Advanced Network Configuration and Troubleshooting (2.205.2)

The candidate should be able to configure a network device to implement various network authentication schemes. This objective includes configuring a multi-homed network device, configuring a virtual private network and resolving networking and communication problems.

Key files, terms and utilities include:

/sbin/route
/bin/ifconfig
/bin/netstat
/bin/ping
/sbin/arp
/usr/sbin/tcpdump
/usr/sbin/lsof
/usr/bin/nc

Resources: Bird01, Drake00, FreeSWAN.

Virtual Private Network

What Is A VPN

VPN stands for Virtual Private Network. A VPN uses the Internet as its transport mechanism, while maintaining the security of the data on the VPN. What does a VPN do? It allows you to connect two or more remote networks securely over an insecure connection. The reason it is called *virtual* is that there is no physical connection between the two networks. Instead a non-dedicated *secure tunnel* is created through an insecure connection (like the Internet). This tunnel is generally encrypted and is only decrypted when it arrives at the destination host.

Why would you need such a network? In today's connected world, employees always need access to the data on the corporate network. Until recently this was done by dialing directly into the servers. The issue with this is that long-distance bills can be expensive as well as the cost of equipment and extra phone lines. By letting the employee connect via local internet access wherever he is, the costs are dramatically reduced.

Another reason for implementing a VPN is to integrate the LANs in several office or branches. By connecting the two networks, a user in Los Angeles can access network services in New York while still using their regular Internet connection.

VPN Types

There are many ways to implement a VPN. Many companies use proprietary software implementation while others use their routers because many routers have VPN support built in. These VPNs can be as simple as an SSH tunnel or very complicated. But all implementations create a virtual tunnel through an insecure network. Some VPN implementations include:

- IPSEC
- VPND
- SSH

- Many Cisco Routers

This book will only outline the implementations of an SSH tunnel and IPSEC.

SSH and PPP

The system described here is to implement a VPN using **ssh** and **ppp**. Basically **ssh** creates a tunnel connection which **pppd** can use to run TCP/IP traffic though. That's what makes up the VPN.

The real trick to getting **ssh** and **pppd** to play well together is the utility **pty-redir** written by Arpad Magosanyi that allows the redirection of standard in and standard out to a pseudo tty. This allows **pppd** to talk through **ssh** as if it were a serial line. On the server side, **pppd** is run as the users shell in the ssh-session, completing the link. After that, all you need to do is the routing.

The Server

The server is set up by creating an entry in `/etc/passwd` for a user with `/usr/bin/pppd` as the shell:

```
vpn_user:*:1004:101:VPN User,,,:/home/vpn-users:/usr/sbin/pppd
```

This line assumes that the user cannot login using a password. All authentication is done via **ssh**'s public key authentication system. This way, only those with keys can get in, and it's pretty much impossible to remember a binary key that's 530 characters long.

The Client

The link is created by running **pppd** through a pseudo terminal that is created by **pty-redir** and connected to **ssh**. This is done with something similar to the following sequence of commands:

```
/usr/sbin/pty-redir /usr/bin/ssh -t -e none -o 'Batchmode yes' \
           -c blowfish -i /root/.ssh/identity.vpn -l joe > /tmp/vpn-device
sleep 10

/usr/sbin/pppd `cat /tmp/vpn-device`
sleep 15

/sbin/route add -net 172.16.0.0 gw vpn-internal.mycompany.com \
                                                    netmask 255.240.0.0
/sbin/route add -net 192.168.0.0 gw vpn-internal.mycompany.com \
                                                    netmask 255.255.0.0
```

Simply, what this does is run **ssh**, redirecting its input and output to **pppd**. The options passed to **ssh** configure it to run without escape characters (`-e`), using the blowfish crypto algorithm (`-c`), using the identity file specified (`-i`), in terminal mode (`-t`), with the options *`'Batchmode yes'`* (`-o`). The **sleep** commands are used to space out the executions of the commands so that each can complete their startup before the next is run.

If the client is a standalone machine, that's all there is to it. If, on the other hand, you have more demanding routing needs (e.g., when the client is a firewall to a larger network) you will have to set up routes accordingly. On the server, this can be accomplished by running a cron job every minute that quietly sets back routes. It's not a big deal if the client isn't connected, **route** will just spit out an error (that can be sent to `/dev/null`.)

IPSEC

A more full-blown approach to VPN tunnels is the IPSEC protocol. IPSEC provides encryption and authentication services at the IP (Internet Protocol) level of the network protocol stack. IPSEC can be used on any machine which does IP networking. Dedicated IPSEC gateway machines can be installed wherever required to protect traffic. IPSEC can also run on routers, on firewall machines, on various application servers and on end-user desktop or laptop machines.

The basic idea of IPSEC is to provide security functions, authentication and encryption at the IP-level. This requires a higher-level protocol (IKE) to set things up for the IP-level services (ESP and AH).

Three protocols are used in an IPSEC implementation:

ESP, Encapsulating Security Payload

> Encrypts and/or authenticates data;

AH, Authentication Header

> Provides a packet authentication service;

IKE, Internet Key Exchange

> Negotiates connection parameters, including keys, for the other two.

The term IPSEC is slightly ambiguous. In some contexts, it includes all three of the above, but, in other contexts, it refers only to AH and ESP.

FreeS/WAN is the linux implementation of IPSEC. The FreeS/WAN implementation has three main parts:

- KLIPS (kernel IPSEC) implements AH, ESP and packet handling within the kernel
- Pluto (an IKE daemon) implements IKE, negotiating connections with other systems
- various scripts provide an administrator interface to the machinery

The config file contain three parts:

one or more connection specifications

> Each connection section starts with **`conn ident`**, where `ident` is an arbitrary name which is used to identify the connection.

connection defaults

> This section starts with **`conn %default`**. For each parameter in it, any section which does not have a parameter of the same name gets a copy of the one from the *`%default`* section. There may be multiple %default sections, but only one default may be supplied for any specific parameter name and all *`%default`* sections must precede all non-*`%default`* sections of that type.

the config section

The config section starts with **`config setup`** and contains information used when starting the software.

A sample configuration file is shown below:

```
# basic configuration
config setup
        interfaces="ipsec0=eth0"
        klipsdebug=none
        plutodebug=none
        plutoload=%search
        plutostart=%search
        # Close down old connection when new one using same ID shows up.
        uniqueids=yes

# defaults that apply to all connection descriptions
conn %default
        # How persistent to be in (re)keying negotiations (0 means very).
        keyingtries=0
        # How to authenticate gateways
        authby=rsasig

# VPN connection for head office and branch office
conn head-branch
        # identity we use in authentication exchanges
        leftid=@head.example.com
        leftrsasigkey=0x175cffc641f...
        left=45.66.9.170
        leftnexthop=45.66.11.254
        leftsubnet=192.168.11.0/24
        # right s.g., subnet behind it, and next hop to reach left
        rightid=@branch.example.com
        rightrsasigkey=0xfc641fd6d9a24...
        right=17.120.138.134
        rightnexthop=17.120.138.1
        rightsubnet=192.168.0.0/24
        # start automatically when ipsec is loaded
        auto=start
```

To avoid trivial editing of the configuration file for each system involved in a connection, specifications are written in terms of *`left`* and *`right`* participants, rather than in terms of local and remote. Which participant is considered *`left`* or *`right`* is arbitrary; IPSEC figures out on which it is running on based on internal information. This permits identical connection specifications to be used on both ends.

The *`left`*, *`leftsubnet`* and *`leftnexthop`* (and the *`right...`* counterparts) determine the layout of the connection:

```
         subnet 192.168.11.0/24            =leftsubnet
                 |
         interface 192.168.11.x
            left gateway machine
         interface 45.66.9.170             =left
                 |
         interface 45.66.11.254            =leftnexthop
              router
         interface we don't know
                 |
            INTERNET
                 |
         interface we don't know
              router
```

```
        interface 17.120.138.1            =rightnexthop
                |
        interface 17.120.138.134          =right
           right gateway machine
        interface 192.168.0.x
                |
        subnet 192.168.0.0/24             =rightsubnet
```

The *leftid* and *leftrsasigkey* are used in authenticating the left participant. The *leftrsasigkey* is the public key of the left participant (in the example above the RSA keys are shortened for easy display). The private key is stored in the `/etc/ipsec.secrets` file and should be kept secure.

The IKE-daemon of IPSEC is called **pluto**. It will authenticate and negotiate the secure tunnels. Once the connections is set up, the kernel implementation of IPSEC can route traffic through the tunnel.

`plutoload=%search` and `plutostart=%search` tell the **pluto** daemon to search the configuration file for *auto=* statements. All connections with *auto=add* will be loaded in the pluto database. Connections with *auto=start* will also be started.

For more information on FreeS/WAN and IPSEC, please see the references at the beginning of this chapter.

Troubleshooting

Network troubleshooting is a very broad subject with thousands of tools available. There are some very good books available on this topic, so we will limit our discussion to an overview of some of the tools which can be used to solve or detect network problems.

ifconfig

ifconfig checks the network interface configuration. Use this command to verify the user's configuration if the user's system has been recently configured or if the user's system cannot reach the remote host while other systems on the same network can.

When **ifconfig** is entered with an interface name and no other arguments, it displays the current values assigned to that interface. For example, checking interface *eth0* system gives this report:

```
$ ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:10:60:58:05:36
          inet addr:192.168.2.3  Bcast:192.168.2.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1398185 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1411351 errors:0 dropped:0 overruns:0 carrier:0
          collisions:829 txqueuelen:100
          RX bytes:903174205 (861.3 Mb)  TX bytes:201042106 (191.7 Mb)
          Interrupt:11 Base address:0xa000
```

The ifconfig command displays a few lines of output. The third line of the display shows the characteristics of the interface. Check for these characteristics:

UP

> The interface is enabled for use. If the interface is “down”, bring the interface “up” with the **ifconfig** command (e.g. **ifconfig eth0 up**).

RUNNING

> This interface is operational. If the interface is not “running”, the driver for this interface may not be properly installed.

The second line of **ifconfig** output shows the IP address, the subnet mask and the broadcast address. Check these three fields to make sure the network interface is properly configured.

Two common interface configuration problems are misconfigured subnet masks and incorrect IP addresses. A bad subnet mask is indicated when the host can reach other hosts on its local subnet and remote hosts on distant network, but cannot reach hosts on other local subnets. **ifconfig** quickly reveals if a bad subnet mask is set.

An incorrectly set IP address can be a subtle problem. If the network part of the address is incorrect, every **ping** will fail with the “no answer” error. In this case, using **ifconfig** will reveal the incorrect address. However, if the host part of the address is wrong, the problem can be more difficult to detect. A small system, such as a PC that only connects out to other systems and never accepts incoming connections, can run for a long time with the wrong address without its user noticing the problem. Additionally, the system that suffers the ill effects may not be the one that is misconfigured. It is possible for someone to accidentally use your IP address on his own system and for his mistake to cause your system intermittent communications problems. This type of configuration error cannot be discovered by **ifconfig**, because the error is on a remote host. The **arp** command is used for this type of problem.

ping

The basic format of the ping command on a Linux system is:

```
$ ping [-c count] host
```

host

> The hostname or IP address of the remote host being tested.

count

> The number of packets to be sent in the test. Use the count field and set the value low. Otherwise, the ping command will continue to send test packets until you interrupt it, usually by pressing CTRL-C (^C).

To check that `www.snow.nl` can be reached from your workstation, send four packets with the following command.

```
$ ping -c 4 www.snow.nl
PING home.NL.net (193.67.79.250): 56 data bytes
64 bytes from 193.67.79.250: icmp_seq=0 ttl=245 time=32.1 ms
64 bytes from 193.67.79.250: icmp_seq=1 ttl=245 time=32.1 ms
64 bytes from 193.67.79.250: icmp_seq=2 ttl=245 time=37.6 ms
64 bytes from 193.67.79.250: icmp_seq=3 ttl=245 time=34.1 ms

--- home.NL.net ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 32.1/33.9/37.6 ms
```

This test shows a good wide-area network link to `www.snow.nl` with no packet loss and fast response. A small packet loss, and a round-trip time an order of magnitude higher, would not be abnormal for a connection made across a wide-area network. The statistics displayed by the ping

command can indicate a low-level network problem. The key statistics are:

- The sequence in which the packets are arriving, as shown by the ICMP sequence number (icmp_seq) displayed for each packet;
- How long it takes a packet to make the round trip, displayed in milliseconds after the string time=;
- The percentage of packets lost, displayed in a summary line at the end of the ping output.

If the packet loss is high, the response time is very slow or packets are arriving out of order, there could be a network hardware problem. If you see these conditions when communicating over great distances on a wide area network, there is nothing to worry about. TCP/IP was designed to deal with unreliable networks, and some wide-area networks suffer from a high level of packet loss. But if these problems are seen on a local-area network, they indicate trouble.

On a local-network cable segment, the round-trip time should be near 0, there should be little or no packet loss and the packets should arrive in order. If these things are not true, there is a problem with the network hardware. On an Ethernet, the problem could be improper cable termination, a bad cable segment or a bad piece of “active” hardware, such as a hub, switch or transceiver.

The results of a simple ping test, even if the ping is successful, can help direct you to further testing toward the most likely causes of the problem. But other diagnostic tools are needed to examine the problem more closely and find the underlying cause.

route

To check the routing of a linux box, the **route** command is usually entered with no parameters or with `-n` to turn off ip-address to name resolution. For example **route -n** might show:

```
$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.11.0    0.0.0.0         255.255.255.0   U     0      0        0 eth0
145.66.8.0      0.0.0.0         255.255.252.0   U     0      0        0 eth1
0.0.0.0         145.66.11.254   0.0.0.0         UG    0      0        0 eth1
```

This host has two interfaces, one on subnet 192.168.11.0/24 the other on subnet 145.66.8.0/22. There is also a default route out on *`eth1`* to 145.66.11.254 (denoted by the `G` under “Flags” and a “Destination” and “Genmask” of 0.0.0.0).

To be able to troubleshoot this information you need to know what the routing should be, perhaps by saving the routing information when the system is known to work.

The two most common mistakes are:

- No network entry for an interface. When a network interface is configured a routing entry should be added that informs the kernel about the network that can be reached through the interface.
- No default route (or two default routes). There should be exactly one default route. Note that two default gateways can go undetected for a long time because the routing could “accidentally” use the proper gateway.

In general, if there is a routing problem, it is better to first locate the part of the network where the

problem originates, e.g. with **ping** or **traceroute** and then use **route** as part of the diagnostics.

traceroute

traceroute is a tool used to discover the links along a path. Path discovery is an essential step in diagnosing routing problems.

The **traceroute** is based on a clever use of the Time-To-Live (TTL) field in the IP packet's header. The TTL field is used to limit the life of a packet. When a router fails or is misconfigured, a routing loop or circular path may result. The TTL field prevents packets from remaining on a network indefinitely should such a routing loop occur. A packet's TTL field is decremented each time the packet crosses a router on its way through a network. When its value reaches 0, the packet is discarded rather than forwarded. When discarded, an ICMP TIME_EXCEEDED message is sent back to the packet's source to inform the source that the packet was discarded. By manipulating the TTL field of the original packet, the program **traceroute** uses information from these ICMP messages to discover paths through a network.

traceroute sends a series of UDP packets with the destination address of the device you want a path to. By default, **traceroute** sends sets of three packets to discover each hop. traceroute sets the TTL field in the first three packets to a value of 1 so that they are discarded by the first router on the path. When the ICMP TIME_EXCEEDED messages are returned by that router, **traceroute** records the source IP address of these ICMP messages. This is the IP address of the first hop on the route to the destination.

Next, three packets are sent with their TTL field set to 2. These will be discarded by the second router on the path. The ICMP messages returned by this router reveal the IP address of the second router on the path. The program proceeds in this manner until a set of packets finally has a TTL value large enough so that the packets reach their destination. Most implementations of **traceroute** default to trying only 30 hops before halting.

An example traceroute on linux looks like this:

```
$ traceroute vhost2.cistron.net
traceroute to vhost2.cistron.net (195.64.68.36), 30 hops max, 38 byte packets
 1  gateway.kabel.netvisit.nl (145.66.11.254)  56.013 ms  19.120 ms  12.642 ms
 2  145.66.3.45 (145.66.3.45)  138.138 ms  28.482 ms  28.788 ms
 3  asd-dc2-ias-ar10.nl.kpn.net (194.151.226.2)  102.338 ms  240.596 ms  253.462 m
 4  asd-dc2-ipc-dr03.nl.kpn.net (195.190.227.242)  95.325 ms  76.738 ms  97.651 m
 5  asd-dc2-ipc-cr01.nl.kpn.net (195.190.232.206)  61.378 ms  60.673 ms  75.867 m
 6  asd-sara-ipc-dr01.nl.kpn.net (195.190.233.74)  111.493 ms  96.631 ms  77.398 m
 7  asd-sara-ias-pr10.nl.kpn.net (195.190.227.6)  78.721 ms  95.029 ms  82.613 ms
 8  ams-icr-02.carrier1.net (212.4.192.60)  90.179 ms  80.634 ms  112.130 ms
 9  212.4.192.21 (212.4.192.21)  49.521 ms  80.354 ms  63.503 ms
10  212.4.194.42 (212.4.194.42)  94.528 ms  60.698 ms  103.550 ms
11  vhost2.cistron.net (195.64.68.36)  102.057 ms  62.515 ms  66.637 ms
```

Again, knowing what the route through your network should be, helps to localize the problem. Note that not all network problems can be detected with a **traceroute**, because of some complicating factors. First, the router at some hop may not return ICMP TIME_EXCEEDED messages. Second, some older routers may incorrectly forward packets even though the TTL is 0. A third possibility is that ICMP messages may be given low priority and may not be returned in a timely manner. Finally, beyond some point of the path, ICMP packets may be blocked.

The results of a **traceroute** is a great tool to narrow down the possible causes of a network problem.

arp and arpwatch

arp is used to manipulate the kernel's ARP cache. The primary options are clearing an address

mapping entry and manually setting one up. For debugging purposes, the **arp** program also allows a complete dump of the ARP cache.

If you know what the ARP address of a specific host should be, the dump may be used to determine a duplicate IP-address, but running **arpwatch** on a central system might prove more effective.

arpwatch keeps track of ethernet/IP address pairings. Changes are reported via syslog and e-mail.

arpwatch will report a “changed ethernet address” when a known IP address shows up with a new ethernet address. When the old ethernet address suddenly shows up again, a “flip flop” is reported.

Detection of duplicate IP addresses is very hard even with **arpwatch**. if, for example, a rogue host uses the IP address of the host running the **arpwatch** program or never communicates with it, a duplicate IP address will go unnoticed. Still, **arpwatch** is a very useful tool in detecting networking problems.

tcpdump

tcpdump is a program that enables network administrators to inspect in real-time every packet going through the network to which his or her computer is attached. This tool is typically used to monitor active connections or troubleshoot occasional network connectivity problems.

tcpdump can generate a lot of output, so it may be useful to limit the reported packets by specifying a port (e.g. `tcpdump port 80`). There are lots of other ways to specify which packets and what information we are interested in – see the manpage of **tcpdump** for more information.

nc

If the network between two hosts seems to be working, but the requested service is not, **nc** may be helpful. **nc**, or netcat, will create a TCP or UDP connection to the given host and port. Your standard in is then sent to the host and anything that comes back is sent to your standard out.

Some of the major features of netcat are:

- Outbound or inbound connections, TCP or UDP, to or from any ports
- Full DNS forward/reverse checking, with appropriate warnings
- Ability to use any local source port
- Ability to use any locally-configured network source address
- Built-in port-scanning capabilities, with randomizer
- Built-in loose source-routing capability
- Can read command line arguments from standard input
- Slow-send mode, one line every N seconds

- Hex dump of transmitted and received data
- Optional ability to let another program service establish connections
- Optional telnet-options responder

Because netcat does not make any assumptions about the protocol used across the link, it is better suited to debug connections than **telnet**.

Prev Up Next
Chapter 5. Networking (2.205) Home Chapter 6. Mail & News (2.206)