CHAPTER 9

Topic 109: Networking Fundamentals

Without a way to accurately identify how to reach devices, all network (and Internet) communication would simply collapse. My computer might be physically connected to yours (and to a million others), but expecting my e-mail to magically find its destination without a routable address is like throwing a bottle with a message inside into the ocean and expecting it to arrive on the kitchen table of a friend who lives ten thousand miles away. Within five minutes.

Fundamentals of Internet Protocols

Broadly speaking, modern networks rely on three conventions to solve the problem of addressing: transmission protocols (like TCP, UDP, and ICMP), network addressing (IPv4 and IPv6), and service ports.

Transmission Protocols

The Transmission Control Protocol (TCP) carries most web, e-mail, and ftp communication. It is TCP's packet verification feature that qualifies it for content that can't afford to arrive incomplete. The User Datagram Protocol (UDP) is a good choice for when verification isn't needed, as with streaming video and VOIP (Voice Over Internet Protocol), which can tolerate some dropped packets. UDP does provide checksums. The Internet Control Message Protocol (ICMP; part of the Internet layer of the Internet protocol suite, rather than the Transport layer) is used mostly for quick and dirty exchanges like ping.

Network Addressing

Every network-connected device must have its own unique IP (Internet Protocol) address. I'll discuss IPv6 addresses a bit later, but for now, let's work with IPv4.

D. Clinton, *Practical LPIC-1 Linux Certification Study Guide*,
DOI 10.1007/978-1-4842-2358-1_9

IPv4

An IPv4 address is made up of four numeric octets, each comprised of a number between 0 and 255, such as this:

```
192.168.0.101
```

Those four octets are divided into two parts: octets toward the left describe networks, and octets to the right describe individual nodes (each of which can be assigned to a single device). Put in slightly different terms, the network is the larger space within which local devices exist and communicate freely with each other. The nodes are those individual devices.

In one possible configuration of the above example, the first three octets (192.168.0) might have been set aside as the network, and the final octet (101) is the node address given to a particular device. Such a network could have as many as 256 devices (although at least three of those addresses—0, 255, and often 1—are reserved for network use).

This can be described either through a netmask, such as 255.255.255.0, or using the CIDR (Classless Inter-Domain Routing) convention, such as 192.168.0.0/24. The 24 in this case represents the network portion, made up of 24 bits, or three 8-bit octets (3*8=24).

However, the same address could actually be used for completely different network structures. Let's say, by way of an example, that our network will grow beyond 256 devices. We could reserve two octets for nodes rather than just one. In this case, only the first two octets would make up the network (subnet) address: 192.168, freeing the other two octets for nodes. This would make more than 65,000 (256*256) addresses available. Here's what the netmask of such an address would look like:

```
255.255.0.0
```

And here's how the same network would be represented using the CIDR format (remember: 2*8=16):

```
192.168.0.0/16
```

Let's go back to the original example (192.168.0.0/24). The third octet was, as you will remember, part of the network address. But you can use it to create multiple subnets. One subnet, allowing (around) 256 nodes, would be:

```
192.168.0
```

But you could create a second subnet that would allow a different set of (around) 256 nodes using this notation:

```
192.168.1
```

In fact, you could create more than 250 separate networks, each supporting more than 250 unique nodes, all the way up to:

```
192.168.254
```

Why would you want to do this? Because networking is about more than just connecting devices, it's also about managing and, sometimes, separating them. Perhaps your company has resources that need to be accessible to some people (the developers, perhaps) but not others (marketing). But marketing might need access to a whole different set of resources. Keeping them logically separated into their own subnets can be a super efficient way to do that.

One more point about subnetting: to make things just a bit more complicated, you can use addresses from a single octet for both networks and nodes. You could, for instance, reserve some of the third octet for networks and the rest for device nodes. It might look something like this in CIDR:

```
172.16.0.1/20
```

and would have a netmask of:

```
255.255.240.0
```

It takes a while to absorb these rules. As always, you'll learn quickest by playing with your own networks (I'll demonstrate the tools you can use for this later). In the meantime, by doing a Google search for "subnet calculator" you can find a number of terrific tools to help you visualize and design an infinite range of subnets.

Network Address Translation (NAT)

You may have noticed that my examples above were all either in the 192.168 or 172.16 address ranges. There's a reason for that (although the basic rules discussed will apply to all network addresses): these are within the address ranges reserved for local networks. Why do we need addresses that can be used only in local, private networks? Because if we didn't do that, we would have run out of network addresses many years ago.

The problem was that the Internet grew far larger than was ever imagined. The number of attached devices had grown into the billions (and now, with the growing Internet of Things, beyond even that). IPv4—by definition—can provide just over four billion theoretical addresses, and that's not nearly enough.

The brilliant solution adopted by Internet architects was to reserve certain address ranges for use **ONLY** in private networks that would communicate with the "outside" world by way of network translation at the router level. This way, you could have millions of devices behind a single physical or virtual router, each with its own privately routable address, but all together using only a single public address.

These are the three private NAT address ranges:

10.0.0.0	to	10.255.255.255
172.16.0.0	to	172.31.255.255
192.168.0.0	to	192.168.255.255

You should be aware that all IPv4 network addresses (not only NAT) fall into one of three classes:

Class:	First octet:
Class A	between 1 and 127
Class B	between 128 and 191
Class C	between 192 and 223

As you can see, each of the three NAT address ranges falls into a different network class. At the same time, I should mention that network class rules are no longer always strictly observed.

IPv6

The IPv6 protocol was a different solution to the problem of limited numbers of available addresses. Largely because of the success of NAT, there's been little pressure to widely adopt IPv6, so you won't see all that much of it yet. But its time will definitely come, and you should be familiar with how it works.

IPv6 addresses are 128-bit addresses and are made up of eight hexadecimal numbers separated by colons.

■ **Note** Hexadecimal numbers (sometimes called base 16 or hex) are simply numbers that use 0-9 (to represent the numbers 0-9), and the first six letters of the alphabet (a-f) representing the numbers 10-15.

This is what an IPv6 address might look like:

```
fd60:0:0:0:240:f8cf:fd51:67cf
```

For those addresses (like the one above) with more than one adjacent field equaling zero, you can also write it with compressed fields replaced by double colons:

```
fd60::240:f8cf:fd51:67cf
```

Just as discussed with IPv4, IPv6 addresses are divided into two parts: the network section (those fields to the left) and the address section (to the right). IPv6 notation, much like CIDR notation, distinguish between network and address fields using an /n value. Given that IPv6 addresses are 128-bit addresses, an address whose four leftmost fields represent networks would be /64, whereas an actual device would be designated as /128.

Service Ports

Even though every network-connected device has its own unique address, because a single server can offer multiple services, incoming traffic will also need to know which service port it wants. By accepted convention, all ports between 1 and 65535 are divided into three types:

1 to 1023	Well-known ports
1024 to 49151	ICANN registered ports (reserved for specific commercial protocols)
49152 to 65535	Dynamic ports (available to anyone for ad hoc use)

■ **Note** We should perhaps pause every now and again to appreciate the many conventions that "rule" the information technology world. Without accepted conventions, there really could be no Internet, or even much of an IT industry. It's especially noteworthy that many of our most important protocols were created through the hard work of very bright people acting as unpaid volunteers.

Table 9-1 lists some of the more common well-known ports with which you should be familiar, both for the LPIC exam and for daily your work as a Linux admin.

Table 9-1. *Common "Well-Known" Network Ports and the Services for Which They're Used*

Port	Uses
21	FTP data control
22	SSH (Secure Shell)
23	Telnet (useful, but not secure)
25	SMTP (Simple Mail Transfer Protocol)
53	DNS (Domain Name System)
80	HTTP (Hypertext Transfer Protocol)
110	POP3
123	NTP (Network Time Protocol)
139	NetBIOS
143	IMAP (Internet Message Access Protocol)
161	SNMP (Simple Network Management Protocol)
162	snmptrap # Traps for SNMP
389	LDAP (Lightweight Directory Access Protocol)
443	HTTPS (Hypertext Transfer Protocol over SSL)
465	URL Rendesvous Directory (Cisco)
514	(UDP) syslog
514	(TCP) cmd (no passwords)
636	LDAP over SSL
993	IMAPS (Internet Message Access Protocol over SSL)
995	POP3 over TLS/SSL

You can see a much more complete and up-to-date list of the well-known and ICANN (Internet Corporation for Assigned Names and Numbers) registered ports in the /etc/services file:

```
less /etc/services
```

Directing a request through a specific service port will often require that you add the port number to the target address. Thus, an HTTP request might look like this:

```
192.168.0.146:80
```

And a request for a service on an ad hoc port might use:

```
192.168.0.146:60123
```

Basic Network Configuration

Now that you are hopefully comfortable with the general principles of networking, let's turn our attention to the practical task of setting up and maintaining network connectivity.

First, I should point out that things are changing in Linuxland: the much-loved ifconfig family of commands from the net-tools package is being deprecated in favor of ip. Net-tools is still installed by default on many distributions, and it can always be installed manually, but the world is—slowly—moving toward ip. The LPIC exam, in its current form, requires knowledge of both systems, so I'll demonstrate them side by side.

You can run ifconfig on its own to see a list of all your recognized network devices and their statuses:

```
$ ifconfig
eth0       Link encap:Ethernet  HWaddr 74:d4:35:5d:4c:a5
      inet    addr:192.168.0.105
Bcast:192.168.0.255
Mask:255.255.255.0
      inet6 addr: fe80::76d4:35ff:fe5d:4ca5/64 Scope:Link
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
      RX packets:194338 errors:0 dropped:0 overruns:0 frame:0
      TX packets:136839 errors:0 dropped:0 overruns:0 carrier:0
       collisions:0 txqueuelen:1000
      RX bytes:191307800 (191.3 MB)    TX bytes:18732326 (18.7 MB)
```

In the case of this partial output, ifconfig shows us that the eth0 NIC has been given the DHCP NAT address of 192.168.0.105 and an IPv6 address. It also displays various other indicators, including the download and upload statistics since the last boot.

Using ip addr list produces a similar output, but with less scope:

```
ip addr list
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP>
mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 74:d4:35:5d:4c:a5 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.102/24 brd 192.168.0.255 scope global eth0
            valid_lft forever preferred_lft forever
         inet6 fe80::76d4:35ff:fe5d:4ca5/64 scope link
            valid_lft forever preferred_lft forever
```

If you're feeling lazy, you can accomplish the same thing using "ip a l" or even just "ip a".

If you don't see an interface that you thought should have been there, it might simply not have been loaded. You can help it along with this (assuming that its name is eth1):

```
sudo ifup eth1
```

In the brave new world of ip, this is how you would do the same thing:

```
sudo ip link set dev eth1 up
```

Let's parse this command: ip will use the set command against the device (dev) whose type is link that's identified as eth1, telling Linux to bring the device up. As you can see, ip syntax is a bit more like human speech.

You can bring a device down using either:

```
sudo ip link set dev eth1 down
```

or:

```
sudo ifdown eth1
```

I should add that, unlike ip link set, ifup and ifdown can also be used to configure (or deconfigure) interfaces.

You will sometimes have to configure an interface manually, which can be done from the command line. This example does it the ip way:

```
sudo ip a add 192.168.1.150/255.255.255.0 dev eth1
```

Here "ip a add" was used to tell the system that you're adding an interface and applying it to the known device, eth1. You assign this interface the IP address of 192.168.1.150 (making sure that it fits with the subnet architecture and will be able to connect to the router), using a netmask of 255.255.255.0. This can also be done by editing the configuration files.

On Fedora, you'll want to work with the appropriate file in the /etc/sysconfig/network-scripts directory. Here's a possible example:

```
nano /etc/sysconfig/network-scripts/ifcfg-enp2s0f0
```

On Debian/Ubuntu machines, it's the interfaces file that you're after:

```
nano /etc/network/interfaces
```

It's more common for interfaces to get their IP addresses automatically from a DHCP (Dynamic Host Configuration Protocol) server. This will often occur behind the scenes during system boot. If, for some reason, it didn't work for your interface, or if it's an interface you just added, you can send a request for a DHCP address using:

```
sudo dhclient eth1
```

For an interface to gain access to the larger network (and to the Internet beyond), it will need a route to the outside world, which will generally go through a router (hence the name). You can view your current route tables using:

```
route
```

If you don't yet have a working route and you do know the IP address of your router, you can add a route, using either:

```
sudo route add default gw 192.168.1.1
```

or:

```
sudo ip route add default via 192.168.1.1
```

By the way, an improperly set route table is a very common cause of network problems. If you've recently updated the address of your router or other gateway device and then suffer some connectivity problem, make sure your route table matches the real-world router.

Basic Network Troubleshooting

So nothing's working. Well, the workstations are all humming away happily, but your users aren't able to download their important productivity documents (and it's not like YouTube is down or anything). Your boss doesn't look very pleased about this and you want to know what to do first.

This is Linux, right? So you open a terminal. As you can see from Figure 9-1, I'll start from the inside and work out to troubleshoot.

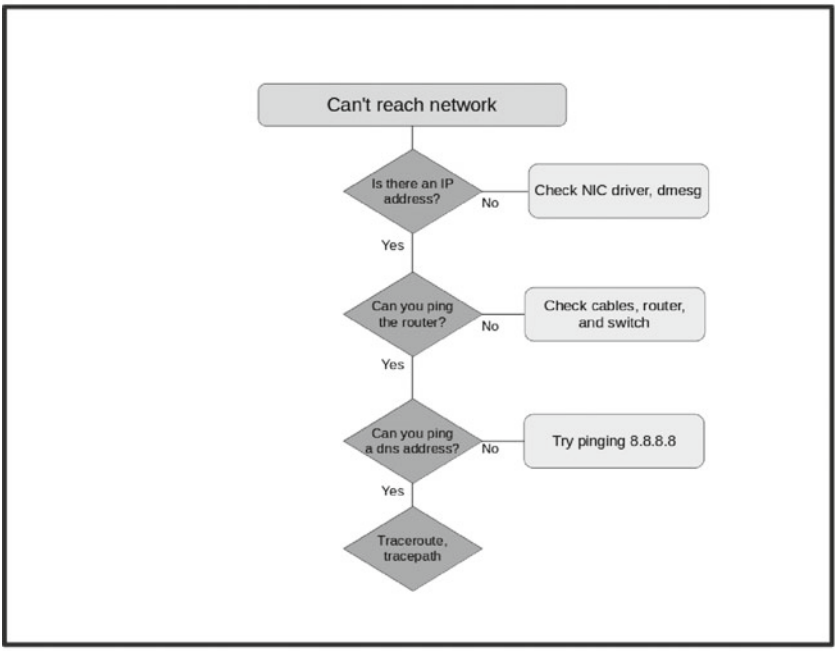

Figure 9-1. *Sample networking troubleshooting flowchart*

Check whether your computer's external interface (usually designated as either eth0 or em0) has an IP address:

```
ifconfig
```

or:

```
ip a
```

If it does, that probably means you've been successfully connected to your DHCP server, so the problem must lie farther afield. If you don't have an IP address, you might want to use dmesg to confirm that the network interface itself was picked up by your system:

```
dmesg | grep eth0
```

If there's no match for eth0 (or eth1, em0, or whatever you suspect might be the designation for your interface), then your kernel-level driver might have crashed (see Chapter 8 on kernel modules), or you might have a hardware problem of some sort.

You can shut down the machine and open the case to confirm that the network card (assuming that it isn't integrated with the motherboard) is properly seated in its slot. Remember to properly ground yourself. If that appears to be fine and you've got a spare card, you can install that to see if Linux recognizes it when you boot up again.

In any case, assuming you get through that stage without discovering what's causing your trouble, check to see if you can ping your router:

```
ping 192.168.0.1
```

Remember: your router address will usually be the same as your DHCP address, but with a 1 in the final field, rather than whatever number you had. Ping, by the way, uses the ICMP transmission protocol to send lots of very small data packets to the specified address, requesting that the host echoes the packets back. If ping was successful, you will be shown something like this:

```
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=57 time=26.2
64 bytes from 192.168.0.1: icmp_seq=2 ttl=57 time=25.9
64 bytes from 192.168.0.1: icmp_seq=3 ttl=57 time=26.7
64 bytes from 192.168.0.1: icmp_seq=4 ttl=57 time=26.9
```

■ **Note** Don't forget that, as the industry moves to IPv6, more and more purpose-built tools will appear to accommodate that change. Ping6 is one of those.

If that works, then the problem is clearly not between your computer and the router. If that doesn't work, then you should check your cabling and switches at all ends. You can also try rebooting your router and switch.

■ **Note** It sometimes feels like 95% of IT problems can be solved by rebooting. Most of the remaining 5% can be taken care of by keeping users away from IT resources, although that may sometimes lead to long-term productivity complications.

Now let's assume that you can successfully ping your router. The problem must be a bit farther out. Try pinging the DNS name for a known web site:

```
ping google.com
```

If that fails, there might be a problem with your Internet provider (which will require a friendly phone call, assuming your phone still works). But it's also possible that your DNS address translation isn't working. To find out, ping the IP address of an external service that you know should work. My favorite for these times is Google's DNS server, because the IP is just so easy to remember:

```
ping 8.8.8.8
```

If that works (even though *ping google.com* did not), then you know it's a DNS issue, which is discussed in the last section of this chapter.

If it doesn't work, it might be your Internet provider's fault, but it can still be helpful to narrow down the location of the blockage. Traceroute (or its IPv6 cousin, traceroute6) can help fill in some of the gaps. Running traceroute against a known address will show you each step your packets took along the route to a destination, including the one right before things failed:

```
traceroute 8.8.8.8
```

Tracepath, by the way, delivers much the same function as traceroute. And both traceroute and tracepath have their IPv6 equivalents: traceroute6 and tracepath6.

If the network problem is inbound, rather than outbound (in other words: people can't access resources on your network), then you'll need a different set of tools. First, you should confirm that all necessary ports are open. So, for instance, if you're running a web server, you'll probably need to open port 80 and, perhaps, 443.

You can use netstat to display all the ports and sockets that are currently listening on your system:

```
netstat -l | grep http
```

Using grep http will, of course, help narrow down your search.

From a computer on an external network you can also use netcat to poke at your network to see what's open. This example will test port 80 (assuming that "your network" is bootstrap-it.com):

```
nc -z -v bootstrap-it.com 80
```

In general, by the way, netcat is an excellent tool for testing your security.

Configure Client Side DNS

As mentioned earlier, all network devices have unique IP addresses. Since, however, people find it a lot easer to work with and remember more human-readable addresses (like bootstrap-it.com), DNS (Domain Name System) servers will translate back and forth between IPs and URLs. When I type *google.com* into the URL bar of a browser, there's a DNS server somewhere that's busy converting that into the correct IP address and sending off my request.

For this to work, you will need to designate a DNS server that can handle translation requests. This can be done on Debian/Ubuntu machines from the file:

```
/etc/network/interfaces
```

and, on Red Hat, using if-up scripts.

Here's what an interfaces file with DNS settings might look like:

```
auto eth0
iface eth0 inet static
    address 10.0.0.23
    netmask 255.255.255.0
    gateway 10.0.0.1
    dns-nameservers 208.67.222.222 208.67.220.220
    dns-search example.com
```

The nameservers being used here are those provided by the OpenDNS service.

The dns-search line tells the system that any searches launched without fully qualified domain names (FQDNs) will be appended to example.com.

Let me explain what that means: if I were to enter just the word "documents" into the URL bar of a browser, the request would normally fail. That's because there is no domain that matches *documents* (and it's a badly formed address in any case). However, with my dns-search value set, the local DNS server would try to find a resource somewhere within the local network called example.com/documents and dutifully fetch that for me.

Often, however, if your machine is a DHCP client, it will take the DNS settings) from its DHCP server. You might see a reference to this in the /etc/resolv.conf file (which, these days, is really nothing more than an autogenerated symlink):

```
# Generated by NetworkManager
search d-linkrouter
nameserver 192.168.0.1
```

The hostname of your own computer can also be used as an alternate to its IP address. You can update your host name by editing both the /etc/hosts and /etc/hostname files.

The hosts file can also be used to create a local alias. Let's say that, for some reason, you need to type commands in the terminal involving long URLs:

```
wget amazon.com
```

Okay. So amazon.com is not a particularly long URL, but you understand what I mean. If you'd like to create a shortcut, add this line to your hosts file:

```
54.239.25.200    www.amazon.com   a
```

From now on, whenever you need to access amazon.com from the command line, typing just the letter "a" will get it done:

```
wget a
```

The host and dig tools can be run against either domain names or IP addresses to return DNS information. Both can be useful for troubleshooting DNS problems:

```
$ host bootstrap-it.com
```

The order your system uses when resolving hostnames to IP addresses is determined by the hosts line in the /etc/nsswitch.conf file. Here's an example:

```
hosts:  files myhostname mdns4_minimal [NOTFOUND=return]
dns mdns4
```

You can edit the order the system) uses by simply changing the sequence of this line.

Now Try This

If you can get your hands on an unused wireless router, plug it in to your PC via a network cable and log in to the interface (it usually works by pointing your browser to 192.168.0.1, you can check the router case for login details). Note the current LAN settings (and, if you're nervous, how to reset it to its factory settings).

Now change the network subnet mask (which will, most likely, be 255.255.255.0) to 255.255.0.0, and create a network in a new subnet. You might want to use a range that's something like 192.168.1.1 to 192.168.1.253. Boot a laptop or a smartphone and log it in as a DHCP client on the new network. Make sure the IP address now used by your device is within the range you set.

Warning: be prepared to do a lot of rebooting and reconfiguring until you get it right. Remember: It's not frustrating, it's fun!

Test Yourself

1. The ping travels using the ____ protocol:
   a. TCP
   b. ICMP
   c. CIDR
   d. UDP
2. A subnet with a CIDR of 172.16.0.1/18 would have which netmask?
   a. 255.255.240.0
   b. 255.255.255.0

CHAPTER 10

Topic 110: Security

Creating a reasonably secure compute environment requires elements from just about every area of system administration. Perhaps that's why the LPI put security at the very end of their exam objectives, because you will, in fact, need all your skills to make this work.

I used the term "reasonably secure" because, when it comes to security, anyone who thinks his resources are 100% safe is fooling himself. An old friend who had worked for a national foreign service once told me that every single one of that country's overseas embassies was provided with a government-issue hammer for use in the event they were overrun. The purpose of the hammer? To physically destroy every hard drive in the building. There really is no better solution (and even that one is imperfect).

The bottom line: when it comes to IT security, there's never enough that can be done and you can never completely relax. Let's get started.

System Security

As discussed previously, user passwords are usually among the weakest links in your system. In Chapter 7, I explained how you can use the command chage to force your users to update their passwords from time to time. While it's not required by the LPIC-1 exam, you can also use PAM (Pluggable Authentication Module) to enforce password complexity. PAM is controlled through the /etc/pam.d/system-auth file (on Red Hat) or the /etc/pam.d/common- password (on Debian systems). Editing the "password required" line to read something like this:

```
password required pam_cracklib.so minlen=12 lcredit=1
ucredit=1 dcredit=2 ocredit=1
```

can make a big difference. This example will force users to create passwords whose minimum length is 12 characters, and where "credit" is given for the presence of at least one character that's in lowercase, one in uppercase, two digits, and one "other" (i.e., nonalphanumeric).

I also discussed how important it is to use a root or admin account as seldom as possible. Administrative powers should be given to only those users who absolutely need it, and even then, they should use those powers only through sudo.

D. Clinton, *Practical LPIC-1 Linux Certification Study Guide*,
DOI 10.1007/978-1-4842-2358-1_10

You can add a user to the sudo group (thereby giving him the right to admin powers for single commands) through chmod:

```
sudo usermod -aG sudo steve
```

This will add a user named Steve to the sudo group. Once you've done that, you can view the /etc/group file and Steve's name should be among those on the sudo line.

```
cat /etc/group | grep sudo
```

You can edit the way that sudo works on your system through the /etc/sudoers file, but you must use the visudo command rather than trying to edit the file directly. If you view the sudoers file (which itself, for obvious reasons, requires sudo), you will notice that there are separate lines defining the privileges given to members of the root, admin, and sudo groups. This allows you to very finely tune the powers you give to each of your administrative users. An example of this would be:

```
# User privilege specification
root    ALL=(ALL:ALL) ALL
# Members of the admin group may gain root privileges
%admin ALL=(ALL) ALL
# Allow members of group sudo to execute any command
%sudo  ALL=(ALL:ALL) ALL
```

So, to review, you should never, ever, EVER log in to a system as root or with persistent admin powers. And that's perfectly true, except where it isn't. There will be times when you have no choice but to start up an admin shell. To do that, type:

```
sudo su
```

and enter your password. Note that, in most cases, your shell prompt will now look something like this:

```
root@newpc:/home#
```

You should also keep track of the users who are logging in to your system. If, for instance, most of the gang where you work are out of the office by five each afternoon, then you should expect there won't be too many of them still logged in a half an hour later. And even if one or two might have left their workstations running (hopefully with a password-protected screensaver), you definitely won't be seeing too much activity.

That's why a quick review of user log ins can be useful. Using "w" will tell you when a user logged in, what system resources he's using, and, most importantly, what process he's running at the moment. Here is an example:

```
$ w
12:48:00 up  4:49,  3 users,  load average: 0.02, 0.18, 0.30
USER     TTY      FROM              LOGIN@   IDLE   JCPU
PCPU WHAT
dbclinto :0       :0                08:00    ?xdm?  32:40
0.22s init --user
dbclinto pts/1    :0.0              09:26   21.00s  0.28s
0.16s ssh dbclinton@1
dbclinto pts/4    :0.0              12:47    0.00s  0.06s
0.01s w
```

This output shows you that, besides my own bootup log in from 8:00 this morning, I've also got two shell sessions running: one, an ssh session into the Fedora laptop right next to me (it sure beats having to turn my chair around to actually use its keyboard), and the other, the shell from which I ran w.

Doesn't look like anyone else is around, and that's what I want to see. I need to be able to account for every single session that's listed.

Running who and last will also list log ins. Last can be particularly useful, as it lists all of the log ins since the beginning of the current month. Adding -d will also show you the origin host of each log in to give you an idea of where they've been coming from:

```
last -d
```

Monitoring files using lsof (LiSt Open Files) can also be a powerful security tool, especially since, in Linux, everything (even a directory) is a file.

You can list all processes (and their users) that have opened a specific file:

```
lsof /var/log/syslog
```

By adding +D, you can list all open files within a directory hierarchy:

```
lsof +D /var/log/
```

Using -u will narrow down your search to only those files opened by the specified user:

```
lsof -u steve
```

But suppose you're Steve and you know (or at least you hope) you're reliable, but you'd like to check into all those other suspicious characters you've seen around. The use of the caret symbol (^) will display everyone **EXCEPT** you:

```
lsof -u ^steve
```

You can use lsof along with kill to close all files opened by a specific user:

```
kill -9 `lsof -t -u steve`
```

You can even use lsof (with -i) to list all open network connections:

```
lsof -i
```

Consider incorporating some of those lsof tools into a script to automatically monitor your system activity.

Much of the same functionality of lsof can also be found in fuser. Fuser, too, will let you zero in on specific files:

```
fuser /var/log/syslog
```

It must be added, however, that fuser can get much more personal about it. Using:

```
fuser -km /home
```

will kill all processes accessing the filesystem /home. Be warned: by kill, it means kill.

By displaying which users and processes are accessing the http port (80), this next example will tell you if there's any unauthorized activity involving your web server:

```
sudo fuser -v -n tcp 80
```

In Chapter 9 you used netcat and netstat to search for open network ports. As mentioned there, from a security perspective, you should always be very interested in making sure there are no unnecessary ports open on the sites you manage. There's one more tool that covers some of the same ground: nmap. Running nmap against local or Internet-based addresses will display all open ports:

```
nmap bootstrap-it.com
```

Using nmap with the -sU flag will perform a UDP scan:

```
sudo nmap -sU 10.0.2.143
```

You can restrict an nmap scan to a specific port:

```
nmap -p 80 bootstrap-it.com
```

or, if you're trying to monitor a larger number of network resources, you can scan a range of addresses, with built-in exclusions:

```
nmap 192.168.2.0/24 --exclude 192.168.2.3
```

Besides looking for unusual things that are going on right now, you should also keep an eye open for vulnerabilities that could be exploited in the future. Files with suid permissions comprise one class of files that can potentially cause trouble.

As mentioned previously, a file with the suid bit can be used by any user AS THOUGH they shared the file's full admin rights. Sometimes, as with the /usr/bin/passwd binary, this is necessary. However, if you notice the suid in unexpected places, you should consider taking a closer look.

You can easily check your whole filesystem for suid using:

```
sudo find / -type f -perm -u=s -ls
```

Nice. The problem is that this will probably return a rather long list of files. How are you to know if there's actually a problem? One solution is to take the stream produced by using find and filter it for entries that are also writable by others, have the sgid bit set, or are unowned by a valid package or user. That will narrow down your search to combinations that definitely deserve more attention.

You can search for files with guid (group) permissions using:

```
sudo find / -type f -perm -g=s -ls
```

And this will find ownerless files:

```
find / -xdev \( -nouser -o -nogroup \) -print
```

You can set limits on the system resources available to specified users or even groups using ulimit. Try reviewing your own default limits by running:

```
ulimit -a
```

Notice the categories that can be controlled, including individual file size, the number of open files, and the maximum number of user processes. Maintaining the right balance of limits can help prevent the abuse of account privileges without unnecessarily restricting your users' legitimate activities. You can edit user and group limits in the /etc/security/limits.conf file. Here are some of the sample settings that illustrate the way the file works:

```
#root          hard    core            100000
#@student      hard    nproc           20
```

```
#@faculty      soft    nproc           20
#@faculty      hard    nproc           50
#@student      -       maxlogins       4
```

You should also keep an eye on active processes using ps:

```
ps aux
```

Of course, that will produce way too much information to be useful. Adding our old friend grep into the mix should help narrow things down:

```
ps aux | grep apache2
```

Host Security

Besides controlling the behavior of local users, it's also vital to be able to limit the things that people coming from beyond your local system can do. At its simplest, that might mean preventing nonroot log ins. Creating a readable file called nologin in the /etc directory will do just that:

```
sudo touch /etc/nologin
```

You can leave the file empty or include a message you'd like users to see that explains why they're currently locked out of the system.

Once you do decide to allow remote log ins, you can, in association with the init.d run-level control system I discussed in Chapter 1, closely control what your guests can do through the inetd (or, on newer distributions, xinetd) system. Inetd is known as a super-server because its job is to listen for requests on all ports listed in the config file and then, when appropriate, start and stop requested services.

The original goal was to save system resources by having one single running server activate services only when they were actually needed, and then shut them down when they're done. But, besides the built-in security benefits of turning off unused services, inetd also provided an added advantage through the ability to apply access control.

A typical /etc/xinetd.conf configuration file will include entries like these:

```
defaults
{
   instances            = 60
   log_type             = SYSLOG        authpriv
   log_on_success       = HOST PID
```

```
    log_on_failure       = HOST
    cps                  = 25 30
}
includedir /etc/xinetd.d
```

Note the includedir /etc/xinetd.d line, which points to the /etc/xinetd.d directory, which itself contains individual files for each service that will be controlled by xinetd. Here's an example of an xinetd file:

```
# default: off
# description: An RFC 863 discard server.
# This is the tcp version.
service discard
{
        disable         = yes
        type            = INTERNAL
        id              = discard-stream socket_type = stream
        protocol        = tcp
        user            = root
        wait            = no
}
# This is the udp version. service discard
{
        disable         = yes
        type            = INTERNAL
        id              = discard-dgram
        socket_type     = dgram
        protocol        = udp
        user            = root
        wait            = yes
}
```

The main value you should be aware of is disable. The current setting for both the TCP and UDP versions is "yes," which means that remote requests for the RFC 863 discard server will be refused. To tell xinetd to accept such requests, the disable value should be changed to "no." Discard, by the way, is roughly the equivalent of the /dev/null directory, a convenient place to dump debugging or testing data that will be immediately destroyed.

Besides discard and the few others you might see in a clean install of Linux, xinetd is also often used to control services like ftp, pop3, rsync, smtp, and telnet.

Once you've enabled a service through xinetd, anyone logging in will be able to launch it. If you'd like to retain control over exactly which remote users get to use a particular service, you should use TCP wrappers. To do that, you'll need to edit the appropriate service file in /etc/xinetd.d/ to tell xinetd to load the tcpd daemon, rather than the service daemon itself. As an example, here's what you would add to the vsftpd file:

```
server                  = /usr/sbin/tcpd
serverargs              = /usr/sbin/vsftpd
```

You would also need to permit remote access by editing the disable line to read *no* rather than *yes*:

```
disable                 = no
```

Any edits to the xinetd configuration require a service restart. If you're using Upstart, run:

```
sudo service xinetd restart
```

For Systemd, use this instead:

```
systemctl reload xinetd.service
```

Now you'll need to edit the hosts.allow and hosts.deny files in the /etc/ directory. You could, for instance, add this line to hosts.deny:

```
vsftpd:    ALL
```

which will deny access to users logging in from any external host. The hosts.deny file will be read first, allowing the contents of hosts.allow the last word. Therefore, if you would add something like this to the hosts.allow file:

```
vsftpd:    192.168.0.101, 10.0.4.23
```

then users coming from specifically those two hosts WOULD be allowed.

Encryption: Securing Data in Transit

Files often need to be moved from place to place. If it's just family photos or recipes, you might as well simply send them as e-mail attachments or via ftp (or fax, for those of you old enough to remember such things, although your family photos might not come through the fax at quite their original resolution).

But you should be aware that data packets moving across the Internet are—legally or otherwise—visible to just about anyone who cares to look. For that reason, you should never transfer files containing financial or other private information (including credit card numbers or passwords) through wireless or public digital networks, unless they've been encrypted first.

Encryption rewrites a data file using an encryption algorithm that makes the file unintelligible. If the encryption was strong enough, the only practical way to decrypt it and get access to its contents is to apply the public half of a decryption key pair that essentially performs the encryption process in reverse. Many network communication tools—like telnet, ftp, and most e-mail services—are **not** encrypted and are, therefore, vulnerable.

Now let's look at using the OpenSSH secure shell for encrypted remote login sessions (something I've used a fair number of times already through the demonstrations in this book), and learn about SSH tunnels and encrypting specific files using GnuPG.

OpenSSH

Once the OpenSSH server package is installed on a computer, it can host remote login sessions:

```
sudo apt-get install openssh-server
```

Users who only need to log on to remote systems as guests can install the client package:

```
sudo apt-get install openssh-client
```

Once everything is properly installed, a user can open a new session using the ssh command and enter the password when prompted:

```
ssh tony@10.0.4.243
tony@10.0.4.243's password:
```

All keystrokes and data that travel back and forth for the duration of this session will be securely encrypted. You can also transfer files between sites using the scp ("secure copy") program that's included with OpenSSH:

```
scp myfile.tar.gz tony@10.0.4.243:/home/tony/
```

Note that you will need to specify a target directory on the remote computer where you'd like the file saved. The target directory has to be one to which the user you're logging in as has access. That means you won't be able to copy a file directly to, say, the /var/www/html/ directory of the remote machine.

You can also use scp the other way, to move a file from a remote host to yours. This will copy the newfile.tar.gz file to the current directory (represented by the dot at the end):

```
scp tony@10.0.4.243:/home/tony/newfile.tar.gz .
```

The /etc/ssh/ssh_config file controls the way local users will access remote hosts, while the /etc/ssh/sshd_config file (assuming that openssh- server is installed) manages how remote users log in to your machine. Running *ls* against the /etc/ssh/ directory will display the key pairs ssh uses to authenticate sessions:

```
        $ ls /etc/ssh
moduli            ssh_host_dsa_key            ssh_host_ecdsa_key.pub
ssh_import_id
ssh_config        ssh_host_dsa_key.pub        ssh_host_rsa_key
sshd_config       ssh_host_ecdsa_key          ssh_host_rsa_key.pub
```

Those keys with a .pub extension are public keys, while the versions without an extension are private keys. This directory includes key pairs using the DSA, ECDSA, RSA, and ed25519 encryption algorithms.

OpenSSH version 1 would probably have used key pair files called ssh_host_rsa and ssh_host_dsa. There are other differences between versions 1 and 2. With version 1, for instance, once the client receives the public key from the server, it would use the server's public key to generate and then send a 256-bit secret key. Now that they both have an identical secret key, the two systems can safely share data. Version 2, on the other hand, will use what's called a Diffie-Hellman key agreement to negotiate a secret key without needing to send any complete key over the network.

Passwordless Access

Even OpenSSH has a potential weakness, and it's an old, familiar complaint: the password. Based on the discussion in previous chapters, you still need to authenticate to the host system using a password.

Besides the inconvenience, this extra step also introduces something of a vulnerability into the process. Therefore, wherever possible, you should configure your ssh connections for passwordless access. You do this by generating a new key pair on your client machine (the computer you plan to use to connect to the server):

```
ssh-keygen -t rsa
```

You'll need to choose a specific algorithm type: the above example uses rsa. You can optionally create a passphrase that you'll use later whenever you log in. In any case, using a passphrase is not required and it's often ignored.

The passphrase should not be confused with the host account password, as this one simply locally decrypts the private key and does not require sending account passwords over a network connection.

The new key pair will be saved to the hidden .ssh directory within your user's home directory. You can view the files through ls with the -a (meaning all) option:

```
ls -a ~/.ssh
```

Now you will have to copy the new public key to the host. Remember: you should only do this using a secure transfer method:

```
scp keyname.pub tony@10.0.4.243:/home/tony
```

Log in to the host machine and add the contents of the new key to the ~/.ssh/authorized_keys file within the home directory of the account you will be accessing:

```
cat keyname.pub >> ~/.ssh/authorized_keys
```

If you do decide to pipe it in this way, make sure to use two greater than signs (>>) rather than one, as > will overwrite the file's current contents!

Next, still on the host machine, make sure that the authorized_keys file can be read ONLY by its owner:

```
chmod 600 ~/.ssh/authorized_keys
```

You can now log out of the host. Now for the fun part: try logging in once again:

```
ssh tony@10.0.4.243
```

You should get in without the need for a password. I'll bet that makes you feel really welcome!

Using ssh-agent

You can maintain a higher level of password security without needing to use (or expose) your passwords with each new session you initiate by employing ssh-agent. Running eval within a shell will pass a passphrase to OpenSSH each time you launch a new ssh session from within that shell.

```
eval `ssh-agent -s `
```

Note the use of the backtick (`) symbol, which is usually found at the top left corner of your keyboard.

You can then add your ssh key to the agent using:

```
ssh-add ~/.ssh/id_rsa
```

where id_rsa is the name of the key you wish to add.

X11 Tunnels

You can use ssh connectivity as a platform, or, as it's sometimes known, a tunnel, for a wider range of connected services. So, for instance, you can make remote use of the graphic functionality of an X11 session on top of an existing ssh session. Let's do that step by step.

On the host machine, edit the /etc/ssh/sshd_config file so that the value of the X11Forwarding line is yes:

```
sudo nano /etc/ssh/sshd_config
X11Forwarding yes
```

On the client machine (i.e., the PC you will use to log in to the host), edit the ForwardX11 line in the /etc/ssh/ssh_config file so that it, too, reads yes:

```
sudo nano /etc/ssh/ssh_config
ForwardX11 yes
```

Now, from your client computer, use ssh to start an X session:

```
ssh -X -l tony 10.0.4.243
```

You will find yourself in what looks like just another terminal session. What's so X about this? Don't trust me? Try running a GUI program like gedit (a graphic text editor):

```
gedit
```

If everything worked the way it should, you will find yourself using gedit on your laptop or workstation, but as part of the filesystem, and relying on the resources of your host. Depending on your network connection and memory limits, you might be surprised at the kinds of tasks you can attempt using such tunnels.

GnuPG Config

Our last stop on this journey will be file encryption. While encrypting networked sessions is a perfectly good solution, there may be times when all you want to do is send a single document containing some sensitive information. Rather than encrypting the whole connection, you can simply encrypt the document itself and provide your recipient with the key to decrypt it at the other end.

GPG (Gnu Privacy Guard) will encrypt a file using one or more public keys. The file can subsequently be decrypted using a private key that corresponds to any one of the public keys used during encryption. So, as illustrated in Figure 10-1, you could choose to

generate a random symmetric key using the public key from the recipient's computer to get things started. It will then encrypt the file and send it, along with the new symmetric key. At the receiving end, GPG will use the recipient's private key to decrypt the message.

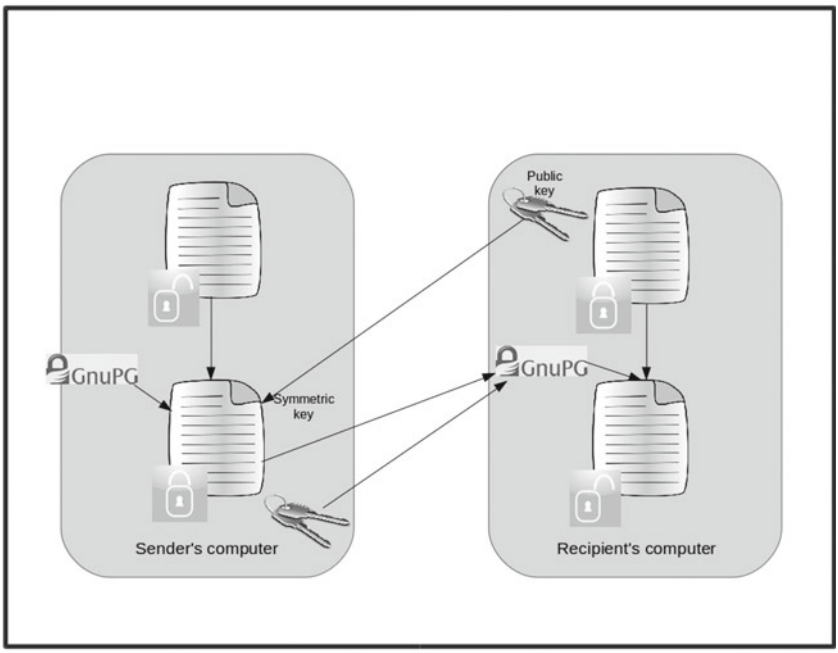

Figure 10-1. *The Gnu Guard key exchange process*

You could also use your own public key for encryption and then send it to your recipients for decryption.

You generate keys using gpg --genkey:

```
gpg --gen-key
```

The program will prompt you for a key type, key length, lifetime (i.e., when, if ever, the key will automatically expire), a username that you can create (and remember for later use), an e-mail address, a comment, and finally, a passphrase. Many of these details are used to help build randomness into the encryption.

The final step will be to produce "noise" (random keystrokes) while GPG generates the encrypted file. GPG expects a LOT of noise, so just hitting a few keys here and there might not do it. What I do is open a new shell (on the same machine, obviously), and create some industrial strength (but harmless) noise using something like this:

```
sudo find / -type f | xargs grep somerandomstring > /dev/null
```

Alternatively, you can install and run the haveged program to quiety generate all the entropy you need. Once it's done, you can go to the hidden GPG directory within your home directory and take a look:

```
cd ~/.gnupg
```

Once you've oriented yourself, it's time to encrypt a file. Assuming there's a file named verysecretdata.txt, here's how it will work:

```
gpg --encrypt \
--recipient tony-key \
--recipient steve-key \
verysecretdata.txt
```

In this example, I used the public keys previously sent to me by both Tony and Steve. I'll discuss importing and exporting public keys in just a moment.

Listing the files in the .gnupg directory once again, you can see that there is now a verysecretdata file with a .gpg extension.

Feel free to send the gpg file to your recipient using any method you like. Remember: it's encrypted so no one else along the way should be able to read it.

Of course, either you or your recipient will have to import each other's key before anything can be done with the file. And before either of you can import it, it will have to be exported and then sent. You export the key using gpg --export:

```
gpg --export username > mygpg.pub
```

The file name you use doesn't matter, as long as it has a .pub extension. Once you've transferred the key to the recipient, they can import it using gpg --import:

```
gpg --import mygpg.pub
```

Assuming that the key is now part of the recipient's GPG collection, they can decrypt it using some variation of this command, where the output value is the name you'd like the decrypted file to have and the value of decrypt is the encrypted file you've received:

```
gpg --output         verysecretdata        --decrypt
verysecretdata.txt.gpg
```

You can view all the keys that are currently part of your system using:

```
gpg --list-keys
```

And, as might sometimes be necessary, you can revoke a key with the key ID that you saw displayed by --list-keys:

```
gpg --gen-revoke 6372552D
```

Using --gen-revoke will generate some output, which you should copy into a file that you can keep safe, as it could prove very useful later. With the file, you will always be able to import your up-to-date key status into a new (or recovery) GPG installation. You import a file using:

```
gpg --import revoked.txt
```

The distribution and updating of keys between users can be managed through online public keyservers. You can retrieve a key using –recv-key and send it with –send-key. Here's what a retrieval command might look like:

```
gpg --keyserver certserver.pgp.com --recv-key 0xBB7576AC
```

Now Try This

Use ssh-keygen to set up passwordless access to a machine (perhaps an LXC container) and try it out to make sure it works. Now, while logged in on that machine, launch a full battery of tests against your own computer, testing for open ports, X access, and remote login access.

If you can't get through your own defenses, pat yourself on the back.

Test Yourself

1. Restricting the output of fuser to a specific port requires which flag?

   a. -v

   b. -i

   c. -sU

   d. -n