Advanced SSH Tunneling

by Bill Brassfield, Dev Ops Technical Consultant, Taos

First, a review of simple TCP SSH tunnels:

Forwarding a local TCP port to a remote TCP port:
(using the -L option)

Usage:
```
ssh -L local-interface-hostname-or-ip:local-port-number:remote-interface-hostname-
or-ip:remote-port-number tunneluser@remotehost.example.com
```

Example #1 (-L option):
```
ssh -L 127.0.0.1:2022:10.150.35.74:22 tunneluser@remotehost.example.com
```

What it does:

Opens up a TCP listener socket on port 2022 on the local loopback interface. This gets forwarded to TCP port 22 at the IP address 10.150.35.74, as seen from the remotehost. Note that the IP address 10.150.35.74 might be one of the network interfaces of the remotehost, but not necessarily -- it could be the IP address of yet another host which is accessible from the remotehost.

Why this command is useful:

Most likely, the host 10.150.35.74 is not directly accessible via SSH from the local host, so we are using remotehost.example.com as an "SSH jump host" to hop through it (via SSH tunnel) to get to 10.150.35.74. After the tunnel gets established, then it should be possible to SSH from localhost to 10.150.35.74 (in another shell) via the command:

```
ssh -p 2022 someuser@127.0.0.1
```

OR

```
ssh -p 2022 someuser@localhost
```

(since 127.0.0.1 is localhost)

Example #2 (-L option):
```
ssh -L 8080:localhost:80 tunneluser@remotehost.example.com
```

What it does:

Opens up a TCP listener socket on port 8080 on the local loopback interface. This gets forwarded to TCP port 80 on the local loopback interface of the remotehost (its HTTP webserver port).

NOTE: Notice that the local-interface-hostname-or-ip has been omitted. In this situation, the local TCP server listening on port 8080 will bind only to the local loopback interface (localhost, or 127.0.0.1). It can forced to bind to all configured network interfaces (all which have IP addresses) if the 8080 is preceded by a ":" or by a "*:" (blank or wildcard interface).

Why this command is useful:

An obvious question would be: Why not just point my web browser directly to remotehost.example.com? (http://remotehost.example.com/) There may be times when the web server port of remotehost may not be accessible from the localhost due to firewall rules (on the remotehost or on some network device in between). If opening up a firewall rule to allow HTTP traffic between localhost and remotehost is not an option, then setting up this SSH tunnel is a quick and easy way to make the web server on remotehost accessible via a web browser. Once the tunnel is established, simply point the web browser to this URL:

```
http://localhost:8080/
```

NOTE: We are using TCP port 8080 on localhost because it's an unprivileged port (> 1023), and any non-root user can open up the port for listening, as long as it isn't currently already in use. Secondly, there might already be a web server running on localhost (on TCP port 80), so the choice of port 8080 avoids this possible conflict.

Example #3 (-L option):
```
ssh -L 192.168.3.45:8080:web01.example.com:80 tunneluser@remotehost.example.com
```

What it does:

Opens up a TCP listener socket on port 8080 on the local network interface with IP address 192.168.3.45. This gets forwarded to TCP port 80 on yet another server (web01.example.com), which is reachable from remotehost.

Why this command is useful:

As in the previous example, the web server web01.example.com is probably not directly accessible from localhost (due to router and/or firewall rules blocking access). If it's not possible or practical to reconfigure the router and/or firewall rules to allow HTTP traffic directly between localhost and web01, then setting up this tunnel is a quick and easy way around the problem. Once the tunnel is established, point the web browser on localhost to the following URL:

```
http://192.168.3.45:8080/
```

NOTE: As in the previous example, we choose TCP port 8080 for the local port number because it's an unprivileged port (any non-root user can set up the tunnel), and it won't conflict with any web server that might be running on the localhost and listening on TCP port 80.

Example #4 (-L option):

```
ssh -L 192.168.1.120:80:192.168.1.120:80 tunneluser@remotehost.example.com
```

What it does:

Opens up a TCP listener socket on port 80 on the local network interface with IP address 192.168.1.120. This gets forwarded to TCP port 80 on yet another server which (by strange coincidence) has the same IP address of 192.168.1.120 but is actually on a different network. It's important to note that since we have chosen local TCP port 80 (which is a privileged port), this SSH tunnel must be created by the superuser (root), either from a root shell or via the "sudo" command. (Any attempt to create this tunnel by a non-root user will fail, due to the inability to bind to a listener port number less than 1023.)

Why this command is useful:

At first glance, this command looks useless and perhaps even ridiculous because the local IP and port number exactly match the remote IP and port number. Aren't we just going in circles and "building a bridge to nowhere"? Actually, no. In this scenario, there is a remote web server at IP address 192.168.1.120 which is not directly accessible from localhost, but is directly accessible from remotehost.

We want to point our browser at the remote web server and use its website (perhaps it's a JIRA issue-tracking server or a Confluence Wiki server). Why, then, do we not just use "localhost:8080" or "localhost:80"? The reason is that some web servers are configured to return redirects and/or web pages with absolute hard-coded IP addresses in them (instead of relative URLs). Any attempt to use the website by pointing the web browser to http://localhost:8080/ or to http://localhost/ would result in the web browser being sent to http://192.168.1.120/ for any follow-up web pages. In order to fix this problem, it's necessary to create a network interface on localhost with the same IP address (192.168.1.120), and bind the local end of the SSH tunnel to TCP port 80 on this interface. But what if localhost has only one physical network interface (eth0), and it's already in use for all of its connectivity to the outside world (including to remotehost)? We can create a "tap0" virtual network device and assign an IP address to it:

```
sudo tunctl -t tap0
```

```
sudo ifconfig tap0 192.168.1.120 netmask 255.255.255.0
```

```
sudo ssh -L 192.168.1.120:80:192.168.1.120:80 tunneluser@remotehost.example.com
```

After these commands have been issued (and assuming success on all of them), the web server at 192.168.1.120 can be browsed transparently by pointing the localhost web browser to the URL:

```
http://192.168.1.120/
```

NOTE: If necessary, a similar tunnel can also be created for TCP port 443 if HTTPS URLs are to be accessed.

A couple of possible "gotchas" which could make setup of these tunnels fail:

In /etc/ssh/sshd_config (on the local or the remote end), TCP port forwarding might be disabled. Normally (by default), it's enabled, but it's possible that a system administrator might be extra paranoid about the security risks of allowing tunneling and may have disabled it. Look for the "AllowTcpForwarding" directive in sshd_config and make sure it's set to either "yes" or "all" instead of "no". Make sure TCP forwarding is enabled on both localhost and remotehost.

Another sshd_config directive which could get in the way is the "GatewayPorts" directive. By default, it's set to "no", and this would not allow binding to any remotehost interface IP other than "localhost" or "127.0.0.1". If it's desired to make the tunnel forward to yet another server (a remote web server accessible from remotehost), then the "GatewayPorts" directive needs to be set to "yes" in /etc/ssh/sshd_config on remotehost.

Forwarding a remote TCP port to a local TCP port:
(using the -R option)

Usage:

```
ssh -R remote-interface-hostname-or-ip:remote-port-number:local-interface-hostname-
or-ip:local-port-number tunneluser@remotehost.example.com
```

Example #1 (-R option):

```
ssh -R localhost:2022:localhost:22 tunneluser@bastionhost.example.com
```

What it does:

Suppose a laptop computer sits behind a NAT'ed firewall/router in an office. The employee wants to leave the laptop on his desk overnight, but wants to be able to access it via SSH from home later in the evening. Simply issue the above command to log into the publicly-accessible (via SSH) bastionhost. A new TCP socket will begin listening on port 2022 on the bastionhost, and this will be forwarded back to TCP port 22 (the SSH port) of the laptop computer. Note that TCP port 2022 is bound ONLY to the "localhost" local loopback interface on the bastionhost, so only processes which are actually running on bastionhost will have access to this server port.

Why this command is useful:

Sometimes, it's necessary to leave a laptop (or desktop or server computer) at work, and continue working later in the evening from home to use it. If the laptop can't be made accessible via VPN (which is actually the preferred way to do it), but an SSH jump-host (bastionhost) is available, then setting up such a tunnel can make the computer accessible. In order to access the computer from home, issue the following commands:

```
ssh joeuser@bastionhost.example.com
```

```
ssh -p 2022 laptopuser@localhost
```

(NOTE: You are trusting the integrity of "bastionhost", that it hasn't been compromised. The second SSH command could potentially leak your laptop password to bastionhost, and if it has been compromised, then a hacker could easily SSH into the laptop.)

A better way:

```
ssh -L 2022:localhost:2022 joeuser@bastionhost.example.com
```

```
ssh -p 2022 laptopuser@localhost
```

(Here, even if bastionhost has been compromised, it has no access to the unencrypted login credentials for the laptop, since the second SSH command actually goes through two SSH tunnels to SSH directly into the laptop from the home computer.)

Example #2 (-R option):

```
ssh -R web99.example.com:80:localhost:80 root@web99.example.com
```

What it does:

Suppose you have a web server running on your laptop or desktop at the office, and for some reason you want to make it temporarily available to the world with via a server with a public IP address (web99). Assume that web99 isn't running its own web server, so TCP port 80 is available for this SSH tunnel to use. This tunnel command opens up a TCP listener on port 80 on the public IP address of web99 and forwards the connection back to TCP port 80 of the laptop (or desktop) computer at the office. From anywhere on the Internet (assuming that web99.example.com has a DNS entry pointing to a public IP address), the office computer's website can be reached via a web browser at this URL:

```
http://web99.example.com/
```

As long as the website uses relative URLs and never hard-codes specific IP addresses into absolute URLs (for links and/or redirects), the website should work just fine, as if it were hosted directly on web99.

Why this command is useful:

This command, like the previous one, allows what would normally be an unreachable (non-internet-routable) computing resource to be made available from anywhere on the Internet -- effectively "piercing the firewall" and reaching a specific server behind the NAT.

One should be aware, though, that commands such as these could be considered a security risk for the internal corporate network, and might even be a violation of IT security policy, so please be careful when considering the use of such reverse-tunneling commands. (They can be considered "back doors" for entry into the corporate network, bypassing the normal authentication mechanisms for connecting to the network from the outside.)

Possible "gotchas" that might make these SSH tunnels fail:

As mentioned for the tunnels based on the -L option, AllowTcpForwarding must be enabled in sshd_config (in this case, on the remote host). Also, if any TCP server ports on the remote end need to be bound to anything other than the local loopback interface, then GatewayPorts must also be enabled in sshd_config. If forwarding a privileged port on the remote host back to the local host, it will be necessary to log in as root on the remote host, since only root can open a privileged socket for listening.

Useful command-line options for the commands discussed above:

One problem with all of the SSH tunnel commands discussed above is that they all run in the foreground and use up a terminal session. This can contribute to the clutter on a user's desktop. Furthermore, if the SSH tunnels need to persist for a long time (days, weeks, or longer), then it's simply not practical to use foregrounded SSH shell sessions into the remote servers (the user who initiated the SSH tunnel would like to log out and go home at the end of the day).

One possible solution (a bit of a kludgy hack) is to run the SSH tunnel commands inside "screen" sessions, then detach from those screens. This effectively daemonizes these sessions until the user reattaches to the screen sessions. Running a screen session just to contain an SSH tunnel connection is a bit of excessive overhead, though. Also, it leaves a shell session (on the remote server) wide open, so anyone who reattaches to the screen session could gain unauthorized access to the shell.

A better solution is to use SSH's built-in **-f** and **-N** options. The -f option tells SSH to run in the background (after prompting the user for a password, if SSH keys aren't being used). The -N option tells SSH to not run any command on the remote end, to only establish the TCP-forwarding tunnel. By using these two command-line options, no unnecessary shell windows or screen sessions are needed because the SSH tunnel commands are effectively daemonized, and the user is returned to the local shell prompt immediately after issuing them. For quick and easy reference, here are the modified SSH commands:

```
ssh -f -N -L local-interface-hostname-or-ip:local-port-number:remote-interface-
hostname-or-ip:remote-port-number tunneluser@remotehost.example.com
```

```
ssh -f -N -L 127.0.0.1:2022:10.150.35.74:22 tunneluser@remotehost.example.com
```

```
ssh -f -N -L 8080:localhost:80 tunneluser@remotehost.example.com
```

```
ssh -f -N -L 192.168.3.45:8080:web01.example.com:80
tunneluser@remotehost.example.com
```

```
ssh -f -N -L 192.168.1.120:80:192.168.1.120:80 tunneluser@remotehost.example.com
```

```
sudo ssh -f -N -L 192.168.1.120:80:192.168.1.120:80
tunneluser@remotehost.example.com
```

```
ssh -f -N -R remote-interface-hostname-or-ip:remote-port-number:local-interface-
hostname-or-ip:local-port-number tunneluser@remotehost.example.com
```

```
ssh -f -N -R localhost:2022:localhost:22 tunneluser@bastionhost.example.com
```

```
ssh -f -N -L 2022:localhost:2022 joeuser@bastionhost.example.com
```

```
ssh -f -N -R web99.example.com:80:localhost:80 root@web99.example.com
```

NOTE: To shut down (destroy) the SSH tunnels created by these commands, it will be necessary to run a command like "ps auxww | grep ssh | grep -v grep" to find the process ID of the SSH tunnel command, then run "kill PID" where PID is the process ID number. (The default kill signal, SIGTERM, should be sufficient to shut down the tunnels; only in rare circumstances should it be necessary to use "kill -9".)

Tunneling for non-TCP network protocols:

All of the discussion above about TCP-forwarding SSH tunnels should be mostly review for veteran users of SSH; most of us have probably used the -L and -R tunneling options to point a web browser at a normally unreachable web server or to connect a database client to a MySQL or Oracle database. But what about other protocols like UDP or ICMP? The -L and -R tunneling options for SSH work only for forwarding TCP-based protocols. Furthermore, they tunnel only one TCP port on one end, to one TCP port on the other end. If an application requires several different TCP connections to remote resources on unreachable servers, then several SSH tunnels need to be created -- all with the correct source and destination port numbers -- before the application will work. And if the app needs to use UDP (perhaps DNS requests) or ICMP ping to communicate with these unreachable servers, then we would seem to be out of luck. Perhaps it is time to contact the network engineers and ask for their help? Install and configure OpenVPN on both ends?

Layer-2 and Layer-3 VPN tunnels over SSH:

Fortunately, SSH has some advanced VPN tunneling features which can solve most of these problems. There are two types of VPN tunnels to choose from, depending on the needs of the user and/or application. The first and easiest choice is a simple and lightweight Layer-3 VPN tunnel. This is good enough to pass network traffic of types TCP, UDP, and ICMP, so most of our most familiar application protocols such as DNS, NTP, TFTP, and ping will work just fine over these tunnels. If protocols such as DHCP are to be used, or if using a tunnel to bridge two remote networks together, then the more heavyweight Layer-2 VPN tunnel feature must be used.

In most cases, OpenVPN would be the preferred solution if a VPN tunnel is going to be needed for production or some other long-term usage. One disadvantage of using OpenVPN is that it's probably not installed on the local or remote systems by default, and it also can be a bit complex to configure properly. This is where the SSH VPN solution shines -- SSH is installed nearly everywhere (especially on Linux systems, regardless of the distribution), and SSH is a familiar and easy to use utility. On the downside, SSH-based VPN tunnels have quite a bit of overhead and therefore aren't as efficient as OpenVPN; overall, they're not really considered to be production-ready or long-term VPN solutions, but only a temporary means of establishing connectivity between remote systems.

Establishing a layer-3 SSH VPN using "tun" devices:

On the local server, issue the following command:

```
sudo ssh -f -w 0:0 root@remotehost.example.com true
```

(NOTE: You must be root on BOTH the local system and the remote system in order to create the "tun0" virtual network devices and connect them via SSH's tunneling protocol.)

After running the command, you should get your localhost shell prompt back. Now type in the command:

```
ifconfig tun0
```

You should see something like this:

```
tun0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          POINTOPOINT NOARP MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Note that the new "tun0" virtual Layer-3 network device has no MAC address, and no IP address/netmask. It's completely unconfigured. Next, configure it with something like this:

```
sudo ifconfig tun0 192.168.1.101 netmask 255.255.255.0
```

Now run "ifconfig tun0" again, and you will see this:

```
tun0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          inet addr:192.168.1.101  P-t-P:192.168.1.101  Mask:255.255.255.0
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Now we need to configure the remote end. From a root shell on the remote host, run essentially the same commands as shown above, except give the tun0 device a different IP address (192.168.1.102). After it's configured, it should look something like this:

```
tun0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          inet addr:192.168.1.102  P-t-P:192.168.1.102  Mask:255.255.255.0
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

Now, from the remote host, try "ping 192.168.1.101". You should see something like this:

```
PING 192.168.1.101 (192.168.1.101) 56(84) bytes of data.
64 bytes from 192.168.1.101: icmp_seq=1 ttl=64 time=59.4 ms
64 bytes from 192.168.1.101: icmp_seq=2 ttl=64 time=59.7 ms
64 bytes from 192.168.1.101: icmp_seq=3 ttl=64 time=59.7 ms
```

From the local host, try "ping 192.168.1.102". You should see something like this:

```
PING 192.168.1.102 (192.168.1.102) 56(84) bytes of data.
64 bytes from 192.168.1.102: icmp_seq=1 ttl=64 time=58.7 ms
64 bytes from 192.168.1.102: icmp_seq=2 ttl=64 time=60.8 ms
64 bytes from 192.168.1.102: icmp_seq=3 ttl=64 time=60.6 ms
```

These successful pings indicate that the Layer-3 VPN tunnel has been successfully created. This tunnel should be able to pass TCP and UDP traffic between these two IP addresses, from any source port number to any destination port number, in either direction, provided that a client application on one end tries to connect to a server application listening on the other end (and bound to the tun0 or the wildcard interface). For example, DNS lookups and NTP time syncs should be able to work over this tunnel.

Next, do a "ps auxww | grep ssh | grep -v grep" and look for the ssh process which created the Layer-3 VPN tunnel. Kill this process (using the default SIGTERM signal), and verify that the ssh process has terminated. Now run the "ifconfig" command on both the local and remote server. You will see that the "tun0" devices no longer show up. Next, run "cat /proc/net/dev" to list all known network interfaces on both ends, and you will see that the "tun0" devices not only are down, but they no longer exist. In fact, these devices existed for only as long as the SSH Layer-3 VPN tunnel process was running.

Potential "gotchas" that could prevent the Layer-3 VPN tunnel from working:

PermitTunnel must be enabled (set to "yes") in /etc/ssh/sshd_config on both the localhost and the remotehost. If it's disallowed, then SSH will fail to create and bind to the "tun0" device.

IPTables firewall rules on either end might be blocking traffic. Make sure that any traffic of interest to/from these two IP addresses is allowed in/out of the tun0 interfaces.

On the remotehost, PermitRootLogin in sshd_config must be set either to "yes" or to "without-password" (ssh key authentication). The "without-password" option is more secure than using "yes". If it's set to "no", then it will be impossible to log into the remote system as root. This, in turn, makes it impossible to create and bind to the tun0 virtual network device on the remote end.

Establishing a layer-2 SSH VPN using "tap" devices:

Sometimes, a Layer-3 VPN tunnel isn't good enough to provided the necessary connectivity. If protocols such as DHCP and spanning-tree-protocol (for network bridging) also need to work, then a Layer-2 VPN tunnel must be set up instead.

The first thing that needs to be done is to create "tap" virtual network devices on both ends (local and remote). If no tap devices exist yet on either end, then they can both be named "tap0". If one or more already exists, then use the first name (in tap1, tap2, tap3, etc.) which doesn't collide with that of an existing interface.

```
sudo tunctl -t tap0
```

(NOTE: If the "tunctl" command isn't available, then it can be installed via the appropriate RPM or Debian package (via yum or apt-get), or the "ip" command can be used instead:

```
sudo ip tuntap add dev tap0 mode tap
```

Verify that the "tap0" devices exist on the local and remote hosts:

On localhost:

```
 ifconfig tap0

 tap0      Link encap:Ethernet  HWaddr 2a:a5:1b:fa:4f:1c
           BROADCAST MULTICAST  MTU:1500  Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:500
           RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

On remotehost:

```
  ifconfig tap0

  tap0      Link encap:Ethernet  HWaddr F6:6D:F9:B3:01:A4
            BROADCAST MULTICAST  MTU:1500  Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:500
            RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

On the local host, configure the tap0 interface:

```
  sudo ifconfig tap0 192.168.1.101 netmask 255.255.255.0

  ifconfig tap0

  tap0      Link encap:Ethernet  HWaddr 2a:a5:1b:fa:4f:1c
            inet addr:192.168.1.101  Bcast:192.168.1.255  Mask:255.255.255.0
            UP BROADCAST MULTICAST  MTU:1500  Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:500
            RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

On the remote host, configure the tap0 interface:

```
  sudo ifconfig tap0 192.168.1.102 netmask 255.255.255.0

  ifconfig tap0

  tap0      Link encap:Ethernet  HWaddr F6:6D:F9:B3:01:A4
            inet addr:192.168.1.102  Bcast:192.168.1.255  Mask:255.255.255.0
            inet6 addr: fe80::f46d:f9ff:feb3:1a4/64 Scope:Link
            UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:21 overruns:0 carrier:0
            collisions:0 txqueuelen:500
            RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```

At this point, the "tap0" devices exist on both the local and the remote systems, but they are not yet connected because the SSH Layer-2 VPN tunnel isn't running yet.

On localhost:

```
  ethtool tap0

  Settings for tap0:
          Supported ports: [ ]
          Supported link modes:   Not reported
          Supported pause frame use: No
          Supports auto-negotiation: No
          Advertised link modes:  Not reported
          Advertised pause frame use: No
          Advertised auto-negotiation: No
          Speed: 10Mb/s
          Duplex: Full
          Port: Twisted Pair
          PHYAD: 0
          Transceiver: internal
          Auto-negotiation: off
          MDI-X: Unknown
          Current message level: 0xffffffa1 (-95)
                                 drv ifup tx_err tx_queued intr tx_done rx_status
pktdata hw wol 0xffff8000
          Link detected: no
```

On remotehost:

```
  ethtool tap0

  Settings for tap0:
          Supported ports: [ ]
          Supported link modes:
          Supports auto-negotiation: No
          Advertised link modes:  Not reported
          Advertised auto-negotiation: No
          Speed: 10Mb/s
          Duplex: Full
          Port: Twisted Pair
          PHYAD: 0
          Transceiver: internal
          Auto-negotiation: off
          Current message level: 0xffffffa1 (-95)
          Link detected: no
```

Now start the SSH Layer-2 VPN tunnel by issuing the following command on the localhost:

```
ssh -o Tunnel=ethernet -f -w 0:0 root@remotehost.example.com true
```

Run the "ethtool tap0" commands again on the local and remote hosts, and you should see "Link detected: yes" for both. This means that the tunnel is up and running successfully.

From the local host, run "
`ping 192.168.1.102`
":

```
PING 192.168.1.102 (192.168.1.102) 56(84) bytes of data.
64 bytes from 192.168.1.102: icmp_seq=1 ttl=64 time=85.9 ms
64 bytes from 192.168.1.102: icmp_seq=2 ttl=64 time=61.8 ms
64 bytes from 192.168.1.102: icmp_seq=3 ttl=64 time=61.0 ms
```

From the remote host, run "
`ping 192.168.1.101`
":

```
PING 192.168.1.101 (192.168.1.101) 56(84) bytes of data.
64 bytes from 192.168.1.101: icmp_seq=1 ttl=64 time=58.8 ms
64 bytes from 192.168.1.101: icmp_seq=2 ttl=64 time=61.9 ms
64 bytes from 192.168.1.101: icmp_seq=3 ttl=64 time=65.9 ms
```

So far, we see that the Layer-2 VPN tunnel using the two "tap0" devices is able to support ICMP ping, just like the Layer-3 VPN tunnel using the "tun0" devices. Additionally, it is able to support TCP and UDP connections between the two IP addresses (from any source port number to any destination port number, in either direction) -- again, just like the Layer-3 VPN tunnel. But additionally, we can take the functionally of the Layer-2 VPN tunnel a step further:

On the localhost:

```
 brctl addbr br-kick
 brctl addif br-kick eth1
 brctl addif br-kick tap0
 brctl stp br-kick on
 ifconfig eth1 0.0.0.0
 ifconfig tap0 0.0.0.0
 ifconfig br-kick 192.168.1.101 netmask 255.255.255.0

 brctl show
 ifconfig br-kick

 bridge name   bridge id               STP enabled     interfaces
 br-kick       8000.004268b5d723       yes             eth1
                                                       tap0

 br-kick   Link encap:Ethernet  HWaddr 00:42:68:b5:d7:23
           inet addr:192.168.1.101  Bcast:192.168.1.255  Mask:255.255.255.0
           UP BROADCAST MULTICAST  MTU:1500  Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:0
           RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

On the remotehost:

```
 brctl addbr br-kick
 brctl addif br-kick eth2
 brctl addif br-kick tap0
 brctl stp br-kick on
 ifconfig tap0 0.0.0.0
 ifconfig br-kick 192.168.1.102 netmask 255.255.255.0

 brctl show
 ifconfig br-kick

 bridge name   bridge id               STP enabled     interfaces
 br-kick       8000.00842a91f349       yes             eth2
                                                       tap0

 br-kick   Link encap:Ethernet  HWaddr 00:84:2a:91:f3:49
           inet addr:192.168.1.102  Bcast:192.168.1.255  Mask:255.255.255.0
           UP BROADCAST MULTICAST  MTU:1500  Metric:1
```

```
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

If localhost is running a DHCP server and was originally listening on the eth1 interface for network 192.168.1.0/24, then it should be re-configured to listen on the br-kick bridge interface. If all of the above is set up correctly (on both ends, only the br-kick bridge interfaces are set up with IP addresses and netmasks; all bridge member interfaces are up but unconfigured; the two tap0 interfaces are connected to each other via a running SSH Layer-2 VPN tunnel), then any other server that's plugged into the same Layer-2 broadcast domain as eth2 on the remotehost should be able to pick up a DHCP IP address from the DHCP server that's running on the localhost. (For example, a PXE-boot network installation of Linux should be possible.)

One additional option that we may want to use for all of the SSH tunnels discussed in this article is this one:

```
-o ServerAliveInterval=30
```

This tells SSH to send a "no-OP" command between the local and remote host every 30 seconds, to prevent the TCP connection from appearing idle for too long. Some network appliances (routers and/or firewalls) will automatically terminate stale TCP connections, based on idle time. It's quite common to see idle SSH shell sessions "bumped off" after sitting idle for only a couple of minutes. Using this option keeps the connection active, so it won't get flagged for termination. Note that if the connection does get broken (perhaps due to a rebooted network device), it won't be automatically reestablished. If it's necessary for a tunnel to stay up (and be automatically restarted if it goes down), then it will need to be carefully monitored by a well-written cron job.

Potential "gotchas" the could prevent the Layer-2 VPN tunnel from working:

The same "gotchas" that can prevent Layer-3 VPN tunnels from working also apply here. PermitTunnel must be enabled on both ends, and PermitRootLogin must be enabled (either "yes" or "without-password") on the remote end. IPTables firewall rules must not be blocking the network traffic which is expected to use the VPN tunnel. The "tap" devices on both ends must be properly configured (interfaces up, different IP addresses on the same network [same netmask], and the "-w" option of the SSH command for starting the tunnel must have the correct numbers on both sides of the colon in its argument (0:0 if it's local tap0 and remote tap0, but perhaps 1:2 if local tap1 and remote tap2). Using "ethtool" to view the "Link detected" status of the tap devices is very useful in debugging problems, as is the "ping" command.

One final note:

When it comes to establishing network connectivity between remote systems, ssh is extremely versatile. With its ability to provide Layer-3 and Layer-2 VPN tunneling (in addition to its much more well-known TCP-only tunneling), it can make remote networks hidden behind multiple layers of NAT and firewalls appear and behave as if they are local networks (plugged into the same physical switch). However, there will be noticeable latency and throughput limitations due to the extra overhead and possibly long distances between the local and remote networks. Doing a PXE-boot kickstart (network OS install) of Linux over an SSH Layer-2 VPN tunnel works, but it's quite a bit slower than doing it over a local wired connection. The same is true for any other high-bandwidth network activity. Using streaming audio or video over such a connection (including VoIP) will very likely reveal the performance limitations of these SSH VPN tunnels.

In a pinch, setting up these SSH-based VPN tunnels works until a more robust and permanent VPN solution is implemented. For Production use (or for any other long-term use even by a Dev or QA team), it's highly recommended to use something like OpenVPN or perhaps hardware-based VPN endpoints by Cisco or Juniper Networks. If an SSH-based VPN tunnel is going to be used for longer than a day or two, then it's highly recommended that it be monitored by a script that can restart the VPN tunnel if it goes down. For better performance, one might consider fine-tuning some of the tap and/or bridge interface configuration parameters such as the MTU, but that's beyond the scope of this article.