

# System Administration 2: Networking and Package Management

## Lesson 1: **Getting Started**

[The Linux Learning Environment](#)

[Using the Linux Learning Environment](#)

[Logging Out](#)

## Lesson 2: **Networking Concepts**

[An Introduction to Networking](#)

[Binary Numbers](#)

## Lesson 3: **Network Addressing**

[IPv4 Addressing](#)

[The IP Address](#)

[The Subnet Mask](#)

[The Network and Broadcast Addresses](#)

[Private Networks](#)

[Subnets Revisited](#)

## Lesson 4: **The Super User**

[The Super User](#)

[Sudo](#)

[The Sudoers File](#)

[A More Convenient Way to Use Sudo](#)

## Lesson 5: **Configuring An Interface**

[First Steps](#)

[Link Status](#)

[Setting An Address](#)

[Network Routes](#)

## Lesson 6: **Persistent Configuration**

[System-Wide Interface Configuration](#)

[Interface Configuration Files](#)

[Configuring the Default Route](#)

## Lesson 7: **DNS**

[A Brief History](#)

[Configuring Your DNS Client](#)

[DNS Client Tools](#)

## Lesson 8: **Package Management and Yum**

[An Introduction to Package Management](#)

[The Yum Package Manager](#)

## Lesson 9: **More Yum Queries**

[Yum List](#)

[RPM](#)

[Yum Clean](#)

Lesson 10: **SSH: The Secure Shell**

The Secure Shell

SSH Keys

Other OpenSSH Tools

SFTP

SCP

Lesson 11: **SSH: The Secure Shell Server**

The Secure Shell Server

Lesson 12: **User Accounts**

The Passwd, Group, and Shadow Files

Account Administration

---

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

# Getting Started

---

Welcome to the second O'Reilly School of Technology (OST) System Administration Course.

## Course Objectives

When you complete this course, you will be able to:

- use basic IPv4 network addressing.
- use super user privileges to make changes to a system.
- configure Ethernet interfaces in a server.
- apply basic DNS concepts.
- install and manage software using the yum package management utility.
- use ssh to connect to other hosts, both with passwords and with public key authentication.
- add and remove user accounts

In this course you will learn basic system administration tasks, such as configuring network interfaces, package management, dns client tools, and user management. These topics are introduced along with the concept of the "super user," an administrative user used for working on the system. By the end of this course, you will be able to configure a Linux-based machine to use the network, install an SSH server, and remotely log into your machine on the network.

From beginning to end, you learn by doing real projects within the OST Learning Sandbox, including a unique environment that allows the student administrative access to your very own virtual server. Besides a browser and internet connection, all software is provided online by the O'Reilly School of Technology.

## Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take the *useractive* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet,

reference books, and online help are all fair game.

- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

## Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:
White boxes like this contain code for you to try out (type into a file to run).
If you have already written some of the code, new code for you to add looks like this.
If we want you to remove existing code, the code to remove <del>will look like this</del> .

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:
The plain black text that we present in these INTERACTIVE boxes is provided by the system (not for you to type). The commands we want you to type look like this.

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:
Gray "Observe" boxes like this contain <b>information</b> (usually code specifics) for you to observe.

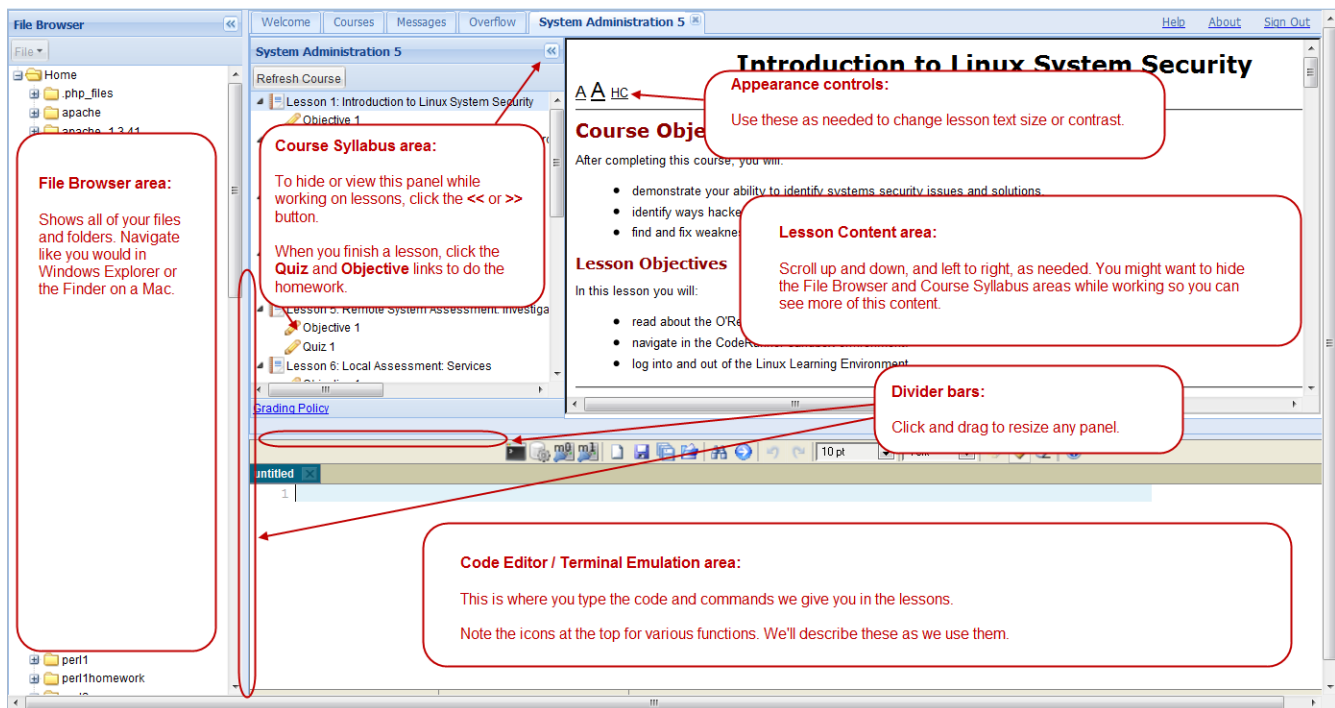
The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

<b>Note</b> Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.
<b>Tip</b> Tips provide information that might help make the tools easier for you to use, such as shortcut keys.
<b>WARNING</b> Warnings provide information that can help prevent program crashes and data loss.

## The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's a quick overview of the various areas of the screen:



These videos explain how to use CodeRunner:

[File Management Demo](#)


[Code Editor Demo](#)

[Coursework Demo](#)

## The Linux Learning Environment

For this course, we'll use CodeRunner to connect to and interact with the O'Reilly School of Technology *Linux Learning Environment*. The Linux Learning Environment gives you access to one or more Linux virtual machines that you can use as a tool to learn Linux. The Linux virtual machine that we'll be using is nearly identical to a physical machine in functionality. Each student receives his or her own virtual machine for this course in order to allow the student complete control over the system.

### Using the Linux Learning Environment

In order to use the Linux Learning Environment, you need to connect to it first. To do that, click on the **Linux Learning Environment, m0** button () located in the toolbar in the CodeRunner window.

#### INTERACTIVE SESSION:

```
cold1 login: username
Password:
```

Your *username* may be entered automatically; otherwise, type it and press **Enter**, and then type your password and press **Enter**.

The following text appears:

#### OBSERVE:

```
Attached to CT 1016948 (ESC . to detach)
```

Press **Enter**.

#### OBSERVE:

```
CentOS release 6.4 (Final)
Kernel 2.6.32-042stab076.8 on an x86_64
smiller-m0.unix.useractive.com login:
Password:
Last login: Thu Jan 23 16:53:27 on tty1
[username@username-m0 ~]$
```

Enter your *username* and password again.

Your Linux virtual machine actually has four consoles, labeled `tty0` through `tty3`, available for you to use to log in. You can access these consoles by clicking the **left** (←) and **right** (→) arrow buttons on the toolbar.

#### Note

You may be curious about why the consoles are called TTYs. The name is derived from early computers that used devices called *teletypes* to allow users to interact with them. Teletypes generally had a keyboard for input and a printer that would print output from the computer so that the user could read it. The term *tty* is derived from the word **t**ele**t**ype.

## Logging Out

It's generally a good idea to log out of your machine when you are finished working with it. In the real world, this keeps unauthorized people from using your account to harm the system. To log out of your Linux system, do this:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ exit
```

A new login prompt appears on the screen. This confirms that you've successfully logged out.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License. See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Networking Concepts

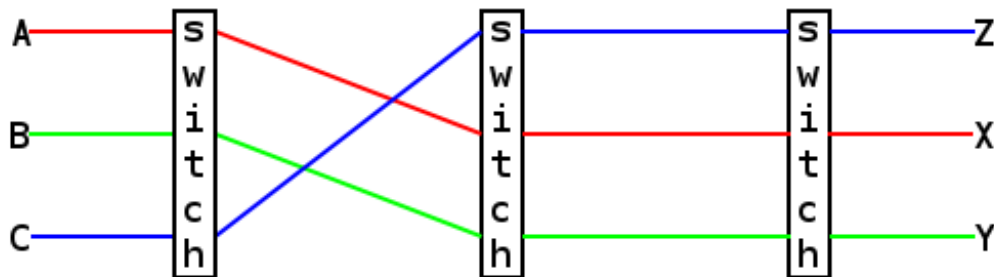
This lesson will cover some basic networking concepts and terminology. Once you're familiar with these concepts, we can start getting into the nuts and bolts of connecting a Linux system to a network.

## An Introduction to Networking

In 1969 the first system connecting computers together in a communications network was deployed. This system, the grandfather of the internet, was called *ARPANET*. ARPA stands for "Advanced Research Projects Agency," and it was the precursor to DARPA, the research division of the armed forces. ARPANET initially connected four universities, but it expanded through the 70s and into the 80s before it was finally decommissioned in 1990. ARPANET provided us with one of the most important elements used in computer networking, the *packet-switched network*.

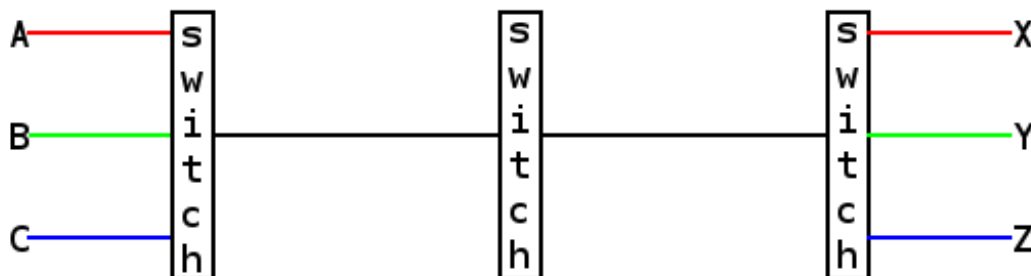
Before the introduction of the packet-switched network, telecommunications (phone and data) occurred over *circuit-switched networks*. That is, any two endpoints that wanted to communicate with each other had to be directly connected via a single circuit composed of multiple point-to-point links connected end-to-end. When a connection was initiated, free links would be assembled into the appropriate circuit by the circuit-switching system. This worked reasonably well for things like telephones, but it had a major downside. Two or more connections could not share a single circuit. Packet-switched networks solved this problem.

**A Circuit Switched Network:**  
A connects to X, B connects to Y and C connects to Z.



In a packet-switched network, data that is intended to be sent from one location to another is assembled into a *packet*. Depending on the protocol being used, this data packet generally contains information about the source of the packet, the destination of the packet, and the actual data to be transmitted. You can think of a packet like a letter to be mailed using the postal system. The envelope has a recipient address and a return address, and inside is the data you want to send. By encoding the source and destination directly in the packet, switching points on a packet-switched network can make decisions about where to send the packet. This allows packets from multiple sources destined for multiple addresses to share the same link.

**A Packet Switched Network:**  
A, B and C share a single link to connect to X, Y and X



Similar to a circuit-switched network, a packet-switched network is often composed of many shorter point-to-point

links. At each of the locations where these links meet, there is hardware designed to *switch* or *route* the packets to their various destinations. The difference between switching and routing lies in how devices are addressed on the network. Packet switching occurs when the packet's source and destination are on the same network. Returning to the postal analogy, it would be similar to sending a letter to someone else who lives in the same town. The post office would get your letter and without sending it to any other post offices, would send it directly to the recipient. Conversely, routing takes place when the source and destination addresses of a packet lie on different networks. This is similar to sending a letter to a person in another town. The local post office on the sender side would not be able to send the letter directly to the recipient, but would need to send the letter to the post office in the recipient's town. That post office would then be able to deliver the letter to the proper recipient. This can all be boiled down to two statements: *packets that are not destined to leave the local network are switched* and *packets that are destined for a remote network are routed to that network*.

The entire internet is composed of an enormous number of smaller networks that are connected together via routers. The actual technologies that are used to connect these routers vary depending on several factors, but that subject is beyond the scope of this course. We'll concentrate here primarily on communications within a *local area network* or *LAN*. We will also briefly discuss routing as it pertains to sending traffic from a LAN out to the internet.

Generally, packets on a LAN are transmitted using a technology called *ethernet*. While there are multiple ways to physically interconnect ethernet-capable devices, the format of ethernet packets does not vary. Ethernet packets or *frames* have the same format whether they are being sent over a Cat 5 copper ethernet cable or a fiber optic cable. In addition to the standard source and destination information encoded in the generic packets we discussed earlier, the ethernet frame also contains *checksum* information, which allows the recipient to determine whether the data in the frame has arrived without transmission errors. This functionality was added to ethernet to prevent data loss when dealing with potentially unreliable connections.

The ethernet protocol is just part of the networking process though. Ethernet describes how to get data between two points on a network. Other protocols exist that use ethernet as a means to transmit data. The internet protocol (IP) describes a way for hosts to be addressed on a network using *IP addresses*. IP addresses differ from the addresses used in ethernet frames known as *media access control* or *MAC* addresses. IP addresses are used to route packets between destinations; MAC addresses are used for switching packets on a network. The IP address is used to look up the MAC address on a LAN segment, and that MAC address is used for communication via ethernet.

The internet protocol is not at the top of the network stack though. Above IP, there are several other protocols, most notably the *Transmission Control Protocol (TCP)*. In this course, we'll concentrate on TCP because it's the protocol used to transmit data for HTTP, email, and much more. TCP is an important piece of the networking puzzle; it allows simultaneous multiple network connections to or from a single host by defining numbered *ports*. Imagine that a computer is like a warehouse, and each TCP port is a loading dock, allowing trucks from numerous destinations to load and unload at the same time without interrupting each other.

Finally, we have applications that use TCP (or any protocol that exists above IP) to transmit data. Each level, from applications to protocols to physical links, is called a *layer*. Commonly, you will hear ethernet referred to as *layer 2* and IP as *layer 3*. Packet switching occurs in layer 2, the ethernet layer, and packet routing occurs in layer 3, the IP layer.

## Binary Numbers

You need a basic understanding of binary numbers in order to understand certain networking concepts. You'll need to be able to convert decimal numbers into binary and back. Before we get into the actual business of conversion, we'll learn about number bases using a system that we're already familiar with: decimal numbers.

The system is called "decimal" because it's in *base 10*. The Latin prefix "deci" means "tenth." You've worked with decimal numbers all your life, so by now thinking in base 10 is natural to you. Even so, we'll have to deconstruct this process now in order to fully understand what it means to be "base 10." The first concept I want to introduce is the *exponent*. You've likely learned about exponents at some point in your life, but let's refresh your memory. In the next example, **10** is our base and **2** is our exponent:

$$\begin{array}{c} 10^2 \\ \hline \text{base} \end{array} = 100$$

exponent

When a base is raised to a power, that means that base number is multiplied by itself the number of times that the



power specifies. In our example above, we multiply  $10 \times 10$  to get 100. If it was  $10^3$ , we would multiply  $10 \times 10 \times 10$  to get 1000. 10 raised to any power is called a *power of 10*. In decimal numbers, each digit of the number can be thought of as a multiplier that you multiply against a given power of 10 for that place. Each place has a specific power of 10 that goes with it. We'll demonstrate how this works with the number **953**:

$$\begin{array}{r} 9 \times 10^2 = 900 \\ 5 \times 10^1 = 50 \\ 3 \times 10^0 = 3 \\ \hline 953 \end{array}$$

**Note**  $10^0=1$ . Any number raised to the 0th power is equal to 1.

This may not be new information to you, but it's important to understand. In this example we showed the process of converting a decimal number into exponents and back into a decimal number. We can use almost the same process to convert a binary number into a decimal number.

Binary is a base 2 numbering system (The "bi" prefix means 2, as in bicycle). You will need to familiarize yourself with the powers of 2 in order to do more efficient conversions between binary and decimal. For this course, you will not be working with particularly large binary numbers, so we're concerned for the most part with the 0th power of 2 up to the 7th power of 2:

Power	Value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

Each of these powers corresponds to a digit in a binary number, and much like decimal, the largest or *most significant* digit is to the left. Digits in binary numbers have the special name, *bit*. The name **bit** is a contraction of **b**inary and **d**igit. In the computing world, we often refer to a grouping of 8 bits and call it a *byte*. In this course, we'll work exclusively with binary numbers that are one byte long. That's why the table above shows the first 8 powers of 2, starting at 0.

Okay, let's go ahead and convert a decimal number into a binary number. For this example, we'll use the number 203 (the largest decimal number that will fit into 8 bits or 1 byte is 255). You'll want to have some paper to work on for this task. First, write down the powers of 2 in a row, starting with 128 on the left, then 64 and so on until you reach 1. Then, start with the largest power of 2, 128. Since 203 is larger than 128, put a 1 under 128, and then subtract 128 from 203, leaving 75. Next, do the same for 64. 75 is larger than 64, so mark a 1 under 64 and subtract 64 from 75, leaving 11. Because 11 is smaller than 32, you can mark 0 under 32 and move on to the next smaller power of 2. Repeat this process until you have either a 1 or 0 under all 8 powers of 2 you wrote down. The series of 1s and 0s you wrote underneath the powers of 2 (11001011) is the binary number for 203. You can apply this process for any decimal number, though it will require using higher powers of 2 if the desired number is larger than 255.

The process of converting a binary number into a decimal number is much more straightforward. To demonstrate this, we'll use the binary number 10110010. As with decimal numbers, each of the places in a binary number correspond to a power of two. The power of 2 for a given place is multiplied by the value of the bit at that place. This diagram shows the conversion between binary and decimal, similar to the diagram from above showing how decimal digits work.

$$\begin{array}{r}
 1 \times 2^7 = 128 \\
 0 \times 2^6 = 0 \\
 1 \times 2^5 = 32 \\
 1 \times 2^4 = 16 \\
 0 \times 2^3 = 0 \\
 0 \times 2^2 = 0 \\
 1 \times 2^1 = 2 \\
 0 \times 2^0 = 0 \\
 \hline
 178
 \end{array}$$

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
 See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Network Addressing

In this lesson, we'll discuss how hardware is addressed on a network. Network addresses are used to give packets a source and a destination. These source and destination addresses are used by networking hardware to direct packets properly.

## IPv4 Addressing

The most widely deployed version of the internet protocol is version 4. Hosts on a network are assigned an IP address and a subnet mask. These two addresses together give the host a unique identifier on a network, as well as inform the host which network it is on. By knowing which network a host is on, decisions can be made about how to route packets within and outside of the network.

### The IP Address

As you learned earlier, IP addresses are used by devices on a network to communicate with each other. An IP address consists of four numbers, separated by periods. Each number in an IP address is called a *dotted quad*. For example, the IP address for OST's website is 199.27.144.89. Each device on a particular network must have a unique IP address.

An IP address can be split into two parts, a *network prefix* and a *host identifier*. The host identifier does exactly what its name suggests: it identifies a host on a network. The network prefix identifies the network to which the IP address belongs. The split between network prefix and host identifier can vary and is determined by the *subnet mask*.

### The Subnet Mask

The subnet mask, like an IP address, can be written using dotted quad notation. IP addresses and subnet masks share this notation because they are of the same length. That's no coincidence. The subnet mask is used to "mask off" parts of the IP address using *bit masking*. Bit masking is accomplished by comparing one binary number to another "bitwise," which means that each bit in one number is compared to the corresponding bit in the other number. Here's a table that outlines the results of various comparisons:

Bit 1	Bit 2	Result
0	0	0
1	0	0
0	1	0
1	1	1

Next we'll look at an example of how one 8-bit number (11110000) masks off another 8 bit number (10101100):

First Number:	1	0	1	0	1	1	0	0
	↓	↓	↓	↓	↓	↓	↓	↓
Second Number:	1	1	1	1	0	0	0	0
	↓	↓	↓	↓	↓	↓	↓	↓
Result:	1	0	1	0	0	0	0	0

You can see that when our second number (the mask) is a 1, the first number is preserved as a result.

This is how the subnet masks off bits in the IP address to denote them as being part of the network prefix. Bits that are not a part of the network prefix are part of the host identifier. You can also write subnet masks using *prefix length notation*, which is a quicker way to specify the subnet mask. The prefix length is specified as */n*, where *n* is the number of 1s in the subnet mask, starting from the bit farthest to the left. Below is a diagram that demonstrates the relationship between an IP address (199.27.144.89), a subnet mask (255.255.248.0), a prefix length notation (/21), and the resulting network prefix and host identifier.

	21 Bits	11 Bits
Address In Binary:	11000111.00011011.10010000.01011001	
Subnet Mask In Binary:	11111111.11111111.11111000.00000000	
Network Prefix:	11000111.00011011.10010	
Host Identifier:		000.01011001

The number of bits left when you subtract the prefix length from 32 (the longest prefix length possible) tells us how many hosts we can have on our network. In this example, we have 11 bits for the host identifier, so we can have up to 2048 ( $2^{11}$ ) hosts—or can we?

## The Network and Broadcast Addresses

For all networks with a subnet mask of /30 or smaller, two addresses must be reserved from the pool of available addresses, the *network address* and *broadcast address*. This means that in a /21 network like the one we had in our example, we can only have 2046 ( $2^{11}-2$ ) hosts, not 2048. These two addresses serve a special purpose in the IPv4 protocol. The network address is used to refer to an entire network of hosts at one time. Most commonly, this address would be used in addition to a subnet mask in order to specify a large range of hosts for some operation, such as traffic filtering. The network address is formed by picking the smallest address from the network.

The broadcast address also exists as a shorthand for all the hosts on a network, but there is one important distinction. You can use the network address to refer to an entire network, but you can't send packets to that address. However, the broadcast address can receive packets. Packets sent to the broadcast address go to all hosts on the specified network. The broadcast address is derived from the largest address on the network. Here are some examples of various network and broadcast addresses for networks of varying sizes:

Host Range	Subnet Mask	Network Address	Broadcast Address	Number of Hosts
192.168.0.1 to 192.168.0.254	255.255.255.0 or /24	192.168.0.0	192.168.0.255	254
199.27.144.1 to 199.27.151.254	255.255.248.0 or /21	199.27.144.0	199.27.151.255	2046
10.0.0.9 to 10.0.0.14	255.255.255.248 or /29	10.0.0.8	10.0.0.15	6

### Note

That last entry shows that the network address doesn't always end in a 0, and the broadcast address does not always end in 255.

## Private Networks

Devices on a network don't necessarily need a publicly addressable IP address; they only need to be accessible on your network, and not from the internet at large. In fact, in many situations, for security purposes it's better if devices are *not* accessible directly from the internet. Privately addressed devices can send packets to the internet using *network address translation* or NAT, but NAT is beyond the scope of this course. There are three networks that are designated as private networks. Anyone can use these networks without obtaining permission from one of the internet's regulatory bodies. Here's a summary:

Network	Address Range	Subnet Mask	Number of Hosts
192.168.0.0/16	192.168.0.0-192.168.255.255	255.255.0.0 or /16	65,536 addresses or 65,534 hosts
172.16.0.0/12	172.16.0.0-172.31.255.255	255.240.0.0 or /12	1,048,576 addresses or 1,048,574 hosts
10.0.0.0/8	10.0.0.0-10.255.255.255	255.0.0.0 or /8	16,777,216 addresses or 16,777,214 hosts

**Note** When configured, your machine will be a part of the "172.16.0.0/12" network.

## Subnets Revisited

The subnet mask determines which network includes a particular host. This, coupled with the fact that the length of a subnet mask is variable, allows us to take a network block and split it into several smaller network blocks. This can help with efficient IP address allocation on your network, as well as provide a method by which to segregate machines according to security policies. As an example, let's use a private network, 10.0.0.0/8. With over 16 million allocatable addresses, it's a prime candidate for being broken into smaller blocks of allocatable addresses. In fact, it's possible to have up to 65,536 /24 networks, just inside the 10.0.0.0/8 space! Let's take a look at the differences between two hosts on distinct /24 blocks in the 10.0.0.0/8 space:

**Host: 10.0.0.127/24**

In Binary:	00001010.00000000.00000000.01111111
Subnet Mask:	11111111.11111111.11111111.00000000
Network Prefix:	00001010.00000000.00000000
Host ID:	01111111

**Host: 10.0.1.127/24**

In Binary:	00001010.00000000.00000001.01111111
Subnet Mask:	11111111.11111111.11111111.00000000
Network Prefix:	00001010.00000000.00000001
Host ID:	01111111

As you can see, these two hosts have the same host id, but they're on different networks: 10.0.0.0/24 and 10.0.1.0/24. In fact, without some kind of routing in between them, these two hosts cannot communicate with each other. Now take these two IP addresses, but alter their subnet mask to /23. Now they are no longer two separate networks with 256 addresses each, but instead they comprise a single network with 512 addresses. If you're still a little fuzzy on this concept, draw yourself a chart like the one above and see how the subnet mask affects the network address and host id for each host.

We're three lessons into our course, and you're doing well so far. Keep it up and see you at the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# The Super User

In this lesson you'll be introduced to the concept of the *super user*, and learn how to elevate your privileges to the super-user level.

## The Super User

Earlier, we discussed how every file on the server has three sets of permissions: user, group, and everyone else. This restricts people from accessing files that are critical to the system, or other people's private files. It would be awfully difficult to administer a system if you could only access and change your own files—or if everyone could access and change everyone else's files! There is one person who has access to every file on the server though: the System Administrator. The administrator has access to the super user account with the login name of *root*. Along with the power granted to the super user comes great responsibility. Having access to every file on the system means that you could destroy any file there. A couple of wrong keystrokes and you may have to re-install the whole system from scratch!

If that scares you a little bit, well...it should. It's important to remember to be extremely careful when you don't have any restrictions.

## Sudo

So how do you get super user privileges? In modern Linux distributions, that task is accomplished with the **sudo** command (it's a contraction of "super do," but some pronounce it so it rhymes with "judo"). Prefixing an operation with the **sudo** command allows you to perform that operation as the super user. The first time you invoke **sudo**, it will ask for your password before it performs the operation you've requested. Subsequent calls to **sudo** will not ask for your password for up to five minutes as a convenience, but after that period of time you'll need to enter your password again. Here's an example that illustrates how **sudo** works:

### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo whoami
[sudo] password for username:
root
```


The **whoami** command would normally return your username, but because it was run with **sudo**, it returned "root," which is the super user's username. But wait, if any user can just run the **sudo** command and supply their own password, doesn't that make the system really insecure? Good question!

## The Sudoers File

The short answer is "no." Not every user can employ **sudo** to elevate their privileges. Access to the **sudo** command is controlled by the **sudoers** file, which is located at **/etc/sudoers**. This file contains a list of users and groups, and the commands upon which they can use **sudo**. The **sudoers** file is editable only by someone with super user privileges, so regular users cannot add themselves to the file. There is also a special program used to edit the **sudoers** file, called **visudo**. You can edit the **sudoers** file by hand, but I strongly discourage it. If a mistake is made in the **sudoers** file it could mess up your system; it can be pretty difficult to recover from such mistakes. The **visudo** command edits the file and then does syntax checking on the file to ensure that all entries in the file are compliant. Keep in mind that even syntactically correct entries can have unintended and unwanted consequences, so be really careful when you edit the file.

## A More Convenient Way to Use Sudo

Using **sudo** to run a command here or there is fine, but what happens when you have several tasks to perform that require you to have elevated privileges? **Sudo** provides a mechanism that allows you to keep your elevated privileges for an extended period of time while running **sudo** only once: the **-s** option. It causes **sudo** to launch a new shell with super user privileges. Connect to the Linux Learning Environment by clicking

on the **Linux Learning Environment, m0** button () in the toolbar in the CodeRunner window, and then enter the following commands:

## INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo -s
[sudo] password for username:
[root@username-m0 username]# cd
[root@username-m0 ~]# whoami
root
[root@username-m0 ~]# pwd
/root
[root@username-m0 ~]# exit
exit
[username@username-m0 username]#
```

When you start a shell using **sudo**, you have essentially become root within that shell. You can go anywhere on the system and edit any of the files. Be *very* careful with this power. An inadvertant "rm \*" in the wrong place can render your machine unusable.

We are moving right along. Go ahead and do the homework, and I'll see you in the next lesson...

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

# Configuring An Interface

In this lesson, we'll configure the network interface in your machine for the first time, using the **ip** utility. We'll also go over network routes and learn how they affect the path that your packets take to reach their destinations.

## First Steps

Before you can configure your machine's IP address, you need to know the address to assign to it. To figure out what your IP address should be for this course, use the **host** command. First, launch a terminal in CodeRunner by clicking the **New Terminal** (🖥️) button. This starts a new shell for you on our "cold" login server. If you're not automatically logged in, go ahead and log in. Once you have a command prompt, you can use the **host** command to look up your machine's IP address. The hostname you should use for the lookup is "username-m0.unix.useractive.com," replacing *username* with your username.

### INTERACTIVE SESSION:

```
cold1:~$ host username-m0.unix.useractive.com
username-m0.unix.useractive.com has address 172.16.n.n
```

The IP returned will have numbers in place of each *n* and is the IP address of your LLE. Make a note of your IP address; this will be used as the IP of the ethernet device you will be setting up on your LLE. If you receive any kind of error message, notify your mentor. Once you have your IP address, type **exit** to log out of the terminal session.

## Link Status

If you haven't already started your machine and logged into it, do so now. Before we do any configuration, let's take a look at the state of your ethernet interface, using the **ip** command. The format that the **ip** command uses is:

### OBSERVE:

```
ip object command
```

We'll introduce objects and commands as we use them. The first **object** we're concerned with is **link**, which will allow us to control and view the link status of our interfaces. The **show** command will print out the current status for us:

### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: sit0: <NOARP> mtu 1480 qdisc noop state DOWN
    link/sit 0.0.0.0 brd 0.0.0.0
3: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000
    link/ether fe:fd:00:00:04:14 brd ff:ff:ff:ff:ff:ff
```

That's quite a confusing mess of information! In fact, it's a lot more information than we actually need. Of the three interfaces listed, we are only concerned with the status of one of them: eth0. The name of this interface is derived from **e**thernet interface **0**. If you had two ethernet interfaces in your computer, they would be called "eth0" and "eth1." All ethernet interfaces under Linux are named this way. Let's rerun the **ip** command, but this time we'll tell it we only want to know about eth0. The important part of the output is in **red**:

### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ ip link show eth0
3: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000
    link/ether fe:fd:00:00:04:14 brd ff:ff:ff:ff:ff:ff
```



Much better! We can see that as of this instant, **eth0's link is down**. We can't connect to anything while our interface is down, so let's bring it up. To do this, we need root privileges, so we'll use **sudo** with the **ip** command. Again, we highlighted the important parts in **red**:

#### INTERACTIVE SESSION:

```
[username@username-m0 dbasset]# sudo ip link set eth0 up
We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:
#1) Respect the privacy of others.
#2) Think before you type.
#3) With great power comes great responsibility.
[sudo] password for username:
[username@username-m0 dbasset]# ip link show eth0
3: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN qlen
1000
    link/ether fe:fd:00:00:04:14 brd ff:ff:ff:ff:ff:ff
```

#### Note

There is one weird issue with the output of **ip link show eth0**. The state is **UNKNOWN**. Usually when the interface is brought up, we would see "state UP." But due to limitations with our virtualization environment, the **ip** command can't report the status of the interface correctly. Fortunately, the status flags displayed between the < and > brackets are correct.

Based on the status flags between the < and > brackets, our interface is **UP**. Also take note of the flag that says **LOWER\_UP**. There is an important distinction to be made between "UP" and "LOWER\_UP." "UP" just means that the interface is activated for the OS to use; "LOWER\_UP" means that the interface has an active physical link. If this were a real machine and its ethernet cable was unplugged, you would not see "LOWER\_UP," and you would not be able to transmit data.

## Setting An Address

We have both "UP" and "LOWER\_UP," but we still can't do anything useful yet. We still have to assign an IP address and netmask to the interface. This can also be done with the **ip** command, using the **addr** (address) object. We'll give the **ip** command an address to assign to the interface using dotted quad notation, and we'll specify the netmask using prefix length notation. You may recall that your machine will be on the 172.16.0.0/12 network. This means that you use /12 (or 255.240.0.0 in dotted quad notation) as your subnet mask. You can use this information, along with the IP address you found at the beginning of this lesson, to set the IP address of your machine (substitute your IP address for "172.16.n.n"):

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo ip addr add 172.16.n.n/12 dev eth0
[username@username-m0 ~]$ ip addr show eth0
3: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN qlen
1000
    link/ether fe:fd:00:00:04:14 brd ff:ff:ff:ff:ff:ff
    inet 172.16.n.n/12 scope global eth0
    inet6 fe80::fcfd:ff:fe00:414/64 scope link
        valid_lft forever preferred_lft forever
```

Let's break down that **ip** command a little to see what we've done. Running **ip** with the **addr** object tells it that we want to work with the address settings for some device (in our case, **eth0**, as specified at the end of the command). We want to add an address to this interface, so we use the **add** command. The **add** command requires that we specify an address (with netmask), and a device where we'll add that address. You specify the device as **dev devicename**. After we've added the address, we want to check to make sure it's actually there; we can use the **show** command with the **addr** object to check the current status of the interface. As with the **link** object, specifying the device you want to show information for is optional, but useful.

Now that your interface is up and has an address, you can test to see if it's working. A good tool for this task is **ping**. Ping sends a special kind of data packet, called an ICMP packet, to a specified host. When that host receives the ICMP

packet, it generally sends back a response to let you know that it can be reached. In some circumstances, however, the host on the other end will be configured not to respond to pings. If you know of a host that is configured to return pings, you can ping that address as a quick test to make sure your network interface is sending and receiving traffic. Let's ping an address on our network to see if everything is working as it should (use **Ctrl+c** to stop ping, shown below as **^C**):

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ ping 172.16.0.1
PING 172.16.0.1 (172.16.0.1) 56(84) bytes of data.
64 bytes from 172.16.0.1: icmp_seq=1 ttl=64 time=15.2 ms
64 bytes from 172.16.0.1: icmp_seq=2 ttl=64 time=0.673 ms
^C
--- 172.16.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1859ms
rtt min/avg/max/mdev = 0.673/7.982/15.292/7.310 ms
```

It looks like everything is working properly. Now let's try pinging a publicly accessible address, the O'Reilly School of Technology's website:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ ping 199.27.144.89
connect: Network is unreachable
```

Network is unreachable? But we were just able to ping something else! This is what happens when your machine doesn't know where to send a packet. At this point, you are able to ping hosts on your network, but your machine doesn't know how to communicate with machines outside of your network. To tell your machine what to do with packets destined for other networks, you use a *network route*. The idea behind a network route is that a packet that is not destined for your network can be sent to another device on the network (like a router or gateway) that knows where the packet should go and can handle forwarding it for you. In normal usage, defining something called the *default route* is good enough. The default route is where all packets not destined for your network (or in more advanced usage, another specifically defined route) are sent. Ideally, the device on the other end of the default route will know where to send your packets.

## Network Routes

Once again, the **ip** command can help us achieve our goal. We want to set a default route. Fortunately, **ip** has a **route** object for us to use. Before we add anything, let's take a look at the status of our *routing table*. It holds information regarding routes to various networks:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ ip route show
172.16.0.0/12 dev eth0 proto kernel scope link src 172.16.n.n
```

#### Note

It seems that there's a pattern developing here. Whenever you want to see the status of an aspect of the networking on your machine using **ip**, you run **ip object show**.

We have exactly one route in our routing table right now, and that route points to our own network. We could anticipate this to be the case because our machine knows how to handle packets bound for our network. Now let's add a default route so we can send packets outside:

## INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo ip route add default via 172.16.0.1
[username@username-m0 ~]$ ip route show
172.16.0.0/12 dev eth0 proto kernel scope link src 172.16.n.n
default via 172.16.0.1 dev eth0
```

Now we have two routes. Any additional routes displayed via **ip** will look like the entry for the default route, with the network address displayed instead of "default." For example, if we added a route to the network 192.168.0.0/24, and the gateway to that network was listening on 172.16.0.5, the route entry would look like "192.168.0.0/24 via 172.16.0.5 dev eth0." We've set our route; let's see if we can ping outside of our network now:

## INTERACTIVE SESSION:

```
[username@username-m0 ~]$ ping 199.27.144.89
PING 199.27.144.89 (199.27.144.89) 56(84) bytes of data.
64 bytes from 199.27.144.89: icmp_seq=1 ttl=63 time=34.9 ms
64 bytes from 199.27.144.89: icmp_seq=2 ttl=63 time=0.859 ms
^C
--- 199.27.144.89 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1710ms
rtt min/avg/max/mdev = 0.859/17.921/34.984/17.063 ms
```

Brilliant! You can now reach the outside world. You may want to take a look at **ip**'s **man** page to see what else it's capable of doing before you move on. In the next lesson we'll cover configuring your network interface so that it's automatically brought up at boot time. I'll look for you there!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# Persistent Configuration

In this lesson, we'll learn to define the network configuration in the system so that your interface is configured automatically at boot time. We'll also cover the **ifup** and **ifdown** tools.

## System-Wide Interface Configuration

On a vast majority of servers, one or more addresses are assigned to the installed interfaces, and those addresses stay the same on that server for a long time. Most Linux-based operating systems provide a means by which to define a static configuration for installed interfaces that is applied whenever the system boots. This configuration is also read whenever the administrator uses tools such as **ifup** and **ifdown** to manually start and stop the interface. We'll cover those two tools later in the lesson.

### Interface Configuration Files

If you took the first course in this series, you may recall that system configuration files are almost always located in `/etc`. The same holds true for network configuration. On Redhat (and Redhat derivatives such as CentOS), network configuration files are located in `/etc/sysconfig/network-scripts`. Each configuration file contains a set of parameters and values in the format **PARAMETER=value**. These configuration files are read by the scripts that are responsible for bringing the interfaces up and down. Go ahead and take a look at the configuration file for `eth0`, `/etc/sysconfig/network-scripts/ifcfg-eth0`, using your favorite method (`vi`, `less`, `cat`, or other method). You'll see something like this:

#### OBSERVE:

```
DEVICE=eth0
BOOTPROTO=none
ONBOOT=no
```

The first line makes sense, but what do "BOOTPROTO=none" and "ONBOOT" mean? The "BOOTPROTO=none" parameter defines how the IP address is assigned to the given interface, either statically (we supply the address that the interface uses) or with *Dynamic Host Configuration Protocol (DHCP)*. DHCP allows the host to send a message to a server on the network that is responsible for assigning IP addresses and other information to hosts that make a DHCP request. This is how most home or work computers are configured; servers usually have their IP addresses statically assigned. The logic behind this is that if the DHCP server becomes unavailable, servers will not have their address assignments affected by the outage. For this course, we will use static address assignments, so edit the configuration file as shown (the file is owned by root, so you will need to open it with **sudo**):

#### CODE TO TYPE:

```
DEVICE=eth0
BOOTPROTO=static
ONBOOT=yes
```

At this point we have instructed the machine to bring `eth0` up on boot, use a static IP address, and to bring up our interface on boot. But we haven't supplied it with an address or netmask to assign to the interface. We do this with the "IPADDR" and "NETMASK" parameters. In this file, the netmask is specified in dotted quad notation, not in prefix length notation. Add the "IPADDR" (the IP address you found in the last lesson) and "NETMASK" lines to your configuration file and supply the information you obtained about the address and netmask in the last lesson (as with most configuration files, it is important that you enter only one parameter per line):

#### CODE TO TYPE:

```
DEVICE=eth0
BOOTPROTO=static
ONBOOT=yes
IPADDR=172.16.n.n
NETMASK=255.240.0.0
```

Now write the file out, quit the text editor, and reboot your machine using the **reboot** command (you'll need to use **sudo** again). When it finishes booting up, use the **ping** command to test out your interface, trying both

172.16.0.1 and 199.27.144.89:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ ping 172.16.0.1
PING 172.16.0.1 (172.16.0.1) 56(84) bytes of data.
64 bytes from 172.16.0.1: icmp_seq=1 ttl=64 time=0.844 ms
64 bytes from 172.16.0.1: icmp_seq=2 ttl=64 time=0.561 ms
^C
--- 172.16.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1348ms
rtt min/avg/max/mdev = 0.561/0.702/0.844/0.143 ms
[username@username-m0 ~]$ ping 199.27.144.89
connect: Network is unreachable
[username@username-m0 ~]$
```

## Configuring the Default Route

You can't ping 199.27.144.89, right? That's because you haven't defined a default route yet. Use the **ip** command to confirm this. The default route is not set in the **ifcfg-eth\*** files; it is defined in the **/etc/sysconfig/network** file. Open this file for editing so we can define a default route. You'll see that it has the same "PARAMETER=value" format as the **ifcfg-eth\*** files. You'll also see that it's where the hostname for your machine is set. Edit the file as shown:

#### CODE TO TYPE:

```
NETWORKING=yes
HOSTNAME=username-m0.unix.useractive.com
GATEWAY=172.16.0.1
```

This time, instead of rebooting your machine to apply the changes, we'll do it another way. Rebooting a server for small system configuration changes causes costly downtime and should be avoided whenever possible.

We can manually force the network configuration scripts on our system to take down and bring up interfaces using the commands **ifup** and **ifdown**. The syntax for these commands is identical, and looks like this:

#### OBSERVE:

```
ifup interface
```

Stop **eth0** using **ifdown**, and then start it back up using **ifup**. When the interface is back up, confirm that the default route has now been configured using **ip**. Ping 199.27.144.89 to confirm that everything is working as expected.

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ ifdown eth0
[username@username-m0 ~]$ ifup eth0
[username@username-m0 ~]$ ping 199.27.144.89
PING 199.27.144.89 (199.27.144.89) 56(84) bytes of data.
64 bytes from 199.27.144.89: icmp_seq=1 ttl=63 time=13.2 ms
64 bytes from 199.27.144.89: icmp_seq=2 ttl=63 time=0.905 ms
^C
--- 199.27.144.89 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1349ms
rtt min/avg/max/mdev = 0.905/7.076/13.248/6.172 ms
[username@username-m0 ~]$
```

So far we have referred to other hosts using their IP address rather than host name. That's because until now your machine has not been configured to look up IP addresses based on a given host name using the *Domain Name Service (DNS)*. This service is critical to the operation of most systems. We'll learn how to set

that up in the next lesson. See you there!

*Copyright © 1998-2014 O'Reilly Media, Inc.*



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

# DNS

The Domain Name Service, or DNS, is integral to network computing. It provides a mechanism by which IP addresses can be looked up using host names, and vice versa. If it weren't for DNS, you wouldn't be able to type "google.com" into your browser and get to the search engine. Instead, you would have to remember the IP address that points to Google's web server. It would be like having to remember the phone numbers of every person you know! DNS is like the phonebook of internet addresses.

## A Brief History

When the internet was first created, DNS did not exist. Instead, each internet-connected machine had a file that mapped host names to IP addresses. Most systems still have this file, though it's usually empty, or only contains a couple of entries. This file is located at `/etc/hosts`. When new hosts were added to the internet, their address had to be added to the hosts file, and the new file had to be sent out to all computer users to install on their machines. As the internet grew, it became increasingly difficult to keep the hosts file current. This problem was solved by the creation of DNS.

DNS allows for centrally managed databases that map host names to address. Each domain is responsible for managing its own DNS entries. If you send a query to your domain's DNS server for a host that is not within your domain, your DNS server has the ability to query the DNS server for the correct domain and return an answer to you. This structure helps to reduce the work associated with passing around the `/etc/hosts` file. In the next course we'll discuss DNS servers at length, but for now we'll just cover DNS on the client side.

## Configuring Your DNS Client

By default, almost all machines can query DNS, but they may not be set up to do it automatically. This is the case with your machine. Configuring the DNS client to query a DNS server involves adding a few lines to the `/etc/resolv.conf` file. The name of this file is derived from the term used to describe the response to a DNS query. If you were to look up the host "oreillyschool.com," and get back the address "199.27.144.89," you would say that "the host oreillyschool.com resolved to the address 199.27.144.89." Create the file `/etc/resolv.conf` with these contents (you'll need to use root privileges to do this):

### CODE TO TYPE:

```
domain useractive.com
nameserver 172.16.0.1
nameserver 172.16.0.2
```

### Note

We use the "useractive.com" domain here partly as a tribute to our humble beginnings. When you define a domain parameter, you can use shorter hostnames later (for example, "cold" rather than "cold.useractive.com").

After you've written the file, you can use the **host** command to see whether DNS is working:

### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ host oreillyschool.com
oreillyschool.com has address 199.27.144.89
oreillyschool.com mail is handled by 20 ALT2.ASPMX.L.GOOGLE.com.
oreillyschool.com mail is handled by 10 ASPMX.L.GOOGLE.com.
oreillyschool.com mail is handled by 20 ALT1.ASPMX.L.GOOGLE.com.
```

Although there's no indication of it here, the name for the IP address that's returned by this query is an *A record*. There are several classes of records for hosts and domains. The other three entries you see here are *MX* or *Mail eXchanger* records. These tell you which machines handle the email for the given host or domain. We'll cover a few more record types in the next section when we introduce tools that enable you to make more detailed DNS queries.

## DNS Client Tools

Now try running a DNS lookup on "google.com" using **host**. You'll see nearly a dozen IP addresses returned, as well as several MX records. DNS allows one host name to have multiple A records. This is primarily done to achieve

round-robin load balancing. Each time the DNS server is queried for a host with multiple A records, the list of A records is returned in a different order, that way multiple clients can be directed to various servers, allowing the load to be spread among them.

It's also possible to have multiple MX records (as you can see in both the oreilyschool.com and google.com examples). In the MX lines, you'll see a number directly before the host name of the server. This number indicates the priority of the given server. The lower the number, the higher the priority. The highest priority server will be tried first, and if that fails, mail will then be sent to the next higher priority server and so on, until it is successfully sent or the list of servers is exhausted.

Before we introduce any more record types, we'll need to learn about the tool we use to query all record types: **dig**. Dig allows you to perform a DNS lookup and specify one or more record types to return. The format of the command is:

OBSERVE:

```
dig host recordtype
```

Let's make our first dig query using a familiar record type, the A record:

INTERACTIVE SESSION:

```
[username@username-m0 ~]$ dig google.com A

; <<>> DiG 9.7.3-P3-RedHat-9.7.3-8.P3.el6_2.2 <<>> google.com A
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 20502
;; flags: qr rd ra; QUERY: 1, ANSWER: 11, AUTHORITY: 4, ADDITIONAL: 4

;; QUESTION SECTION:
;google.com.                IN      A

;; ANSWER SECTION:
google.com.                 300     IN      A       74.125.224.133
google.com.                 300     IN      A       74.125.224.134
google.com.                 300     IN      A       74.125.224.135
google.com.                 300     IN      A       74.125.224.136
google.com.                 300     IN      A       74.125.224.137
google.com.                 300     IN      A       74.125.224.142
google.com.                 300     IN      A       74.125.224.128
google.com.                 300     IN      A       74.125.224.129
google.com.                 300     IN      A       74.125.224.130
google.com.                 300     IN      A       74.125.224.131
google.com.                 300     IN      A       74.125.224.132

;; AUTHORITY SECTION:
google.com.                 216604  IN      NS      ns3.google.com.
google.com.                 216604  IN      NS      ns4.google.com.
google.com.                 216604  IN      NS      ns1.google.com.
google.com.                 216604  IN      NS      ns2.google.com.

;; ADDITIONAL SECTION:
ns1.google.com.             15102   IN      A       216.239.32.10
ns2.google.com.             15102   IN      A       216.239.34.10
ns3.google.com.             187530  IN      A       216.239.36.10
ns4.google.com.             187530  IN      A       216.239.38.10

;; Query time: 51 msec
;; SERVER: 172.16.0.1#53(172.16.0.1)
;; WHEN: Fri Feb 24 10:20:01 2012
;; MSG SIZE rcvd: 340
```

That's a lot of information to parse! Fortunately, dig splits the code into sections to make it easier to read. Take a look at the QUESTION SECTION first. It tells us what we've asked for, in this case, an A record. The next section is the ANSWER SECTION. This list looks a lot like the list you got back when you ran the **host** command against



google.com. However, it gives us some information that **host** did not. After the host name listed on each line, you see the number "300." This is the *Time To Live* or *TTL*; it's a length of time in seconds. In order to understand what the TTL tells us, we need to understand how DNS caching works.

One DNS server can query another DNS server that acts on behalf the domain of the host you're seeking. The server that acts for the target domain is called the *authoritative DNS server* for that domain. When your DNS server receives a response from the authoritative DNS server for the domain you are querying, it not only responds to your query, it also stores the information in its own database. This is called *caching*. Caching allows your DNS server to respond to any additional queries for that host without having to make a query of the authoritative server for that domain. It saves time and reduces load on authoritative DNS servers. The cached result remains cached until the entry has exceeded its TTL. We'll discuss DNS TTLs in depth in the next course when we learn about setting up a DNS server.

Moving on, we see the **AUTHORITY SECTION**. This section lists the authoritative DNS servers for the domain you've queried. As you can see in this example, it's possible to have multiple authoritative DNS servers for a particular domain. The IP addresses are included in the **ADDITIONAL SECTION**, which follows the **AUTHORITY SECTION**. Much like MX records, NS (**N**ame **S**erver) records contain a host name, not an IP address. That's the reason that the **ADDITIONAL SECTION** contains the A records that correspond to the hosts in the NS and MX records. In this example, we don't see an MX record in the **ADDITIONAL SECTION**, but if you perform an MX query on a domain, you see A records for the mail exchangers (highlighted in **red** here):

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ dig google.com MX

; <<>> DiG 9.7.3-P3-RedHat-9.7.3-8.P3.el6_2.2 <<>> google.com MX
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 12484
;; flags: qr rd ra; QUERY: 1, ANSWER: 5, AUTHORITY: 4, ADDITIONAL: 9

;; QUESTION SECTION:
;google.com.                IN      MX

;; ANSWER SECTION:
google.com.                 600     IN      MX      40 alt3.aspmx.1.google.com.
google.com.                 600     IN      MX      50 alt4.aspmx.1.google.com.
google.com.                 600     IN      MX      10 aspmx.1.google.com.
google.com.                 600     IN      MX      20 alt1.aspmx.1.google.com.
google.com.                 600     IN      MX      30 alt2.aspmx.1.google.com.

;; AUTHORITY SECTION:
google.com.                 309107  IN      NS      ns3.google.com.
google.com.                 309107  IN      NS      ns4.google.com.
google.com.                 309107  IN      NS      ns1.google.com.
google.com.                 309107  IN      NS      ns2.google.com.

;; ADDITIONAL SECTION:
aspmx.1.google.com.        293     IN      A       74.125.53.26
alt1.aspmx.1.google.com.   293     IN      A       74.125.45.26
alt2.aspmx.1.google.com.   293     IN      A       74.125.113.26
alt3.aspmx.1.google.com.   293     IN      A       173.194.67.26
alt4.aspmx.1.google.com.   293     IN      A       74.125.79.26
ns1.google.com.            40309   IN      A       216.239.32.10
ns2.google.com.            40309   IN      A       216.239.34.10
ns3.google.com.            40309   IN      A       216.239.36.10
ns4.google.com.            40309   IN      A       216.239.38.10

;; Query time: 41 msec
;; SERVER: 172.16.0.1#53(172.16.0.1)
;; WHEN: Mon Mar 5 15:03:24 2012
;; MSG SIZE rcvd: 352
```

As I mentioned earlier, it's possible to look up a host name based on an IP address. Give it a try using the **host** command to look up 199.27.144.89—you'll get "www.oreillyschool.com." This is called a *reverse lookup*. Now, try using dig to do the same thing. It doesn't work, does it? In the **QUESTION SECTION**, you'll see that dig has made a query for an A record, but A records are used to map host names to IP addresses, not the other way around. There is a

special kind of DNS record for mapping IP addresses to host names: the PTR record. Try using dig to make a PTR query for 199.27.144.89. Still no answer! What's going on here? Let's take a step back and discuss the inner workings of DNS queries.

When you ask for the A record for "www.oreillyschool.com," your DNS server doesn't know who the authoritative DNS server is for the oreillyschool.com domain. In order to find out that information, the DNS server must query the *root servers* first. This group of servers holds DNS information for all of the *top-level domains*, such as ".com," ".net," ".gov," and so on. Next, because "www.oreillyschool.com" is a part of the ".com" top-level domain, the root DNS server will tell your resolver to query a DNS server that is responsible for the ".com" domain. Then your resolver can query the ".com" server for the NS record for the "oreillyschool" domain. Now your resolver knows the authoritative nameserver for the "oreillyschool.com" domain, so it can make a query of that server for the host "www." The information that your resolver obtains in the process of resolving your query is cached, so the next time you want to look something up in the ".com" domain, you won't have to ask the root servers who's responsible for ".com" first.

So what does this have to do with reverse lookups? Excellent question! You may have noticed that the series of DNS queries involved in resolving a host name starts at the broadest level at the right end of the host name, working to the most specific level at the left end of the host name. In order to make reverse lookups fit into this schema, the IP address must be reversed. That's not all, though. On the right side of the query, you must also append ".in-addr.arpa." The "arpa" domain is another top-level domain, and the "in-addr" subdomain is used specifically for executing IPv4 reverse DNS lookups. Once the query reaches the "in-addr" authoritative servers, it begins locating the authoritative nameservers for increasingly more specific IP address blocks. In our case, these would be 199.in-addr.arpa, 27.199.in-addr.arpa and then finally, 144.27.199.in-addr.arpa. When the resolver locates that last authoritative nameserver, then it can inquire about where 199.27.144.89 (or 89.144.27.199.in-addr.arpa) points. Here's what it looks like when we ask for the PTR record for 89.144.27.199.in-addr.arpa:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ dig 89.144.27.199.in-addr.arpa PTR

; <<>> DiG 9.7.3-P3-RedHat-9.7.3-8.P3.el6_2.2 <<>> 89.144.27.199.in-addr.arpa PTR
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 62613
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 2, ADDITIONAL: 2

;; QUESTION SECTION:
;89.144.27.199.in-addr.arpa.      IN      PTR

;; ANSWER SECTION:
89.144.27.199.in-addr.arpa. 600 IN      PTR      www.oreillyschool.com.

;; AUTHORITY SECTION:
144.27.199.in-addr.arpa. 600 IN      NS       ns2.useractive.com.
144.27.199.in-addr.arpa. 600 IN      NS       ns1.useractive.com.

;; ADDITIONAL SECTION:
ns1.useractive.com.      600 IN      A        199.27.148.66
ns2.useractive.com.      600 IN      A        199.27.148.67

;; Query time: 27 msec
;; SERVER: 172.16.0.1#53(172.16.0.1)
;; WHEN: Mon Mar 5 15:45:04 2012
;; MSG SIZE rcvd: 158
```

Take a few minutes and use **dig** to go through the same process manually that your resolver would go through automatically when it tries to locate the PTR record for "208.201.239.100." That is, first query the "arpa" top level domain for the NS record for "in-addr.arpa", then ask the "in-addr.arpa" nameserver for the NS record for "208.in-addr.arpa" and so on. Practice using and reading the output of dig. Remember, there may be multiple authoritative nameservers for a particular domain.

Another useful tool is **nslookup**. In addition to the servers that you have defined, it gives you the option of querying a specific DNS server that may not be defined in your resolv.conf file. By default, running **nslookup** with a hostname causes it to behave just like any of the other DNS utilities we've discussed so far (that is, it queries your configured DNS server). However, when invoked as **nslookup hostname dns-server**, nslookup will use the server you specify. First, we will try using nslookup without specifying a nameserver.

### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ nslookup www.oreillyschool.com
Server:          172.16.0.1
Address:         172.16.0.1#53
```

#### Non-authoritative answer:

```
www.oreillyschool.com canonical name = blogs.oreilly.com.
Name:   blogs.oreilly.com
Address: 67.222.96.148
```

As you can see in our example, in addition to the answer it received, **nslookup** tells you which DNS server it used to complete the query. A particularly interesting bit of output has been highlighted in **red**. By querying a DNS server other than the O'Reilly School of Technology's DNS server, we have gotten a *Non-authoritative answer*. Now would be a great time to figure out what the authoritative nameservers for the oreillyschool.com domain are! We can then use this information with nslookup to get an authoritative answer. We will use dig again for this purpose, but this time we will be asking for NS records. The important information is highlighted in **red**:

### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ dig oreillyschool.com NS

; <<>> DiG 9.7.3-P3-RedHat-9.7.3-8.P3.el6_2.2 <<>> oreillyschool.com NS
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 62613
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 3

;; QUESTION SECTION:
;oreillyschool.com.          IN      NS

;; ANSWER SECTION:
oreillyschool.com.         600     IN      NS      ns1.useractive.com.
oreillyschool.com.         600     IN      NS      ns2.useractive.com.
oreillyschool.com.         600     IN      NS      ns3.useractive.com.

;; ADDITIONAL SECTION:
ns1.useractive.com.        78839   IN      A        199.27.148.66
ns2.useractive.com.        78839   IN      A        199.27.148.67
ns3.useractive.com.        78839   IN      A        54.225.197.215

;; Query time: 13 msec
;; SERVER: 172.16.0.1#53(172.16.0.1)
;; WHEN: Thu Oct 17 12:58:07 2013
;; MSG SIZE rcvd: 148
```

Looking at our dig query here, we have a list of nameservers in the ANSWER SECTION. Luckily, dig also gives us the A records for each nameserver in the ADDITIONAL SECTION, so we can see what their IP addresses are. We can take this information and plug it in to an nslookup command to find an authoritative answer:

### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ nslookup www.oreillyschool.com 199.27.148.66
Server:          199.27.148.66
Address:         199.27.148.66#53

www.oreillyschool.com canonical name = blogs.oreilly.com.
Name:   blogs.oreilly.com
Address: 67.222.96.148
```

This time we do not get a message about having a non-authoritative answer. Now try using nslookup to query

google.com without specifying an alternate DNS server, then try using dig to find one of Google's nameservers to see if you can get an authoritative answer instead. Remember to do your assignments and we will see you in the next lesson!

*Copyright © 1998-2014 O'Reilly Media, Inc.*



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

# Package Management and Yum

---

Most modern Linux distributions allow administrators to add software to their systems using *packages*. Package management systems like RPM (which we'll cover a bit later in the course) provide a method for administrators and software distributors to keep track of the software that's installed and which *dependencies* must be met in order for new software to be installed. In this lesson, we'll learn about a tool called *Yum* that can be used to locate and install RPM packages and dependencies.

## An Introduction to Package Management

Software packages in Linux are designed to allow software to be installed, updated, and removed efficiently. While the primary function of the software package is to contain the files that are directly associated with a piece of software, the package generally also contains information about any additional packages that must be installed in order for the software to work correctly. These additional required software packages comprise *software dependencies*. Modern package management tools have the ability to interpret these dependencies from a package and install the required packages automatically.

Package management tools that ship with Linux distributions generally have an entire infrastructure associated with their operation. The distribution you use will most likely have a set of packages maintained for it by the distributor. These packages are usually made available online in a *repository*. In order to spread the load of serving these packages up to end users, these repositories will be *mirrored* to several locations. This allows end users to download packages from the nearest and/or best performing repository. The O'Reilly School of Technology maintains its own mirror of packages for the CentOS distribution, which is the distribution used on your Linux Learning Environment machine. Others can maintain repositories of software that isn't included in the standard distribution as well, which makes it much simpler to distribute software to end users.

Package management tools download packages from the repository (both the packages you have requested to install, and the packages required to satisfy dependencies) and install them in the proper order. At the same time, they update a database on your system that contains a list of all of the packages you have installed. This database will be used for dependency checks and determining upgrades. Package management tools also give you the ability to query software repositories to determine which packages they contain. Packages in the repository can then be searched using a name or a short description of the functionality of the software, allowing administrators to locate the software they need.

## The Yum Package Manager

In this course, we'll use the **yum** package management tool for downloading and installing software. This package manager is used by Red Hat Linux-based systems (including CentOS). In order to install software using yum, you will need root privileges, which means you'll be using `sudo`. The general format for using yum is:

OBSERVE:
<code>yum subcommand objects</code>

There are quite a few yum subcommands, but you'll primarily use **search**, **install**, **update**, **upgrade**, **remove**, and **clean**. We will start with the **search** subcommand, and use it to find a piece of software called "ssh" that allows us to log into other computers on the network. Root privileges are *not* required to use **yum** to query software repositories:

## INTERACTIVE SESSION:

```
[username@username-m0 etc]$ yum search ssh
Loaded plugins: fastestmirror
===== Matched: ssh =====
=====
libssh2.x86_64 : A library implementing the SSH2 protocol
libssh2.i686 : A library implementing the SSH2 protocol
libssh2-devel.i686 : Development files for libssh2
libssh2-devel.x86_64 : Development files for libssh2
libssh2-docs.x86_64 : Documentation for libssh2
openssh.x86_64 : An open source implementation of SSH protocol versions 1 and 2
openssh-askpass.x86_64 : A passphrase dialog for OpenSSH and X
openssh-clients.x86_64 : An open source SSH client applications
openssh-ldap.x86_64 : A LDAP support for open source SSH server daemon
openssh-server.x86_64 : An open source SSH server daemon
pam_ssh_agent_auth.i686 : PAM module for authentication with ssh-agent
pam_ssh_agent_auth.x86_64 : PAM module for authentication with ssh-agent
trilead-ssh2.noarch : SSH-2 protocol implementation in pure Java
trilead-ssh2-javadoc.noarch : Javadoc for trilead-ssh2
ksshaskpass.x86_64 : A KDE version of ssh-askpass with KWallet support
jsch.noarch : Pure Java implementation of SSH2
python-paramiko.noarch : A SSH2 protocol library for python
python-twisted-conch.x86_64 : SSH and SFTP protocol implementation together with client
s and servers
kdeutils.x86_64 : KDE Utilities
krb5-appl-clients.x86_64 : Kerberos-aware telnet, ftp, rcp, rsh and rlogin clients
krb5-appl-servers.x86_64 : Kerberos-aware telnet, ftp, rcp, rsh and rlogin servers
perl-Digest-BubbleBabble.noarch : Create bubble-babble fingerprints
pexpect.noarch : Pure Python Expect-like module
```

Some items match based on package name, some match in their description, and some don't seem to match at all. The package we're interested in shows up on the list as **openssh-clients.x86\_64**. Sometimes the brief description of the package found in the "yum search" output isn't enough. If you want more information, including a longer description, the size of the package, and the version, you can use the **info** subcommand. Before we install the openssh-clients package, let's take a look at its extended information. We don't need to use the full name of the package to do this. We can omit the *architecture* or *arch* string (x86\_64) at the end of the package name:

## INTERACTIVE SESSION:

```
[username@username-m0 ~]$ yum info openssh-clients
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Available Packages
Name      : openssh-clients
Arch      : x86_64
Version   : 5.3p1
Release   : 70.el6_2.2
Size      : 357 k
Repo      : updates
Summary   : An open source SSH client applications
URL       : http://www.openssh.com/portable.html
License   : BSD
Description : OpenSSH is a free version of SSH (Secure SHell), a program for logging
            : into and executing commands on a remote machine. This package includes
            : the clients necessary to make encrypted connections to SSH servers.
```

Installing this package will put the ssh client software on our machine. Let's go ahead and do that using the **install** subcommand. Just like the **info** subcommand, you don't need to use the full package name "openssh-clients.x86\_64." You'll rarely need to specify the architecture of the package that you want to install; yum will install the correct package for your architecture in most cases:

## INTERACTIVE SESSION:

```
[username@username-m0 etc]$ sudo yum install openssh-clients
...
Dependencies Resolved

=====
=====
Package                        Repository      Arch      Size      Version
=====
=====
Installing:
openssh-clients                updates        x86_64     357 k     5.3p1-70.el6_2.2
Installing for dependencies:
fipscheck                      base          x86_64      14 k     1.2.0-7.el6
fipscheck-lib                  base          x86_64      8.3 k     1.2.0-7.el6
openssh                         updates        x86_64     235 k     5.3p1-70.el6_2.2

Transaction Summary
=====
Install      4 Package(s)
Upgrade      0 Package(s)

Total download size: 614 k
Installed size: 1.7 M
Is this ok [y/N]: y
Downloading Packages:
-----
Total
                    5.1 MB/s | 614 kB      00:00
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction
  Installing      : fipscheck-lib-1.2.0-7.el6.x86_64      1/4
  Installing      : fipscheck-1.2.0-7.el6.x86_64         2/4
  Installing      : openssh-5.3p1-70.el6_2.2.x86_64      3/4
  Installing      : openssh-clients-5.3p1-70.el6_2.2.x86_64 4/4

Installed:
openssh-clients.x86_64 0:5.3p1-70.el6_2.2

Dependency Installed:
fipscheck.x86_64 0:1.2.0-7.el6      fipscheck-lib.x86_64 0:1.2.0-7.el6
openssh.x86_64 0:5.3p1-70.el6_2.2

Complete!
```

Some of the output from that command was removed to save space which is indicated by the ellipsis. As our example shows, installing "openssh-clients" requires the installation of three additional packages. These packages are installed in order to resolve dependencies associated with the openssh-clients package. Let's do a quick check to make sure that ssh is working properly. In this example, we'll use ssh to log into the cold.useractive.com server. Your username and password will be the same as the one you use for your Linux Learning Environment machine. Once you've confirmed that you can log in, you can log out at any time using **exit**:

## INTERACTIVE SESSION:

```
[username@username-m0 etc]$ ssh cold.useractive.com

username@cold.useractive.com's password:
Last login: Tue Feb 28 13:10:28 2012 from clustergw.useractive.com
cold1:~$ exit
logout
Connection to cold.useractive.com closed.
```

Pretty neat, huh? We'll discuss ssh in depth in the next lesson. For now though, we'll continue to focus on the package manager. Let's try removing a package using (you guessed it!) the **remove** subcommand. When your system was originally installed, we included a completely unnecessary package just so we could demonstrate package removal now. (We are crafty that way sometimes.) This package is called **cdrdao**. Normally you'd install this package so that you could burn optical discs on your machine, but since you don't have a CD or DVD burner installed in your virtual machine, it's totally useless. Go ahead and remove **cdrdao** with **yum** as shown:

## INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo yum remove cdrdao
Loaded plugins: fastestmirror
Setting up Remove Process
Resolving Dependencies
--> Running transaction check
---> Package cdrdao.x86_64 0:1.2.3-4.el6 will be erased
--> Finished Dependency Resolution

Dependencies Resolved

=====
=====
Package                        Arch      Size      Version
Repository                    Size
=====
Removing:
cdrdao                        x86_64    1.1 M     1.2.3-4.el6
@base
Transaction Summary
=====
=====
Remove      1 Package(s)

Installed size: 1.1 M
Is this ok [y/N]: y
Downloading Packages:
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction
  Erasing      : cdrdao-1.2.3-4.el6.x86_64
                                           1/1

Removed:
  cdrdao.x86_64 0:1.2.3-4.el6

Complete!
```

This package has now been removed from your system. If you remove a package that has configuration files associated with it, those files might be left on your system. By leaving those files in the system, the administrator won't



lose a configuration file that may be needed later. Even so, it's always a good idea to back up any configuration files you think you may need before you do a package removal. Or, you may want to purge configuration files for removed software manually to save space or to make sure that sensitive data isn't left lying around. Yum *never* removes user data, so you will not need to worry about inadvertently removing user-generated data during a package removal.

You're making great progress! More yum to come in the next lesson!

Copyright © 1998-2014 O'Reilly Media, Inc.



*This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.*

# More Yum Queries

In this lesson we'll cover a few more uses for yum, including getting more information about installed packages and updating the system.

## Yum List

Yum can list packages based on various criteria, as well as list information about selected packages. We'll start this lesson by listing the packages installed on your system using **yum list**. If you run this command as is, you may get far more information than you need. Issuing **yum list** prints out packages you have installed, as well as packages that are available for installation in your configured repositories—in our case, that's over 6000 packages. That much information is not very useful. We can narrow our focus by specifying an expression to match package names against, using the single- and multiple-character wildcards (**?** and **\***, respectively) to filter things a little bit:

### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ yum list openssh-*
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Installed Packages
openssh.x86_64                    5.3p1-70.el6_2.2      @updates
openssh-clients.x86_64           5.3p1-70.el6_2.2      @updates
Available Packages
openssh-server.x86_64            5.3p1-70.el6_2.2      updates
openssh-askpass.x86_64           5.3p1-70.el6_2.2      updates
openssh-ldap.x86_64              5.3p1-70.el6_2.2      updates
```

The output lists packages that are "Installed" and "Available." This gives you a good indication of which capabilities your machine has. Here we see that while we've installed the openssh-clients, we have not yet installed the openssh-server. That means you won't be able to ssh to your own machine. You can also list all the packages you have installed on your machine by running **yum list installed**. If you do that, you'll see a list of over 170 packages. This will also allow you to see whether your machine has a particular capability.

## RPM

Suppose you need information about where a package has installed various files on your system, because it has installed a binary or configuration file in a non-standard location. This is a good time to use the **rpm** command. This command was the original tool used for interacting with *Redhat Package Manager (RPM)* packages, as well as working with the RPM database. You can use the flags **-ql** (for query and list) to instruct RPM to list the locations of files that were installed by a particular RPM package. Let's try this for the openssh-clients package:

## INTERACTIVE SESSION:

```
[username@username-m0 ~]$ rpm -ql openssh-clients
/etc/ssh/ssh_config
/usr/bin/.ssh.hmac
/usr/bin/scp
/usr/bin/sftp
/usr/bin/slogin
/usr/bin/ssh
/usr/bin/ssh-add
/usr/bin/ssh-agent
/usr/bin/ssh-copy-id
/usr/bin/ssh-keyscan
/usr/share/man/man1/scp.1.gz
/usr/share/man/man1/sftp.1.gz
/usr/share/man/man1/slogin.1.gz
/usr/share/man/man1/ssh-add.1.gz
/usr/share/man/man1/ssh-agent.1.gz
/usr/share/man/man1/ssh-copy-id.1.gz
/usr/share/man/man1/ssh-keyscan.1.gz
/usr/share/man/man1/ssh.1.gz
/usr/share/man/man5/ssh_config.5.gz
```

You can use the **rpm** command for a variety of tasks, from installing and removing packages to listing information about packages, but it's generally not as user-friendly as **yum**. Also, **rpm** doesn't provide a method to retrieve packages from repositories or to resolve dependencies automatically, so you probably won't use it too often. Still, some tasks, like listing the files in a package, for example, can't be accomplished with **yum**. Fortunately, you'll have both **rpm** and **yum** at your disposal.

## Yum Clean

Now let's talk about the **clean** subcommand. In order to see some of the effects of **clean**, let me first introduce a handy little utility known as **df**. The **df** command displays disk usage on your host in actual bytes used on the disk and as a percentage of total space. By default, the output of **df** is in kilobytes:

## INTERACTIVE SESSION:

```
[username@username-m0 ~]$ df
Filesystem      1K-blocks      Used Available Use% Mounted on
/dev/ubda         1032088    561012    418648   58% /
none              6784352   5237624   1202104   82% /repos
none             419430400 290629952 128800448   70% /home/username/.handin
```

(Your numbers are probably different.) Try running **df** with the **-h** flag. It should be easier to read now. The **-h** flag tells **df** to print the output in "human readable format"—it prints the sizes in sensible units (megabytes, gigabytes, and so on.). Later when we use **df** to demonstrate how the **clean** subcommand in **yum** works, we'll want the output in kilobytes though.

Now, let's move on to the **clean** subcommand itself. You'll probably invoke the **clean** subcommand with the **all** option most of the time. An important reason to run **yum clean all** is to clear out the cached list of packages that are available for installation. When new versions of packages become available, the system does not become aware of them until it rereads the list of packages from its configured repositories. The package manager won't reread this list from the repository if it has a locally cached list of packages. Running **yum clean all** removes this cached list.

Another good reason to run the occasional **clean all** is to remove unneeded files from yum's cache directory, and free up disk space. Downloaded package files use up the most disk space in yum's cache. When a package is installed, the files contained within that package are written to the disk, but the package file itself is not removed afterward, even though it's no longer needed. Running **yum clean all** removes these files, along with other pieces of cached data. Run **df** on your system, followed by **yum clean all** (you must use **sudo** to do this), and then run **df** again; compare the amount of space used on the **/dev/ubda** disk. You'll see that a significant amount of space has been made available:

## INTERACTIVE SESSION:

```
[username@username-m0 ~]$ df -k
Filesystem      1K-blocks      Used Available Use% Mounted on
/dev/ubda        1032088      590192      389468   61% /
none            6784352      5237624      1202104   82% /repos
none           419430400    290629952    128800448   70% /home/username/.handin
[username@username-m0 ~]$ sudo yum clean all
Loaded plugins: fastestmirror
Cleaning repos: base extras ostum1 updates
Cleaning up Everything
Cleaning up list of fastest mirrors
[username@username-m0 ~]$ df -k
Filesystem      1K-blocks      Used Available Use% Mounted on
/dev/ubda        1032088      561012      418648   58% /
none            6784352      5237624      1202104   82% /repos
none           419430400    290630016    128800384   70% /home/username/.handin
```

Finally, let's address the **update** subcommand. As the name implies, this command updates the packages on your system. You can upgrade individual packages to new versions by specifying them on the command line, or you can run **yum update** to update all installed packages with updates on your system. Keeping the packages on your system up-to-date is an important part of system administration. Often, security vulnerabilities are found in important pieces of software, and running out-of-date versions of packages can lead to serious security issues. When you run a system update, you'll want to run **yum clean all** first, in order to clear out yum's cached list of packages and ensure that you're getting the most recent list of packages. Go ahead and run **yum update** to update your system:

## INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo yum clean all
Loaded plugins: fastestmirror
Cleaning repos: base extras ostum1 updates
Cleaning up Everything
Cleaning up list of fastest mirrors
[username@username-m0 ~]$ sudo yum update
...output...
```

This operation may or may not update any packages, depending on how long it's been since your machine was installed. Often this operation will not require your system to reboot, but if it installs a new kernel you'll need to reboot in order to start the new kernel.

Go ahead and experiment using these commands on your own until you feel confident in your ability to use them. Complete any homework or projects and I'll see you in the next lesson where we'll take on ssh...

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# SSH: The Secure Shell

One of the great advantages of having a machine connected to the network is that you don't have to be sitting in front of it in order to work on it. Before the days of computer networking, a "remote" login meant having a dumb terminal (a display terminal that provided keyboard input and screen output, but no data processing capability) connected to a computer sitting far away, using a long serial line. Due to the physical restrictions on serial line transmissions, this meant you could be, at most, several hundred feet away. When computers started to be networked, one of the first tasks early computer scientists worked out was creating a way for users to log in over the network. Over the years, many different protocols and methods have been used. Our current best option is ssh, the **secure shell**.

## The Secure Shell

The ssh protocol was invented to correct many of the problems associated with earlier protocols, such as *telnet*. The major advantage that ssh has over telnet is that ssh encrypts your data, which makes it much more difficult for people to eavesdrop on your transmissions. While we'll be concentrating on the usage of *OpenSSH*, there are many different programs that implement the ssh protocol on the client and server sides. Earlier, we installed the OpenSSH client suite which allows you to log into other networked machines and even transfer files to them. Later in this course, we'll install the OpenSSH server which will allow you to log into your Linux Learning Environment machine from elsewhere.

The main component of the OpenSSH client software suite is the **ssh** command. **ssh** allows you to log into other servers remotely. As a system administrator, you will probably spend most of your time logging into remote machines from your desk, rather than working directly on the console. Managing an ssh client is now required of every sysadmin. Since you've already seen the primary function of the ssh client, we will spend some time here focusing on one of the other features of the ssh client, as well as a couple of other tools that ship with the OpenSSH client suite.

If you've never tried to ssh *from* cold, you'll need to create the **.ssh/** directory. Open a new Terminal session and log in to cold, and then create the **.ssh/** directory.

### INTERACTIVE SESSION:

```
cold1:~$ mkdir .ssh
cold1:~$ ls -al .ssh
total 48
drwx-----  2 username webusers  4096 Nov  5 13:29 .
drwx----- 33 username webusers 16384 Nov  3 17:14 ..
```

## SSH Keys

The SSH protocol allows you to use a *public/private key pair* instead of a password in order to authenticate on remote machines. A public/private key pair is a set of cryptographic keys that can be used to encode and decode information. As the names of the keys imply, one is intended to be shared with the public and can, in theory, be read by anyone without compromising security. Alternatively, the private key must be kept secret and should only be readable by you. The public key cannot be used to derive the private key. Any data that is encrypted with the public key can then be decrypted using the private key. OpenSSH uses this property to authenticate you based on a public/private key pair. When you try to initiate a connection, the OpenSSH server on the remote end uses your public key to encrypt a message. It sends that message back to you on your local machine where your ssh client decrypts the message using your private key. Next, your client sends the decrypted message back to the server. Then the server compares the message it received, to the unencrypted version of the message it sent you. If they match, it confirms that the public/private key pair match, and you are authorized. If they do not match, then you do not have the private key that matches the public key on the other end and your connection request is denied.

In practice, you will use a tool that ships with the OpenSSH client suite to generate this key pair. Then you'll distribute your public key to machines that you plan on logging into in the future. Specifically, the contents of the public key are added to the file **~/.ssh/authorized\_keys** on the remote host. You can add a passphrase to your private key in order to keep anyone else from taking it and pretending to be you. If you store your private key on a machine that others may log into, using the passphrase is the better option. If you plan to store your private key on a machine that only you can log into, security is less of a concern and you can omit a passphrase. If you supply a passphrase, you will be prompted to enter it to unlock your private key for use by the ssh client. One possible application of an ssh private key without a password would be to allow automated scripts to log into remote machines and perform actions. We'll cover this kind of functionality more in the scripting course, but for now we'll focus on using ssh keys for authentication. Let's generate an ssh public/private key pair using the **ssh-keygen** program. For now, the defaults are sufficient, so just press

Enter when prompted for a filename to save the key in—but in general, you'll want supply a passphrase:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/username/.ssh/id_rsa):
Created directory '/home/username/.ssh'.
Enter passphrase (empty for no passphrase): (doesn't appear as you type)
Enter same passphrase again: (doesn't appear as you type)
Your identification has been saved in /home/username/.ssh/id_rsa.
Your public key has been saved in /home/username/.ssh/id_rsa.pub.
The key fingerprint is:
5d:e3:03:74:0b:18:4c:f2:c0:10:90:5c:37:69:09:b3 username@username-m0.unix.userac
tive.com
The key's randomart image is:
+--[ RSA 2048 ]-----+
| ..+*==ooo .      |
| o  +==o. o .     |
|   E. . . +       |
|   . . + .        |
|   S . o          |
|                   |
|                   |
|                   |
|                   |
+-----+
```

You now have a public/private key pair in the directory ~/.ssh. Take a look and see what's there:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ ls -l .ssh
total 8
-rw----- 1 username username 1743 Mar 13 14:44 id_rsa
-rw-r--r-- 1 username username 424 Mar 13 14:44 id_rsa.pub
```

Notice the permissions. The file "id\_rsa.pub," your public key, is readable by everyone. The file "id\_rsa," your private key, is readable only by you. This is exactly what we want. Now we just need to transfer your public key to a remote host, cold.useractive.com, to test out the key pair you just generated. We will use a neat trick to send the file from one host to another using ssh:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ cat .ssh/id_rsa.pub | ssh cold.useractive.com cat - ">
>" .ssh/authorized_keys
username@cold.useractive.com's password:
```

Let's take a closer look:.

#### OBSERVE:

```
cat .ssh/id_rsa.pub | ssh cold.useractive.com cat - ">>" .ssh/authorized_keys
```

We used **cat** to read the file **.ssh/id\_rsa.pub**, and we **redirected cat's standard output stream to ssh**. ssh can execute an arbitrary command on the remote host, rather than executing a shell, so we let it. In this case, we **request that cat be executed on the remote host**. Giving cat the argument **-** on the remote host, instead of giving cat a filename, instructs it to read from its standard input stream, which will come from the standard output stream of the cat on the localhost via ssh. Does that make sense. Pause and let that sink

in for a second. Finally, we **tell cat, on the remote host, to append its standard output stream to the file .ssh/authorized\_keys**. The double quotation marks are necessary here in order to keep the local shell from parsing the >> operator and redirecting the output to a local file. Fortunately for us, there are less complicated ways to transfer files to and from remote hosts.

## Other OpenSSH Tools

The OpenSSH suite includes a couple of other tools that are useful for using and administering Linux systems: **sftp** and **scp**. Both of these tools can transfer files securely between hosts, but their interfaces operate in vastly different ways. If you've used ftp before, then you're also familiar with the workings of sftp. It's similar to the way scp operates like regular cp.

### SFTP

The **Secure File Transfer Protocol (sftp)** tool that comes with OpenSSH works the same way as regular FTP, except that all of your traffic is encrypted using the SSH protocol. This is important because the FTP protocol sends all data in plain-text, meaning that anybody who can see your packets on the network can see exactly what you're doing. As with telnet, FTP may allow an intruder to steal your password without much difficulty. That's why you should *always* use SFTP rather than FTP if you have the option.

To connect to another host with sftp, you run **sftp remote\_host**. Go ahead and connect to cold.useractive.com using sftp now:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sftp cold.useractive.com
Connecting to cold.useractive.com...
Enter passphrase for key '/home/username/.ssh/id_rsa':
sftp>
```

When you see the "sftp>" prompt, you're connected. Several of the commands available within the sftp shell are similar to commands available in bash. For example, **ls** lists files on the remote machine, **cd** changes directories on the remote machine, and so on. Type **help** for a full list of available commands. Commands that act on the remote host such as **ls** and **cd** have locally acting counterparts like **lls** and **lcd**. The **lls** command, for example, will show you files in the present working directory on your local machine. This allows you to review a list of files that you may want to upload using sftp. The two most important commands though, are **get** and **put**. These commands allow you to download and upload files, respectively. Let's test the **put** command after we use another feature of sftp to create a file on the local machine:

#### INTERACTIVE SESSION:

```
sftp> !touch sftp_test
sftp> put sftp_test
Uploading sftp_test to /users/username/sftp_test
sftp_test                               100%    0    0.0KB/s   00:00
```

The **!** operator allows you to execute an arbitrary command on your local machine from the sftp shell. In this case, we used it to touch the file sftp\_test. The **get** command operates in a similar way. If you have any files in your home directory on cold, feel free to use them to try out the get command.

### SCP

**Secure cp (scp)**, like sftp, provides a more secure version of a standard command for use where security can be an issue. The general format of this command is **scp source\_file destination\_file**. It's possible for the source file, destination file, or both to reside on remote machines. When the file is located on your local machine, you specify the filename as you would normally specify it to cp. The remote file format is similar, but you must also include the host name of the remote machine and optionally, a username that will be used to log into the machine. The remote file specification will look like this: **username@hostname:/path/to/file**. Similar to ssh, if you don't specify a username, the username that you are currently logged in with on your local machine will be used. Let's use scp to copy a file from your Linux Learning Environment machine to the OST login host, cold.useractive.com.

## INTERACTIVE SESSION:

```
[username@username-m0 ~]$ touch scp_test
[username@username-m0 ~]$ scp scp_test username@cold.useractive.com:~/
Enter passphrase for key '/home/username/.ssh/id_rsa':
scp_test                                100%    0    0.0KB/s    00:00
```

### Note

As with cp, you can specify a directory as the destination for the file copy. In this case, our directory is "~/", which is shorthand for your home directory.

Feel free to ssh to cold and verify that the file has copied over. You can also copy entire directories with scp by specifying the "-r" flag, "r" standing for "recursive." This behavior is identical to that of regular cp, which can use the "-r" flag to copy directories as well.

You're getting pretty good at this stuff. Just two lesson left in the course! I'll see you in the next one...

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.



# SSH: The Secure Shell Server

---

Having the OpenSSH client tools installed on your machine is useful for connecting to other hosts, but without the OpenSSH server installed, you can't use SSH to connect to your own machine. In this lesson we'll cover the installation of the OpenSSH server and some related concepts.

## The Secure Shell Server

The OpenSSH server runs as a process in the background on a machine and it listens for incoming connection requests on a specified *TCP port*. You may recall from the networking concepts lesson that TCP, the **T**ransmission **C**ontrol **P**rotocol, provides a method for data packets to be directed to a numbered port. This functionality allows a *service*, such as the OpenSSH server, to listen for incoming connections without disrupting other services that are listening for connections. The standard port that ssh servers listen on is port 22. If you try to initiate an ssh connection from a client and do not specify a target port, the client will assume that port 22 is to be used. An extensive list of well known ports is available in the **/etc/services** file. Let's go ahead and use **yum** to install the **openssh-server** package:

## INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo yum install openssh-server
[sudo] password for dbasset1:
Loaded plugins: fastestmirror
Determining fastest mirrors
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package openssh-server.x86_64 0:5.3p1-70.el6_2.2 will be installed
--> Processing Dependency: libwrap.so.0()(64bit) for package: openssh-server-5.3p1-70.e
l6_2.2.x86_64
--> Running transaction check
---> Package tcp_wrappers-libs.x86_64 0:7.6-57.el6 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

=====
=====
Package                                Arch                                Size                                Version
Repository
=====
Installing:
openssh-server                        x86_64                                297 k                                5.3p1-70.el6_2.2
updates
Installing for dependencies:
tcp_wrappers-libs                    x86_64                                62 k                                7.6-57.el6
base
Transaction Summary
=====
=====
Install      2 Package(s)

Total download size: 359 k
Installed size: 780 k
Is this ok [y/N]: y
Downloading Packages:
-----
-----
Total                                3.6 MB/s | 359 kB      00:00
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction
  Installing : tcp_wrappers-libs-7.6-57.el6.x86_64                1/2
  Installing : openssh-server-5.3p1-70.el6_2.2.x86_64            2/2

Installed:
openssh-server.x86_64 0:5.3p1-70.el6_2.2

Dependency Installed:
tcp_wrappers-libs.x86_64 0:7.6-57.el6

Complete!
```

The ssh server is now installed. We can test to make sure it's working using ssh to connect to *localhost*. Localhost is a special hostname reserved for a special IP address, 127.0.0.1. This address is assigned to a *loopback* interface. On your machine this interface is called "lo." A loopback interface takes any packet that it receives and spits it right back

out. This means that any packets sent to the address 127.0.0.1 from your machine will also be received by your machine. So using your ssh client to connect to port 22 at the address 127.0.0.1 will connect to the local machine on port 22. Let's go ahead and connect to the ssh server that you just installed:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ ssh localhost
ssh: connect to host localhost port 22: Connection refused
```

That didn't work out so well. To see why, use `ps` to look for the ssh server's process, which should be called "sshd." It's not even running—that explains why we couldn't connect to it! If a service you expect to be able to connect to is not responding, check to find out whether it's actually running on the host. Unfortunately, when an ssh server is not running, it probably means you'll have to go to work directly on the machine's console. Fortunately, you are already logged into your machine's console, so no ssh server is necessary to connect to it. We can start the sshd process by using the **service** command. This handy command is used to control the services that will run on your machine. The general format of the command is **service service\_name action**, where *action* can be **start**, **restart**, **stop**, or **reload**. Let's use the **start** action to get our ssh server running:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo service sshd start

Starting sshd: [ OK ]
```

As long as you see an "OK" in the brackets, your ssh server should be running. Now let's try to connect to localhost again:

#### Connecting to localhost with ssh:

```
[username@username-m0 ~]$ ssh localhost
The authenticity of host 'localhost (127.0.0.1)' can't be established.
RSA key fingerprint is 44:51:fb:df:57:db:a3:76:df:24:bc:e0:e6:60:81:ab.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (RSA) to the list of known hosts.
username@localhost's password:
Last login: Wed Mar 14 11:01:22 2012
[username@username-m0 ~]$
```

Now your ssh server is working! But what's the deal with that RSA key fingerprint and known hosts stuff? This is another security feature that's built into ssh. Each ssh server will generate a public/private key pair for itself. The first time you connect to the server, you don't have its public key, so your ssh client asks you to confirm that you are connecting to the server that you think you are. When you confirm that you are, you download the ssh server's public key and store it in the **known\_hosts** file in the `~/.ssh` directory. When you connect to this server in the future, your ssh client uses this stored public key to verify the identity of the ssh server using the same method that we discussed earlier for using public/private key pairs to authenticate users. This security feature prevents something called a *man in the middle attack*. In this kind of attack, a hacker could redirect traffic intended for a legitimate ssh server to his own ssh server, where he can then trick users into providing their passwords. He can then collect these passwords and use them to get into the legitimate server. When you manage an ssh server, you have to make sure that the ssh server's private key is safe.

While you're logged into your own machine via ssh, issue the command `w`. The `w` command gives you information about who is logged in, how they logged in, and statistics about what they have done since they logged in:

## INTERACTIVE SESSION:

```
[username@username-m0 ~]$ w
11:10:46 up 5 days, 23:54, 2 users, load average: 0.00, 0.00, 0.00
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU   WHAT
username  tty0      -               11:01    0.00s  3.43s  0.29s  ssh localhost
username  pts/0     localhost       11:01    0.00s  0.38s  0.00s  w
```

If you are logged into only one console on your machine, your output will look like that. In our example, we see that someone is logged in on the first console (tty0) and that person is currently running the command **ssh localhost**. On the next line, we see that same user, this time logged into pts/0 and running the **w** command. A single pts is called a *pty* or *pseudo-terminal*. A pseudo-terminal is given out to users for things like remote logins, rather than a tty, which is given out for logins on the local console. On the second user line, we see that **w** tells us where the user is logged in from, in this case, localhost. Finally, **w** gives us some statistics about when the user logged in (LOGIN@), how long it's been since the user did anything (IDLE), and the amount of CPU that the user is using. JCPU describes the total amount of CPU being consumed by jobs in that particular tty or pty, and PCPU tells us how much CPU is being used by the job specified in the WHAT column.

Now, add a new user called "testuser" to your system. If you need a reminder on how to do that, you can find instructions [here](#). When you have added testuser, open another console on your Linux Learning Environment machine and log in as yourself. Now we'll login as another user using ssh. Connect to your local machine as testuser using **ssh testuser@localhost**. Switch back to the other console where you're logged in and issue the **w** command again. You'll see something like this:

## INTERACTIVE SESSION:

```
[username@username-m0 ~]$ w
14:26:53 up 6 days, 3:10, 3 users, load average: 0.00, 0.00, 0.00
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU   WHAT
username  tty0      -               11:01    0.00s  4.21s  0.00s  w
username  tty1      -               11:13    8.00s  0.00s  0.00s  ssh testuser@localhost
testuser  pts/0     localhost       14:26    4.00s  0.32s  0.32s  -bash
```

Starting a connection with the ssh client using **ssh username@hostname** will request a login on the remote machine with that username. Congratulations! You have just successfully installed your first system service!

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.

# User Accounts

Welcome to the final lesson of the course. In this lesson we'll cover creating, removing, and managing user accounts on your system. Controlling who can or can't log into the system you are managing is an important part of system administration.

## The Passwd, Group, and Shadow Files

We'll start with the **passwd** (pronounced "password") file, located at **/etc/passwd**. This file holds information about all of the user accounts on the system. Let's have a look, shall we?

### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ head -n5 /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
```

At first glance it might not make much sense, so let's break it down and see what it all means. Each line corresponds to one user on the system, and each line is divided into seven fields. Each field is separated by a colon (:). Here's what each field means for the root user:

root	:	x	:	0	:	0	:	root	:	/root	:	/bin/bash
username		password		uid		gid		name		homedir		shell

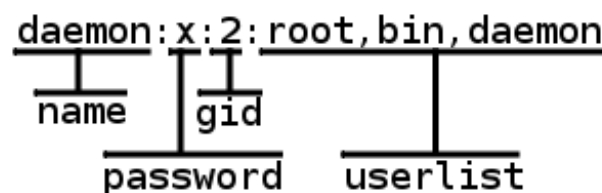
You're already familiar with the first field, **username**. It's the name that the user types into the system at login. This name is *not* how the system actually identifies the user though. The second field may contain **the user's encrypted password**, or an **x** to signify that the password is actually stored in the *shadow* file (we'll get to that in a minute). The third field, **uid**, is the *user id number*, which the system uses to identify the user. Next is the **group id number (gid)** field. Each user can belong to one or more groups, but every user must belong to one *primary* group. This is the number that defines a user's primary group membership. The mappings between group name and group id number are located in the **group** file. The fifth field, **name**, contains the full name of the user. Root's entry just says "root" because the root account is not an actual person. Following that is the **homedir**, or home directory, field. This tells the system which directory is the user's home directory. Finally, there is the **shell** field, which tells the system which shell the user should be given at login. Go over the entire passwd file using **less** and locate your own user entry.

Next, we'll look at the **/etc/group** file. This file contains the mappings between gid number and group name, as well as lists of users who belong to various groups. Like the passwd file, the group file contains one entry per line, with fields separated by a colon(:):

## INTERACTIVE SESSION:

```
[username@username-m0 ~]$ less /etc/group
mail:x:12:mail,postfix
uucp:x:14:uucp
man:x:15:
games:x:20:
gopher:x:30:
video:x:39:
dip:x:40:
ftp:x:50:
lock:x:54:
audio:x:63:
nobody:x:99:
users:x:100:smiller
```

Let's break down and define each field:



The **first field** is the name of the group. This name is mostly for display purposes. The **second field** is for the group password, which is a rarely used feature. Next is the **group id number**. Every group has a unique id number, and is used by the system to determine group membership and group ownership of resources. Finally, there is an optional comma-separated **list of users** who belong to the group.

Last we have the **shadow** file. On most recent Linux distributions, the shadow file, rather than the passwd file, contains the encrypted user passwords. By placing the encrypted passwords in a separate file, we can make the passwd file readable by all. The shadow file can only be read or modified by root. While it's possible to edit the shadow file directly, it's a good idea to use a tool like the **passwd** command to update user passwords and related settings. The format of the shadow file is identical to that of the passwd and group files; each line contains a user entry, and each field is separated by a colon (:). In most cases, you'll be concerned with only the first two fields in the file, the username and the encrypted password field. If the second field is empty, the user has no password. If the second field contains an exclamation point (!) or an asterisk (\*), then the corresponding user can't log in using the traditional shadow file method. Other methods may be available, but that is beyond the scope of this course. Additionally, if the second field contains an encrypted password that is prefixed with an exclamation point (!), that user account is locked. Removing the ! will unlock the account.

## Account Administration

Now that we've discussed the files involved with system and user accounts, we can work on adding, removing, and modifying user accounts. Let's start by adding a user to your system using the **useradd** command. We can use the **getent** command to get the new entry we've created from the passwd file:

## INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo useradd foobar
[username@username-m0 ~]$ getent passwd foobar
foobar:x:1045:1045::/home/foobar:/bin/bash
```

Your uid and gid will probably be different, but the rest of the entry will be identical. Keep in mind that you need to use super user privileges to add, delete, or modify a user. Look in **/home** and **/var/mail**; **useradd** not only adds an entry to **/etc/passwd**, but it also creates a home directory and a mail spool for the new user (don't worry if you aren't familiar with mail spools for now). If no primary group is specified, **useradd** also creates a group with the same name as the user and sets the user's primary group to this new group. Now, using **sudo**, look for the new user's entry in **/etc/shadow**:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo less /etc/shadow
...
foobar:!!:15523:0:99999:7:::
```

Your new user's account is **locked**! The **useradd** command creates the user but it does not set a password for the account. By default, the account is locked, and it must have a password set before the new user can log in. To set a password for your new user, use the **passwd** command. Your keystrokes will not appear on the screen, but **passwd** requests the new password twice so it can confirm that you typed it correctly:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo passwd foobar
Changing password for user foobar.
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
```

#### Note

If your password is deemed too simple, **passwd** displays a message to convey that, but still prompts you to re-enter it, and allows you to set the password. Search the internet for suggestions on [password strength](#).

Now try using **ssh** to log into your machine as the new user, using the password you just set. Once you have confirmed that the account is working, you can log out of the ssh session. Using the **groupadd** command, let's create a group and add foobar to it. To specify a group id number for your new group, use the **-g** flag (keep in mind that gid numbers of 999 and below are reserved for system groups, so you should use numbers of 1000 or larger); otherwise a gid will be chosen for you:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo groupadd -g 5555 testgroup
[username@username-m0 ~]$ getent group testgroup
testgroup:x:5555:
```

Your new group has been created, but it's empty for now. Create two more groups named **staff** and **employees**. If you want to specify gid numbers for these groups, remember to use numbers greater than or equal to 1000. After you have created your groups, add your new user to one of them. This is where the **usermod** command comes in handy. Usermod is used to add and remove users from groups, change user information, and lock accounts:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo usermod -G testgroup foobar
[username@username-m0 ~]$ groups foobar
foobar : foobar testgroup
```

The **groups** command lists the groups to which a user belongs. Now let's see what happens when you try to add the user to the other two groups you created:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo usermod -G staff,employees foobar
[username@username-m0 ~]$ groups foobar
foobar : foobar staff employees
```

Where did testgroup go?! Look at the **man** page for **usermod** to see if you can figure out why testgroup disappeared from the list of groups of which foobar was a member. Keep in mind that foobar's primary group membership is with the group "foobar" and all other groups are secondary. Take a look at **/etc/group** to see how it looks after adding groups and then adding a user to those groups. It's possible to edit the **group** file by hand to add a group or to add users to groups if you are more comfortable doing it that way. Feel free to add more users or groups, or add users to groups, until you're comfortable with the process. Also, experiment with the **groupmod** command to modify the names and gid numbers of some of the groups you've created. Be careful not to modify any system groups as there could be adverse consequences!

The last thing we'll cover in this lesson is removing users and groups from the system using the **userdel** and **groupdel** commands. Both are fairly straightforward to use, in fact, groupdel takes only one argument: the group to be deleted. Let's see it in action:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ groups foobar
foobar : foobar staff employees
[username@username-m0 ~]$ sudo groupdel employees
[username@username-m0 ~]$ groups foobar
foobar : foobar staff
```

In this example, when a group is removed, any user who is in that group has it removed from his or her list of secondary groups. There's really nothing else to the groupdel command. The userdel command has a couple of options. Just executing **userdel** and passing it a username will delete that user's account, but it does not remove that user's home directory or mail spool. Although sometimes you'll want to remove the user and all of his or her data. To do that, run **userdel** with the **-r** flag. Let's get rid of the user foobar. (I never liked him anyway):

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ sudo userdel -r foobar
```

Afterwards, this is the state of our system:

#### INTERACTIVE SESSION:

```
[username@username-m0 ~]$ ls /home
username
[username@username-m0 ~]$ ls /var/mail
username
[username@username-m0 ~]$ grep foobar /etc/passwd
[username@username-m0 ~]$
```

Fooobar and all of his data are gone. Take a moment and think about what you've done. When you use your super user powers, you can do things like remove someone's account and delete all of their data. What if you were removing an account and made a typo and removed the wrong one? It's *extremely* important to understand and respect the power you will have as an administrator with super user privileges. Anything from a mistyped **userdel** to an errant **rm** can destroy data and lead to down-time and loss of productivity. Always think before you type and, perhaps more importantly, keep solid backups!

And there you have it: the final lesson of the course. You've done some great work here, and I'm sure you'll do even more for your final project! Thanks for working through the course and I hope to work with you again.

The next System Administration course deals with installing and configuring additional important and widely used services.

Copyright © 1998-2014 O'Reilly Media, Inc.



This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.  
See <http://creativecommons.org/licenses/by-sa/3.0/legalcode> for more information.