping endpoints. Another passes back, addressed in reverse. And, importantly, they are both ping (ICMP echo) packets (i.e., IP packets that carry ping packets). The dump looks similar to this:

```
IP 10.1.1.2 > 10.2.2.2: ICMP echo request, id 44314, seq 1, length 64
IP 10.2.2.2 > 10.1.1.2: ICMP echo reply, id 44314, seq 1, length 64
```

Now do the same thing but watch the data passage at node3's exterior interface. That's the one with IP 100.1.1.1, on interface enp0s3. In the node3 terminal window initiate tcpdump:

tcpdump -nnti enp0s3

and back in the node4 terminal window again ping:

ping -c 1 node0

A packet passes through, and another passes back reverse-addressed. But they are addressed between nodes 3 and 1, the tunnel endpoints; not between nodes 4 and 0, the ping endpoints. And importantly, *they are **not** ping packets*. The dump looks like this:

```
IP 100.1.1.1 > 200.2.2.1: IP 10.1.1.2 > 10.2.2.1: ICMP echo request, id 38682, seq 1, length 64 (ipip-proto-4)
IP 200.2.2.1 > 100.1.1.1: IP 10.2.2.1 > 10.1.1.2: ICMP echo reply, id 38682, seq 1, length 64 (ipip-proto-4)
```

In terms of carried data, rather than ping packets they are IP packets; that is, instead of IP packets that carry ping packets, we have IP packets that carry other IP packets (the essence of IP-over-IP). And the addressing of the carried ones, differing from the carrying ones, is between *ping* endpoints not *tunnel* endpoints.

Repeat this for yourself on a few of the other interfaces. Note that north of node3 and south of node1 all is normal ping. But at any of the in-between interfaces the traffic is tunnel traffic, IP-over-IP. If you add a -v to your tcpdump command to make it verbose, it will reveal the IP header's protocol field, showing on the interior interface something like:

```
IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto: ICMP (1), length: 84) 10.1.1.2 > 10.2.2.2: ICMP echo request, id 47130, seq 1, length 64
```

and on the exterior interface:

```
IP (tos 0x0, ttl 63, id 0, offset 0, flags [DF], proto: IPIP (4), length: 104) 100.1.1.1 > 200.2.2.1: IP (tos 0x0, ttl 63, id 0, offset 0, flags [DF], proto: ICMP (1), length: 84) 10.1.1.2 > 10.2.2.2: ICMP echo request, id 47642, seq 1, length 64
```

Note the "proto: ICMP (1)" in the interior case, and the "proto: IPIP (4)" in the exterior. Also in the exterior case notice that the carried data its represented as the IP packet it is, complete with its own "proto: ICMP (1)".

Let's neutralize the tunl0 interfaces. On node1 and node3:

ifconfig tunl0 down
ifconfig tunl0 0.0.0.0

3. ssh

The most common, baseline use of ssh is to establish a character-mode login session on another machine, much like telnet. The machine where you wish to log in must run an ssh server, and you must have an account there. The advantage of ssh is that your client and the remote server automatically encrypt and decrypt everything they send to each other. What if you're interested in communication not with the ssh server, but another computer to which it is attached in a local network? ssh has a secondary feature that accomplishes this, called ssh port forwarding.

Let's restate this in terms of our experiment setting. If you are at node4 and have an account on node1 then you can gain a login shell on node1 provided it runs an ssh server and you run an ssh client. Everything going back and forth will be encrypted. But what if you're interested in communication not with node1, but node0 to which it is attached? ssh port forwarding can help. You're not limited just to logging in to node0, but can communicate with any service it runs with the help of ssh to get your packets through. (This *could* encompass login in to node0 if node0 itself runs an ssh server, but can instead be browsing node0 should it run a web server or sending files to node0 should it run an ftp server, or using any service node0 offers.) The communication traffic between node4 and node0, whatever it is, travels in encrypted form as far as node1; it is unencrypted there and goes the rest of the way unencrypted.

Establish a forwarding tunnel to the target ports 7 and 80 on the target machine node0

First, from node4 use ssh to log in to node1. But supply the extra syntax for the port forwarding feature. On node4:

ssh -fN -L 1007:node0:7 -L 1080:node0:80 student@node1
(local ports 1007 and 1080 become surrogates for remote 7 and 80)

When prompted for login, give your user username and password (the same as you use for DETER iteself). You become logged in to node1. However the -fN options provide a little trick, putting this logged-in ssh client session into the background. Consequently you don't get node1's shell prompt as usual, but revert back immediately to your original shell on node4. Nevertheless, your login to node1 is sustained and the port forwarding for those two ports is in place. (If you want to see it, the "ps ax" command will show the ssh process.) Check that ssh on your machine is indeed listening to ports 1007 and 1080. On node4:

netstat -pant | grep -E "1007|1080"

Connect to the target port of target machine with a client that matches the service running on that port

Test that both echo and http traffic gets though. On node4:

lynx 127.0.0.1:1080
and
echo hellooooooooooooooooo | nc -t 127.0.0.1 1007

Note that node4 is talking to itself (127.0.0.1)! Yet the the conversation is carried to node0 and that's where the responses come from. If you care to sniff the echo interaction, you'll see that the data is encrypted (you can't see hellooooooooooooooooo) north of node1 but visible between there and node0.

We're done, but don't forget that ssh client is running in the background. Let's clean it up. On node4:

killall ssh

4. stunnel

In the standard routed scenario above, the lynx client on node4 connects and talks directly to the apache server on node0, port 80. We want to reorganize this a bit by making the following changes:

1) disconnect node4 lynx from node0 apache, have him talk to node3 stunnel instead.
2) place stunnel on node3 and node1 and have them talk to each other.
3) disconnect node0 apache from node4 lynx, have him hear from node1 stunnel instead.

Configure stunnel on node 3 as a client and node1 as a server, for which the config files are provided. First on node3:

cp /home/public/stunnel-client.conf /etc/stunnel/

Then on node1:

cp /home/public/stunnel-server.conf /etc/stunnel/

Here are those two configuration files you just copied:

Server config on node3:	Client config on node1:
client=yes [speak to web server] accept=2000 connect=200.2.2.1:30000	cert=/etc/stunnel/stunnel.pem [hear web browser] accept=30000 connect=10.2.2.2:80

We want to preserve the association of apache with port 80. By contrast, we don't care what ports the other 2 conversations utilize. So let node4 lynx reach node3 stunnel using node3's port 2000, let node3 stunnel reach node1 stunnel using node1's port 30000, and let node1 stunnel reach node0 apache using node0's port 80. That's what these files configure. The choices of ports 2000 and 30000 are arbitrary.

The server copy of stunnel needs a certificate for this to work, and we can create one. Do the following on the stunnel server machine, node1:

cd /etc/stunnel
openssl req -new -x509 -days 3650 -nodes -out stunnel.pem -keyout stunnel.pem (accept all the defaults)
chmod 600 stunnel.pem

Finally, run the 2 copies of stunnel giving their respective config files on the command lines. On node3:

stunnel /etc/stunnel/stunnel-client.conf

And on node1:

stunnel /etc/stunnel/stunnel-server.conf

Nothing happened. But stunnels are running on both boxes, willing to listen to the browser and talk to the server (respectively). Check for stunnel listening on the node3's port 2000 and on node1's 30000 by running this command on both:

netstat -pant | grep stunnel

It lists the various ports being listened to and picks out the one(s) with stunnel as listener. You should see something like:

```
[root@node3 dbm]# netstat -pant | grep stunnel
tcp        0      0 0.0.0.0:2000                0.0.0.0:*                   LISTEN      13467/stunnel
```

and:

```
[root@node1 stunnel]# netstat -pant | grep stunnel
tcp        0      0 0.0.0.0:30000               0.0.0.0:*                   LISTEN      13365/stunnel
```

Now use your stunnels. On node4 bring up the lynx web browser again. But this time, point it to node3 and also specify the port to talk to. That port is 2000. On node4:

lynx http://node3:2000

You should see the server's default web page appear. Now, however, the traffic is passing between the pair of stunnels imposed between browser and server. And they are encrypting it. Watch tcpdump on node2 while doing this and note the targeting of node1's port 30000 as specified in the stunnels' configurations. Verify the conversation's dependency on stunnel by killing one of the stunnels. On node3:

killall stunnel

then on node4 try to refresh the browser. It won't. It's cut off from stunnel so the traffic route is disrupted.

5. OpenVPN

We will run 3 OpenVPN scenarios.

- a routed tunnel, unencrypted
- a routed tunnel, encrypted using static preshared keys
- a bridged tunnel, encrypted using SSL

The tunnel endpoints will be node4 and node1 (see the network diagram). node4 plays the role of the road warrior in a hotel, while node1 plays that of the gateway in an office network, that network being 10.2.2.0/24 in our diagram. Warrior node4 wants access to machines in the office, such as node0.

OpenVPN is installed on node4 and node1. Producing the 3 scenarios is a matter of putting the corresponding configuration files on these nodes. OpenVPN uses directory /etc/openvpn to keep its config files. I've prepared appropriate files for you to put there. Do so, separately on each of the 2 nodes, as follows:

On node4:
cd /etc/openvpn
tar -xvf /home/public/etcopenvpn-node4.tar

On node1:
cd /etc/openvpn
tar -xvf /home/public/etcopenvpn-node1.tar

Scenario 1: routed tunnel, unencrypted

On node1:
openvpn /etc/openvpn/server-unencrypted.conf &

Then on node4:
openvpn /etc/openvpn/client-unencrypted.conf &

Let's look at the resulting screenshots and try to interpret in light of the config files. The screens will look something like this:

root@node4:/etc/openvpn

```
[root@node4 openvpn]# openvpn /etc/openvpn/client-unencrypted.conf
Wed Nov 11 13:48:34 2009 OpenVPN 2.1_rc4 i386-redhat-linux-gnu [SSL] [LZO2] [EPOLL] built on Apr 26 2007
Wed Nov 11 13:48:34 2009 IMPORTANT: OpenVPN's default port number is now 1194, based on an official port number assignment by IANA.  OpenVPN 2.0-beta16 and earlier used 5000 as the default port.
Wed Nov 11 13:48:34 2009 ******* WARNING *******: all encryption and authentication features disabled -- all data will be tunnelled as cleartext
Wed Nov 11 13:48:34 2009 TUN/TAP device tun0 opened
Wed Nov 11 13:48:34 2009 /sbin/ip link set dev tun0 up mtu 1500
Wed Nov 11 13:48:34 2009 /sbin/ip addr add dev tun0 local 10.20.30.2 peer 10.20.30.1
Wed Nov 11 13:48:34 2009 UDPv4 link local (bound): [undef]:1194
Wed Nov 11 13:48:34 2009 UDPv4 link remote: 200.2.2.1:1194
Wed Nov 11 13:48:43 2009 Peer Connection Initiated with 200.2.2.1:1194
Wed Nov 11 13:48:45 2009 Initialization Sequence Completed
```

root@node1:/etc/openvpn

```
[root@node1 openvpn]# openvpn /etc/openvpn/server-unencrypted.conf
Wed Nov 11 13:48:33 2009 OpenVPN 2.1_rc4 i386-redhat-linux-gnu [SSL] [LZO2] [EPOLL] built on Apr 26 2007
Wed Nov 11 13:48:33 2009 IMPORTANT: OpenVPN's default port number is now 1194, based on an official port number assignment by IANA.  OpenVPN 2.0-beta16 and earlier used 5000 as the default port.
Wed Nov 11 13:48:33 2009 ******* WARNING *******: all encryption and authentication features disabled -- all data will be tunnelled as cleartext
Wed Nov 11 13:48:33 2009 TUN/TAP device tun0 opened
Wed Nov 11 13:48:33 2009 /sbin/ip link set dev tun0 up mtu 1500
Wed Nov 11 13:48:33 2009 /sbin/ip addr add dev tun0 local 10.20.30.1 peer 10.20.30.2
Wed Nov 11 13:48:33 2009 UDPv4 link local (bound): [undef]:1194
Wed Nov 11 13:48:33 2009 UDPv4 link remote: 10.1.1.2:1194
Wed Nov 11 13:48:43 2009 Peer Connection Initiated with 10.1.1.2:1194
Wed Nov 11 13:48:45 2009 Initialization Sequence Completed
```

(At this point press enter on both to see a fresh command prompt; OpenVPN continues to run in the background leaving you free to issue other commands.) The config files you invoked, responsible for this, are:

On node4 OpenVPN client	On node1 OpenVPN server
remote 200.2.2.1 dev tun0 ifconfig 10.20.30.2 10.20.30.1 route 10.2.2.0 255.255.255.0	remote 10.1.1.2 dev tun0 ifconfig 10.20.30.1 10.20.30.2

In these files
line 1 - each copy of OpenVPN is pointed to the machine where the other one is
line 2 - they are told to construct new local interfaces, both named tun0, to become tunnel endpoints
line 3 - they are told what addresses to give their interface, and what address the other will have (note the specs are reciprocal)
line 4 - the client is told to give its routing table a route to the network behind the server, gatewayed through the server's tunnel endpoint address

Read the screen messages in this light. Note the chosen tunnel endpoint addresses (arbitrarily 10.20.30...) are outside any existing subnet. The endpoint machines can still refer to each other with their original addresses but can now also do so by these new ones. Traffic to the new one will be encapsulated through a tunnel that uses UDP to carry its encapsulated traffic. Note also the warning in the screen messages that this tunnel doesn't encrypt what it passes.

Investigate some of this. On both machines:

```
ifconfig
```

Note the presence of the two tun0 interfaces. Note their type and addresses. They are point-to-point (good only to reach a single machine-- the other). Next, on both machines:

```
route  -n
```

Note that there is a new route on the server, to the client (10.20.30.2), through the tun0 interface. And that there are *two* new routes on the client. A reciprocal one to the server (10.20.30.1) through the tun0 interface. Plus another to the remote network (10.2.2.0/24), through the server as gateway (owing to line4 in the config file).

Now let's send 2 pings from client to server, one addressed to the server's physical interface address and the other to its new tunnel interface address. While doing it, sniff the passing traffic at node2. On node2:

```
tcpdump  -nnti  ethX  icmp  or  udp
```

Then on node4:

```
ping  -c1  -p48656c6c6f 200.2.2.1; ping  -c1  -p48656c6c6f  10.20.30.1
```

This compound command sends two pings, one to each address and carrying the word "Hello" (in ASCII, 48656c6c6f).

Look at both pings on node2. A ping is a paired ICMP "echo request" and answering "echo reply". In the node2 dump do you see such a pair? That's a ping. node4 pinged twice, so do you see *two* such pairs? You see 2 exchanges, but only one of them is ICMP echo. That's the ping to 200.2.2.1. But what about the ping to 10.20.30.1, do you see it? Do you even see "10.20.30.1" anywhere? It is more revealing to ask tcpdump to show the entire content of the traffic. So modify the node2 tcpdump command slightly and repeat. On node2:

```
tcpdump  -xXnnti  enp0s8  icmp  or  udp
```

Then on node4 again:

```
ping  -c1  -s30  -p48656c6c6f  200.2.2.1; ping  -c1  -s30  -p48656c6c6f  10.20.30.1
```

node2 looks something like this:

```
root@node2:/users/dbm
[root@node2 dbm]# tcpdump -xXnnti eth1 icmp or udp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 96 bytes
IP 10.1.1.2 > 200.2.2.1: ICMP echo request, id 43064, seq 1, length 38
        0x0000:  4500 003a 0000 4000 3f01 66bd 0a01 0102  E..:..@.?.f.....
        0x0010:  c802 0201 0800 3aad a838 0001 ec48 fb4a  ......:..8...H.J
        0x0020:  ce29 0700 6c6f 4865 6c6c 6f48 656c 6c6f  .)..loHelloHello
        0x0030:  4865 6c6c 6f48 656c 6c6f                 HelloHello
IP 200.2.2.1 > 10.1.1.2: ICMP echo reply, id 43064, seq 1, length 38
        0x0000:  4500 003a 1565 0000 3f01 9158 c802 0201  E..:.e..?..X....
        0x0010:  0a01 0102 0000 42ad a838 0001 ec48 fb4a  ......B..8...H.J
        0x0020:  ce29 0700 6c6f 4865 6c6c 6f48 656c 6c6f  .)..loHelloHello
        0x0030:  4865 6c6c 6f48 656c 6c6f                 HelloHello
IP 10.1.1.2.1194 > 200.2.2.1.1194: UDP, length 58
        0x0000:  4500 0056 0000 4000 3f11 6691 0a01 0102  E..V..@.?.f.....
        0x0010:  c802 0201 04aa 04aa 0042 2110 4500 003a  .........B!.E..:
        0x0020:  0000 4000 4001 ea98 0a14 1e02 0a14 1e01  ..@.@...........
        0x0030:  0800 5f9f a938 0001 ec48 fb4a a837 0700  .._..8...H.J.7..
        0x0040:  6c6f 4865 6c6c 6f48 656c 6c6f 4865 6c6c  loHelloHelloHell
        0x0050:  6f48                                     oH
IP 200.2.2.1.1194 > 10.1.1.2.1194: UDP, length 58
        0x0000:  4500 0056 0000 4000 3f11 6691 c802 0201  E..V..@.?.f.....
        0x0010:  0a01 0102 04aa 04aa 0042 2110 4500 003a  .........B!.E..:
        0x0020:  7826 0000 4001 b272 0a14 1e01 0a14 1e02  x&..@..r........
        0x0030:  0000 679f a938 0001 ec48 fb4a a837 0700  ..g..8...H.J.7..
        0x0040:  6c6f 4865 6c6c 6f48 656c 6c6f 4865 6c6c  loHelloHelloHell
        0x0050:  6f48                                     oH

4 packets captured
4 packets received by filter
0 packets dropped by kernel
[root@node2 dbm]#
```

Of these two packet-pair exchanges, which one's packets are fatter? What do they have that the skinny ones don't? Are they encrypted? Are they tunneled?

What if node4 wants to ping through node1 all the way in to node0? Let's do it and see which kind of exchange is chosen for that. On node2:

tcpdump -xXnnti ethX icmp or udp

Then on node4 again:

ping -c1 -p48656c6c6f 10.2.2.2

Does this use the skinny- or fat-packet exchange method? ICMP echo or UDP? tunneled or not? Encrypted? The dump you're looking at on node2 tells you.

We ran OpenVPN in the background so to stop it we must manually kill it. On both node1 and node4:

killall openvpn

Scenario 2: routed tunnel, encrypted using static preshared keys

We will do much the same thing now with the addition of a static key, a copy of which each node shares. Below we now create it, distribute it, modify the config files to use it, and re-run OpenVPN to do so.

On node1:

cd /etc/openvpn
openvpn --genkey --secret static.key
scp static.key <your account name>@node4:/tmp (give your password when prompted)

On node4:

mv /tmp/static.key /etc/openvpn/

Run OpenVPN on the endpoint nodes. On node1:
openvpn /etc/openvpn/server-statickey.conf &

Then on node4:
openvpn /etc/openvpn/client-statickey.conf &

The config files you invoked are just a little different:

On node4 OpenVPN client	On node1 OpenVPN server
remote 200.2.2.1 dev tun0 ifconfig 10.20.30.2 10.20.30.1 route 10.2.2.0 255.255.255.0 secret /etc/openvpn/keys/static.key	remote 10.1.1.2 dev tun0 ifconfig 10.20.30.1 10.20.30.2 secret /etc/openvpn/keys/static.key

We added the last line to each file telling it to use the key, that is, to encrypt. The key files on the two nodes are identical.

Note that the "encryption features disabled" warning message doesn't appear on screen this time.

On node2:

tcpdump -xXnnti enp0s8 icmp or udp

Then on node4, press enter to gain a shell prompt, and again:

ping -c1 -p48656c6c6f 10.2.2.2

What's different this time? Does this use the skinny- or fat-packet exchange method? ICMP echo or UDP? tunneled or not? Encrypted? The dump you're looking at on node2 tells you.

On both node1 and node4:

killall openvpn

Scenario 3: bridged tunnel, encrypted using SSL

NOTE November 6, 2020 This section will not work using the files distributed within your VMs, as written. That's because, apologies, they include certificate files that have now expired. Workaround 1 - temporarily set the date back to a point before November 2019. Do this on your host computer, the one on which you run VirtualBox. I find that merely setting the date on VMs doesn't work because the VMs revert to the host's date. Do this only if you are willing. There might be other things that depend on the host date. Or, Workaround 2 - read this section without performing it, or omit it altogether. You can answer the questions below without necessarily having performed this section.

We will do it yet again, this time changing the encryption method. We will also use bridging instead of routing, meaning that an IP address from the office network 10.2.2.0/24 will be extended and applied to warrior node4. As if a cable from the office switch were extended cross-country to him. That is, as if he were sitting in the office.

OpenVPN won't do the bridging, but will use it. For bridging there are several commands (e.g., brctl), installed on node1 already, that do the job. We will run them, from a script named bridge-start, before we run OpenVPN. Prior to using that script a small edit is required. Line 18 needs to be replaced such that the line will read "eth=enp0s3". Make that change with an editor, or you can do it with the following stream editor (sed) command, on node1:

sed -i -e '19i\eth=enp0s3' -e '18d' bridge-start

Instead of a shared, static key we will place key and certificate files generated by OpenSSL. Creating and placing them is a process sufficiently complex that some "simplification scripts" are available from a package named easy-rsa. For this exercise I pre-ran the scripts, produced the files, and put them in the tar file you used. In effect you distributed them when you unpacked the tar file, above. They are already in place. (Distributing this way defeats security, but here the purpose is tutorial.)

The new config files are substantially different:

On node4 OpenVPN client	On node1 OpenVPN server
  client dev tap0 proto udp remote 200.2.2.1 1194 resolv-retry infinite nobind persist-key persist-tun ca /etc/openvpn/keys/ca.crt cert /etc/openvpn/keys/node4.crt key /etc/openvpn/keys/node4.key comp-lzo verb 3	port 1194 proto udp dev tap0 ca /etc/openvpn/keys/ca.crt cert /etc/openvpn/keys/node1.crt key /etc/openvpn/keys/node1.key dh /etc/openvpn/keys/dh1024.pem ifconfig-pool-persist ipp.txt server-bridge 10.2.2.0 255.255.255.0 10.2.2.50 10.2.2.59 keepalive 10 120 comp-lzo user nobody group nobody persist-key persist-tun status openvpn-status.log verb 3

The biggest differences are 1) the references to the cryptographic files in the "ca", "cert", "key", and "dh" directives, 2) the use of "dev tap0" instead of "dev tun0" as name of the virtual interface device to be constructed, and 3) the "server-bridge 10.2.2.0 255.255.255.0 10.2.2.50 10.2.2.59" directive on the server side. The crypto directives make OpenVPN use SSL. tap devices as opposed to tun devices encapsulate data-link layer ethernet as opposed to network layer IP. That is, they bridge traffic through instead of routing it through. And the server-bridge directive engages OpenVPN with the bridge produced by brctl and allocates addresses to clients out of the given subrange (10.2.2.50-59) of the given network (10.2.2.0/255.255.255.0). In practice, to maintain address uniqueness, a system administrator would avoid allocating the subrange IPs to office computers and leave them for dynamic allocation to connecting warriors.

Run OpenVPN on the endpoint nodes with the new configuration:

First on node1:
/etc/openvpn/bridge-start
openvpn /etc/openvpn/server-bridged.conf &

Then on node4:
openvpn /etc/openvpn/client-bridged.conf &

Let's see what we have done. On node4 (press enter to regain shell prompt):

ifconfig

Note the presence of the tap0 interface. Note its address and type. The address, probably 10.2.2.50, comes from the range in the "server-bridge" directive on node1. It is of the regular shared-medium subnet type just like a regular ethernet NIC would be (as opposed to point-to-point) bearing an address from the same office subnet as office machines. It's on the same footing as office machines. It is to an office machine like any other office machine. It might as well actually be in the office instead of the hotel. Look at its routing table. On node4:

route -n

There is a route like this:

```
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.2.2.0        0.0.0.0         255.255.255.0   U     0      0        0 tap0
```

telling node4 that its tap0 interface is the avenue to the 10.2.2.0/255.255.255.0 network. Note it is *not* gatewayed. Putting a frame out the tap0 interface is putting it at the doorstep of the computers in the 10.2.2.0 network, with no intermediating help from any stepping-stone router. This is standard intra-LAN, local delivery. Warrior node4 is LAN-attached, effectively.

We can demonstrate this by showing that ethernet broadcast traffic from node4 reaches node1. To generate ethernet broadcast let's use arp. It emits its discover requests in the form of ethernet broadcast frames, addressed to the broadcast address FF:FF:FF:FF:FF:FF. Trying to ping a machine that's considered local will generate an arp broadcast looking for it. First on node0:

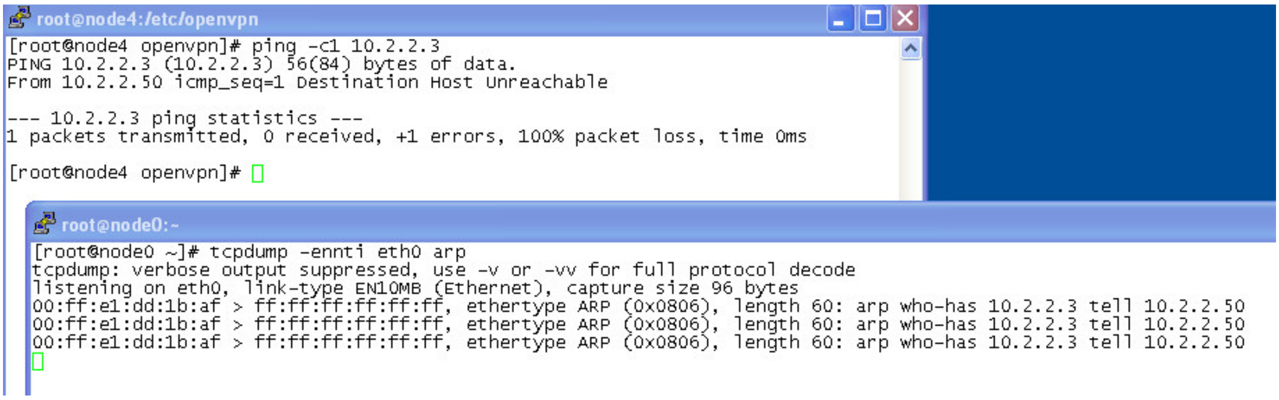
tcpdump -ennti ethX arp

tcpdump's "e" option will reveal ethernet frame headers. Then on node4:

ping -c1 10.2.2.3

10.2.2.3 doesn't exist so won't reply, but node4's appeals seeking it, which are frame broadcasts, will be seen at node0. Surprise number 1 is that node4 evidently thinks node0 is local, and surprise number 2 is that this thinking is not misplaced because the broadcasts get where they need to go.

You should see something like this. It's well-known that ethernet can't natively cross a router. Yet here it is:



This is not IP. Routing routes IP only, not ethernet. There is no IP in these frames so they did not get routed here, rather their source node4 is "here" in the first place. Moreover, between node4 and the gateway node1 the traffic was encrypted.

What can go wrong

You can execute a command on a different node than intended. Pay attention to which nodes the instructions ask you operate on. Some instructions are for one node, others for a different one.

Questions for you to answer

1. In the topology diagram for this experiment there are 2 pairs of arrows showing where an encrypted tunnel's endpoints might be placed-- either on nodes 1 and 3 or on nodes 1 and 4. The experiment included examples of both usages. ssh and stunnel did it the first way and OpenVPN used the second. Does it matter? Which tunnel endpoint option is "better," node3 or node4? Answer in two parts 1a and 1b per below.

a. Considering the fact that the application clients in this experiment are always housed on node4, choose one of the tunnel endpoint options and make the case for why it is the better one.

b. Imagine that diagram's upper network 10.1.1.0/24 were expanded to 100 or 200 nodes. Would any new or different rationales and arguments for placement of the tunnel endpoint arise? Considering the fact that the population of machines that could potentially benefit from secure communication is large, choose one of the tunnel endpoint options and make the case for why it is the better one.

2. We implemented 3 products that perform encryption but said little about how they do it. Do they all use the same cipher algorithm in common? If not, does each one use the same algorithm every time? Could we say that one product is more secure, in the encryption it applies, than another? Key sizes? Encryption modes? HMACs? Do a little bit of research, simply at the level of reading the man pages to find out what they have to say about this ("cipher" is a good search key). Then write a very short summary of how things work and what you find to be the salient issues discussed in the man pages. For your convenience here are the man pages: ssh sshd stunnel openvpn

You could go deep into this but that's not what we want. Just demonstrate having visited the man pages and awareness what the issues are.

3. For servers in this experiment we used an echo server and an http server on node0. As tests of node4-to-node0 connectivity in the above instructions we sometimes tried echo, sometimes http. Sometimes we even tried ping. When it came to stunnel in particular, we tried http. We found it went through successfully. However we never tried echo nor ping. If we'd added those extra tests would they have gone through or not? Are you aware of any arrangement that could broaden coverage to *all* traffic seeking to pass from node4 to node0 so that, independent of service or protocol, anything addressed between the two nodes would be successfully encrypted and tunneled through?