



Linux for Researchers

Chapter 11: Ssh and Rsync

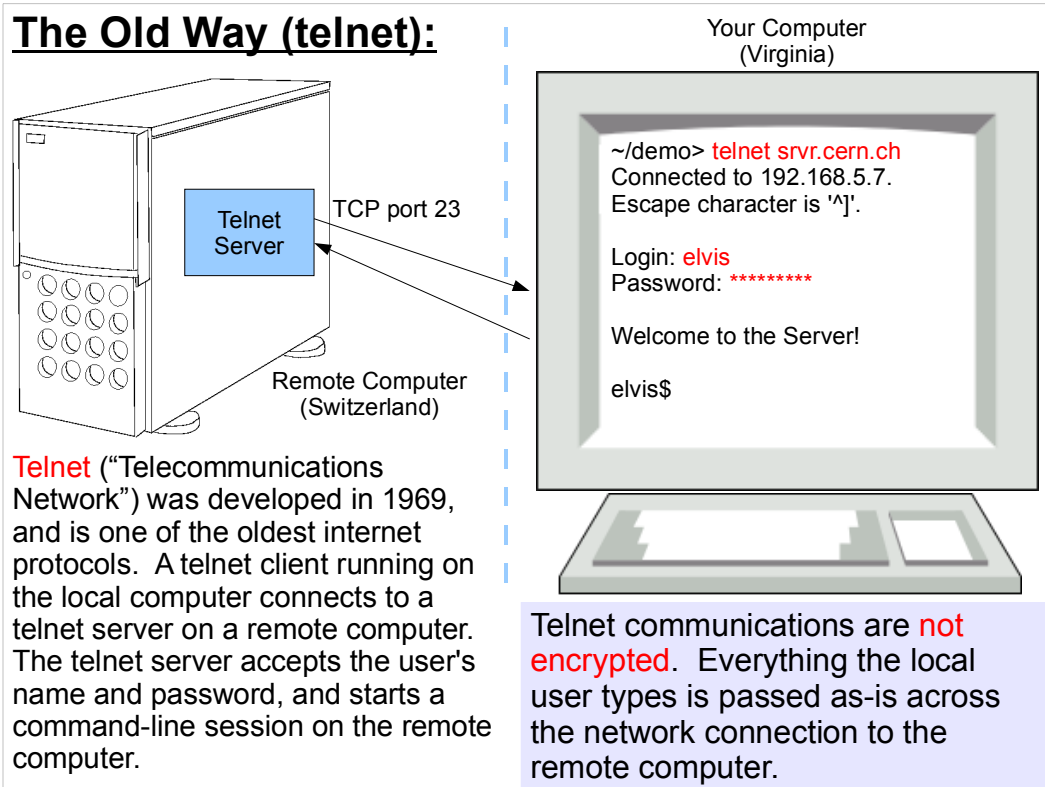
We've talked about using command-line tools on the local computer and, until now, we've just assumed that there were ways to execute the same commands on remote machines, too. Today we'll look in detail at how to log in to a remote computer and execute commands there, using the ssh protocol.

We'll also look at “rsync”, a very useful tool that uses ssh. When I finished these slides I realized that there was only one slide devoted to rsync, but don't let that fool you. Rsync is a very powerful and very useful command.

Part 1: Remote Shells



Computer networks have been around for a long time now. One of the first uses of these networks was the execution of commands on remote computers.



“rlogin” (sometimes invoked by “rsh”) is similar to telnet. It also creates an unencrypted command-line connection to a remote computer.

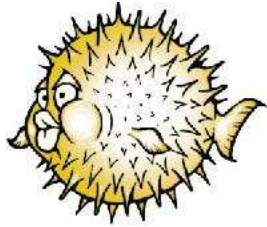
Note that nothing telnet transmits is encrypted, including your username and password when you log in.

Telnet was written at a time when computers were behemoths, and few people had access to them (or the networks that connected them). Just getting access to a remote computer was a Big Deal. Security was of little or no concern.

The New Way (ssh):

In 1995, Finnish researcher **Tatu Ylönen** wrote “ssh” (“Secure Shell”) as a more secure replacement for telnet. This soon evolved into a proprietary commercial product.

In 1999, **Björn Grönvall** began with earlier, free versions of Ylönen's code and began writing what was to become OpenSSH, a completely open-source re-implementation of ssh. This is now the de facto standard ssh implementation, and is included in many operating systems.



OpenSSH uses a blowfish as its logo because “blowfish” is the name of the strong, public-domain encryption algorithm it uses (by default) to encrypt network traffic. See:

[http://en.wikipedia.org/wiki/Blowfish_\(cipher\)](http://en.wikipedia.org/wiki/Blowfish_(cipher))

blowfish was developed in 1993 by **Bruce Schneier**.

From 1969 to the 1990s, telnet was the standard way of getting a command-line connection to a remote computer. It still had little security, but the world had changed. Now many people had access to computers and the network. Following a breakin at his University, Tatu Ylönen finally got fed up with the lack of security in telnet, and wrote “ssh” as a replacement.

Some of the Advantages of ssh:

- **Encryption of all communications.**

Ssh makes use of public-key cryptography principles to securely encrypt all communication between client and server.

- **Ability to verify the identity of a server.**

This allows you to be sure you're really talking to the server you intended, and not some third party masquerading as that server.

- **Freedom from many security bugs of telnet.**

Telnet was written in an age when security was an afterthought, if it was thought of at all. Ssh was written from scratch, with security in mind from the beginning.

In addition, ssh provides these new features:

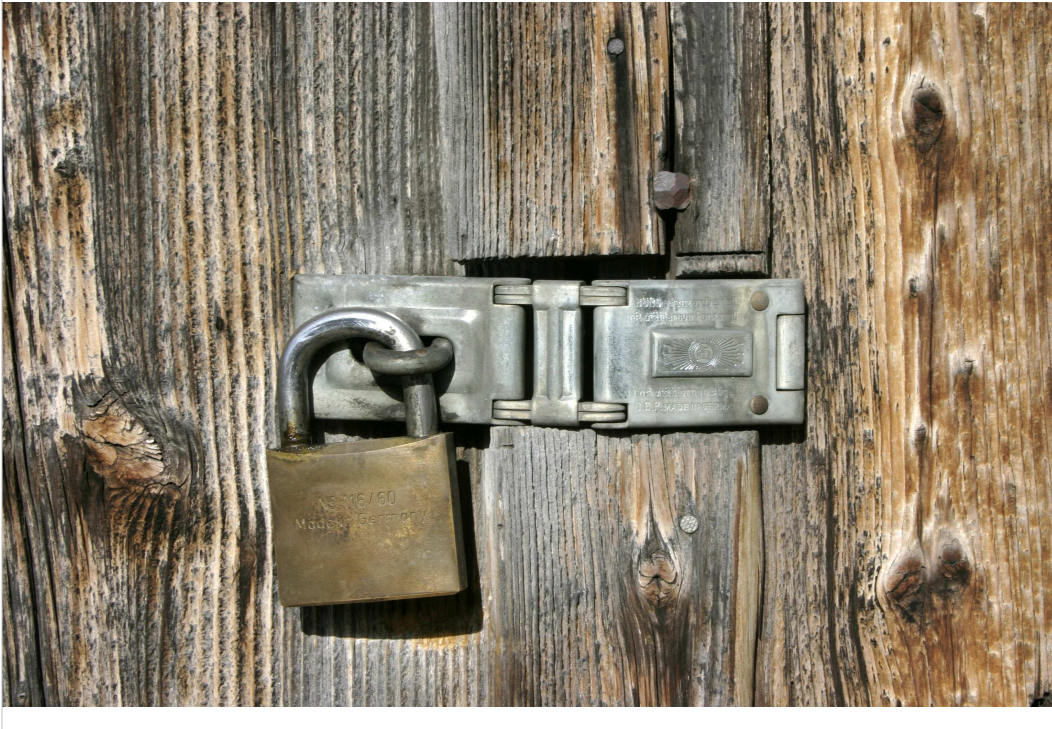
- **Secure File Transfer.**

The ssh protocol also includes mechanisms for **transferring files** securely (allowing it to replace the insecure “ftp” protocol).

- **Encrypted Tunnels.**

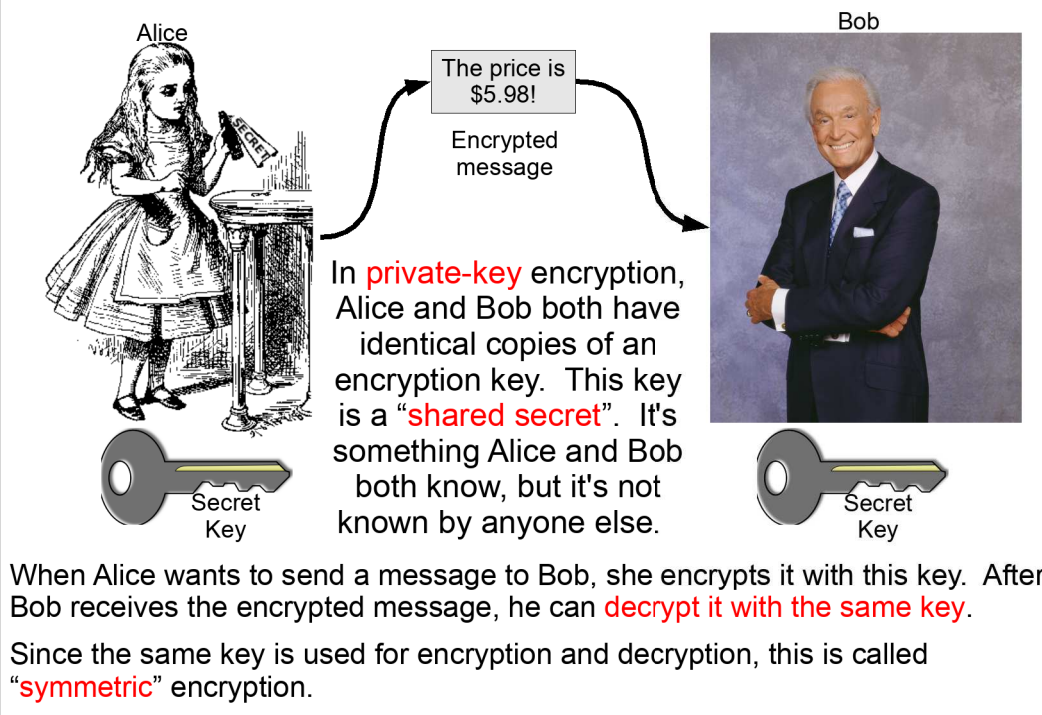
Ssh allows you to create encrypted **tunnels** between two hosts. Tunneling gives users a way to put a wrapper of encryption around other protocols that didn't natively support encryption.

Part 2: Some Encryption Principles



Before we go any further, we need to talk a little about encryption. The following is useful for understanding ssh, but it applies to a lot of other things, too: secure web connections, personal certificates, and e-mail encryption are some examples.

Private-Key (Symmetric) Encryption:



Symmetric encryption has been used since ancient times. It's something every schoolchild comes up with on his or her own, making up secret codes to share messages with friends.

As we'll see, it has some problems.

(Alice and Bob are common in cryptography examples. There's a T-shirt for crypto geeks that says “I know Alice and Bob's shared secret.”)

Some Problems with Private-Key Encryption:

- **Key distribution is hard, and possibly insecure.**

What if Alice is in Portugal and Bob is in Hong Kong? How do they initially get the secret key to each other? What if they need to change it later?

- **Anyone who steals the key can masquerade as Alice or Bob.**

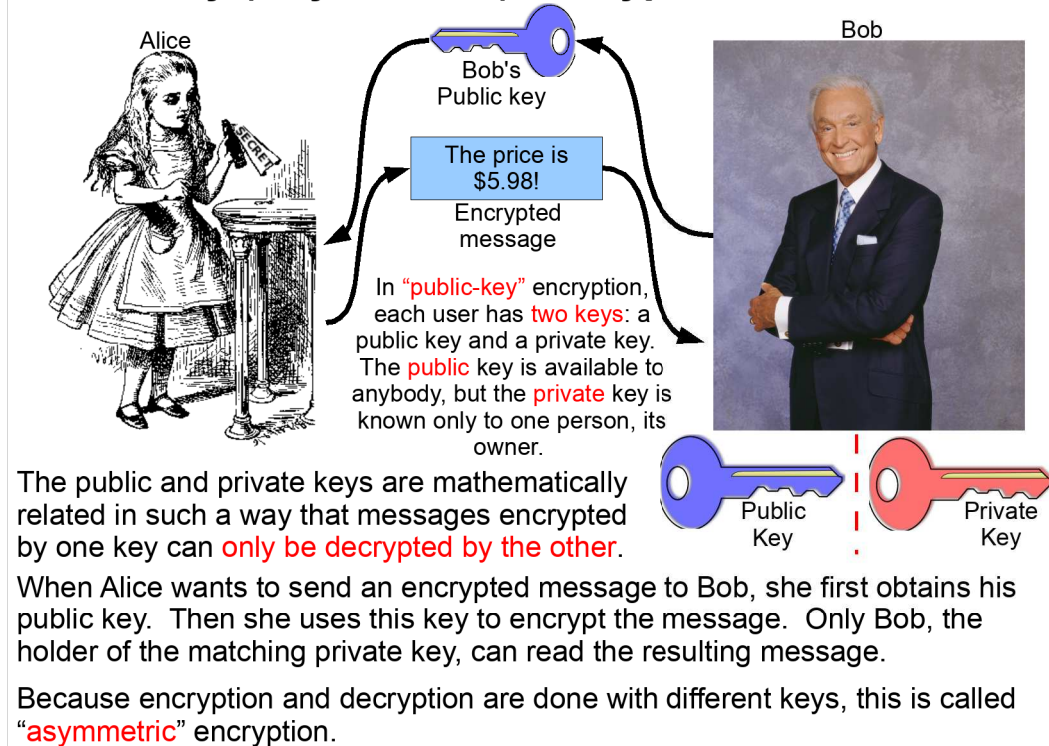
This system has no way of verifying the identity of the sender. If a third party obtains the key, he can send messages that appear to come from Alice or Bob.

- ***“A secret shared by two is compromised. A secret shared by three is no secret.”***

Can Alice really trust Bob to keep the key a secret? Can Bob trust Alice? And what if we have a whole organization full of people who all need to know the secret key?

Even if we ignore most of these problems, we still have to face up to the problem of key distribution. This started out with “How do we get keys to our secret agents scattered around the world?”. By the 1990s it had changed to “How do we get keys to all of the people on the internet who want to buy things from our e-commerce site?”

Public-Key (Asymmetric) Encryption:



In 1973-4, Clifford Cocks, James Ellis and Malcolm Williamson, working for British Intelligence, came up with a solution to the key distribution problem. It remained secret for a few years, though, until their techniques were rediscovered and published by researchers Whitfield Diffie and Martin Hellman in 1976, and Ron Rivest, Adi Shamir and Leonard Adleman in 1978.

Now, each individual could have a unique pair of keys, generated locally. There was no longer any need to distribute keys.

Advantages of Public-Key Encryption:

- **No need to distribute secret shared keys.**

Each person has his or her own public/private key pair, generated locally. Private keys always stay in the hands of their owner, and never need to be transmitted.

- **The sender's identity can be verified.**

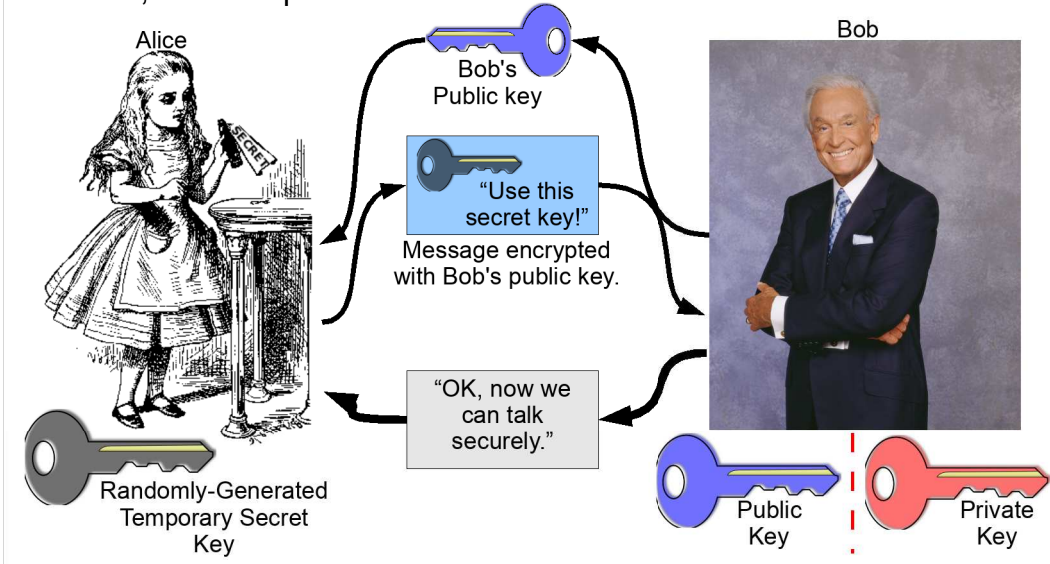
Since each sender has a unique public/private key pair, we can verify his/her identity.

- **Security risks are limited to a single user.**

Alice is responsible for the security of her keys, and Bob is responsible for the security of his keys. The theft of one person's key doesn't compromise the security of the other keys.

Key Distribution by Public-key Encryption:

Public-key cryptography is more computationally expensive than symmetric-key cryptography. Usually, public-key encryption is only used to **exchange a temporary secret key**, which is then used to conduct the remainder of the conversation via symmetric cryptography. This is what ssh does, for example.



Encryption and Legal Problems:

Ssh adoption was delayed for many years by legal problems in the United States. These problems fell into two categories:

- **Patent Restrictions:**

The earliest versions of ssh relied on the patented RSA public-key encryption algorithm. The patent-owner provided a free RSA “reference version” (RSAref), but only allowed it to be used for non-commercial applications. OpenSSH worked around this by substituting the freely-available DSA algorithm. The RSA patent expired in 2000, so this is no longer an issue.

- **Export Restrictions:**

U.S. export restrictions on cryptography were very strict at the time ssh was first written. They've relaxed considerably since then, and primarily apply to a list of seven “terrorist countries”. The law is still very confusing, though. Fortunately, OpenSSH is based in Canada, and the U.S. has no import restrictions on cryptography. To avoid possible legal problems, the OpenSSH project does not accept help from software developers in the U.S.

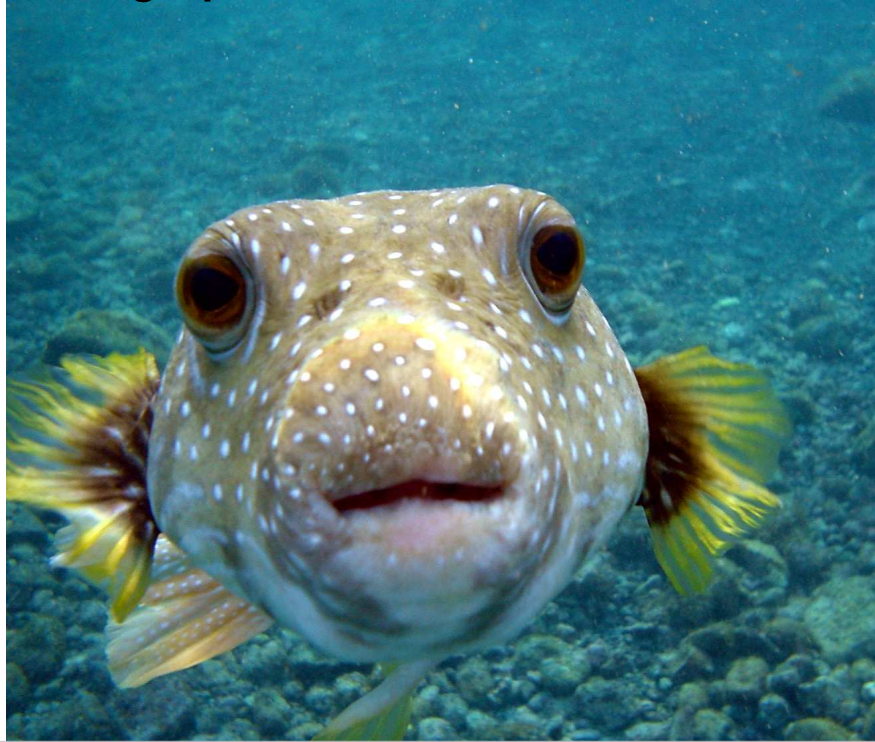
Additionally, In some other countries (e.g., France, Russia, and Pakistan) it may be illegal to use encryption at all.

From U.S. export law's point of view, the seven “terrorist countries” are Cuba, Iran, Iraq, Libya, North Korea, Sudan and Syria.

As an example of the dizzying confusion of U.S. law, a 1997 ruling permitted financial-specific cryptographic applications to be exported only if they could, by design, **only be used to encrypt financial data**. For much more information on U.S. cryptography export law, see:

<http://www.cryptolaw.org/cls2.htm#us>

Part 3: Using OpenSSH



Now let's look at some concrete examples.

Using ssh to Log In:

By default, ssh tries to log into the remote computer using your local username:

```
~/demo> whoami
elvis

~/demo> ssh server.example.com
elvis@server.example.com's password: *****
Welcome to server.example.com!
~>
```

You can tell ssh to use a different user name like this:

```
~/demo> ssh bryan@server.example.com
```

or, equivalently, this:

```
~/demo> ssh server.example.com -l bryan
```


Executing Commands Remotely:

Instead of starting a command-line session you can run a command on the remote computer by appending the command to the end of the ssh command:

```
~/demo> ssh server.example.com df  
elvis@server.example.com's password: *****
```

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
/dev/sda3	149280940	132674576	8900972	94%	/
/dev/sda1	101086	45524	50343	48%	/boot
tmpfs	969692	0	969692	0%	/dev/shm

You need to be careful to quote or escape expressions that would otherwise be interpreted locally:

```
~/demo> ssh server.example.com 'ls *.txt'
```

The command above would execute “ls *.txt” on the remote computer.

Host Keys:

When you connect to an ssh server, it first sends over its public key. If this doesn't match the key the server sent you last time, this server may not really be the one you believe it to be.

```
~/demo> ssh server.example.com
```

The first time you ssh to this server:

```
The authenticity of host 'server.example.com (192.168.5.9)' can't
be established.
RSA key fingerprint is
94:f0:1b:b7:d2:6b:3a:9c:e4:06:f2:e3:98:c4:62:6a.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'server.example.com,192.168.5.9' (RSA)
to the list of known hosts.
```

```
~/demo> ssh server.example.com
```

Connecting again, later:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@  WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!  @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle
attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
14:36:11:81:ef:a1:69:9a:25:64:80:bf:36:00:65:d7.
Please contact your system administrator.
Add correct host key in /home/elvis/.ssh/known_hosts to get rid of this
message. Offending key in /home/elvis/.ssh/known_hosts:424
RSA host key for server.example.com has changed and you have requested
strict checking.
Host key verification failed.
```

Different host keys

Keys are here, and in /etc/ssh/known_hosts

When you first connect to a given ssh server, that server sends you its public “host key”. Depending on your ssh configuration, the key will either be accepted automatically, you'll be asked if you want to accept it, or ssh will refuse to connect until you manually add to your known_hosts file. A “fingerprint” of the key will be displayed for you to look at. The fingerprint is just a short, readable checksum that is highly unlikely to be the same for two different host keys.

Once accepted, a key is stored in the user's known_hosts file. On subsequent connections, ssh will check to make sure the remote host presents the same key. If the key differs, depending on configuration, ssh will either warn you and replace the key in your known_hosts, or just refuse to connect until you manually edit your known_hosts file.

The local computer can also have a global known_hosts file: /etc/ssh/known_hosts. This might contain keys for computers in your department that users are likely to connect to. This saves the user the trouble of adding keys for these hosts when first connecting to them.

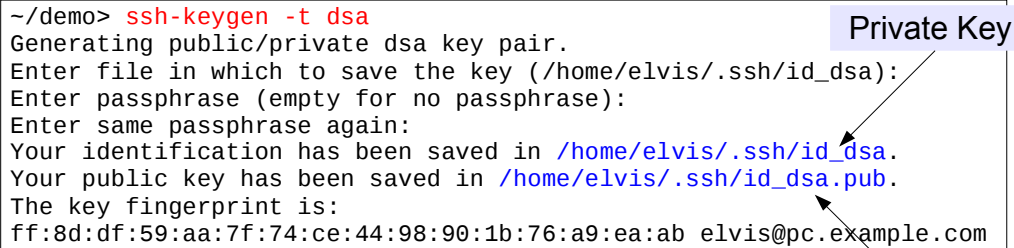
On the remote host, the ssh server's host keys are stored in /etc/ssh. The public keys are named ssh_host_key.pub, ssh_host_dsa_key.pub and ssh_host_rsa_key.pub. Which key gets used depends on which version of the ssh protocol the client chooses. If you tell your client to connect using a different version of the protocol, you may see a different host key.

Identity Keys:

Ssh can verify your identity in several ways. Usually, it does this by asking you for a username and password. Alternatively, you can generate a **public/private key pair**, and use these for authentication.

Keys can be generated with the “ssh-keygen” command:

```
~/demo> ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/elvis/.ssh/id_dsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/elvis/.ssh/id_dsa.
Your public key has been saved in /home/elvis/.ssh/id_dsa.pub.
The key fingerprint is:
ff:8d:df:59:aa:7f:74:ce:44:98:90:1b:76:a9:ea:ab elvis@pc.example.com
```



It's very important that your private key be protected. It shouldn't be readable or writeable by anyone but you.

Ssh version 2 understands keys in two formats, called “rsa” and “dsa”. Ssh version 1 understands a format called “rsa1”. You can choose the format with the “-t” switch. (You may want to generate one of each.)

You can optionally protect your key with a passphrase. If you do this, you'll have to supply the passphrase whenever ssh needs to use the key.

Ssh keys serve the same purpose as digital certificates.

Ssh may refuse to use a key if it (or its parent directory) has the wrong permissions.

Configuring Passwordless ssh Logins:

If your ssh identity key isn't protected by a passphrase, you can use it to enable passwordless ssh logins to a remote computer. Here's how:

After generating your keys, type:

```
ssh-copy-id me@remote.host.edu
```

Alternatively, you can do it by hand like this:

On the **remote** computer, edit the file “~/.ssh/authorized_keys”. To the end of this file, append the contents of your **public key** file.

(Under older operating systems, you may need to use the file “authorized_keys2” instead of “authorized_keys”. If one doesn't work, try the other.)

The ssh server may refuse to use a key if permissions aren't set properly on the authorized_keys2 file or its parent directory. These should only be readable (or writeable) by the user.

To debug problems with passwordless logins, try the following:

```
ssh -v -o  
PreferredAuthentications=publickey  
remotehost.example.com
```

This will cause ssh to report, in detail, what's happening (-v for “verbose”) and cause it to only try to use public key authentication (otherwise, it might fail over to a different type of authentication).

OpenSSH Configuration Files:

The OpenSSH ssh client is configured primarily through two files:

- **/etc/ssh/ssh_config**
System-wide configuration file for all users on the local computer.
- **~/.ssh/config**
A user's personal configuration file. This can supplement or override the options in the system-wide configuration file.

Some useful configuration options are:

StrictHostKeyChecking=ask

This option controls how ssh behaves when presented with a new host key, or a host key that differs from one already stored. Options are:

- **"ask"** (the default), which causes ssh to ask the user for confirmation before accepting a new key, and refuse to connect if a key changes.
- **"no"**, which causes ssh to blindly accept any key, even if it changes.
- **"yes"**, which requires you to manually add each key.

ForwardX11=yes

This option determines whether ssh automatically creates a tunneled X connection to the remote computer. The default is **"no"**, but many system administrators (and some Linux distributions) set it to **"yes"**.

StrictHostKeyChecking=yes is very unfriendly. With this setting, even if you're connecting to a new ssh server, ssh will refuse to connect until you manually edit your known_hosts file and add the host key for the new computer.

There are many other configuration options. See "man ssh_config" for all of them.

Note that, in your personal configuration file, it's sometimes useful to define options that apply only to a particular computer. You can do this by adding a line like "Host myhost.example.com" to your config file. Any subsequent options will apply only to this host, up to the next "Host" line, if any.

Remaining Uses for Telnet:

- Debugging:

```
~/demo> telnet www.mydomain.org 80
Trying 192.168.3.7...
Connected to www.mydomain.org.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 200 OK
Date: Tue, 24 Mar 2009 13:20:31 GMT
Server: Apache/2.2.3 (CentOS)
Last-Modified: Mon, 14 Oct 2002 22:16:57 GMT
Accept-Ranges: bytes
X-Powered-By: ModLayout/5.0
Connection: close
Content-Type: text/html; charset=UTF-8

<head>
<title>My Home Page</title>
</head>
...etc.
```

Telnet accepts an optional port number (default is 23)

In this example, we telnet to port 80, the standard port on which web servers listen, and manually send a "GET" command to the web server.

This is what happens behind the scenes when you point your web browser at a web server. the GET command is a part of "HTTP", the "HyperText Transport Protocol".

Telnet essentially gives you a (almost) raw TCP connection to a port on a remote computer. By default, telnet connects to port 23 (where the telnet server listens), but you can tell it to connect to any other port. This makes telnet useful for manually sending data to servers on remote computers.

Remaining Uses for Telnet (cont'd):

• Pranks: `~/demo> telnet my.mailserver.org 25`
Trying 192.168.1.5...
Connected to my.mailserver.org.
Escape character is '^]'.
220 desktop.mydomain.org ESMTP Postfix
HELO desktop.mydomain.org
250 desktop.mydomain.org
MAIL From: god@heaven.org
250 2.1.0 Ok
RCPT To: bryan@virginia.edu
250 2.1.5 Ok
DATA
354 End data with <CR><LF>.<CR><LF>
Subject: Repentence

Dear Bryan,

Repent!

Sincerely,
The Management

.
250 2.0.0 Ok: queued as 83EBE1540934
^]
telnet> quit
Connection closed.

In this example we use telnet to connect to port 25, the mail server port, and compose an e-mail message.

This is how mail programs talk to mail servers (and how mail servers talk to each other). These commands are part of "SMTP", the "Symmetric Mail Transport Protocol".

This is actually useful sometimes for diagnosing problems with mail servers.

Remaining Uses for Telnet (cont'd):

- Bulletin Board Systems (!):

telnet mono.org

```
bkw1a@glamdring: /home/bryan
File Edit View Search Terminal Help
|V| Monochrome (1.101y 01-Jun-11 wtf 11pm pub) (Last on Fri Apr 5 20:43) |V|
-----
DAPPRMAN'S ANIME AND MANGA SECTION
<EA> from the main menu.

      ^
     / \
    vvvvvv /|_ /|
    | /0,0 |
    | /_ _ \ |
    J /- - - \ |
    | - - - - |W| | /--\ | | /oo |
----- dapprman ~-

Menu [I] = Help and Information on Monochrome
Menu [N] = News and Media
Menu [T] = Science, Technology and Medicine
Menu [E] = Entertainment
Menu [C] = Society and Culture
Menu [R] = Recreation
Menu [M] = Monochrome Users
Hello 'Guest User'. (guest1:1)
<< 1 other user at Sat Apr 6 15:10 GMT (You have new messages) >>
```

Remaining Uses for Telnet (cont'd):

- Configuring network appliances:
(Printers, routers, etc.)

```
~/demo> telnet 192.168.1.5
Trying 192.168.101.11...
Connected to physics_315
(192.168.101.11).
Escape character is '^]'.
HP JetDirect
Password is not set

Please type "menu" for the MENU system,
or "?" for help, or "/" for current
settings.
>
```

The example above shows a telnet connection to a networked printer with an HP JetDirect network card installed.

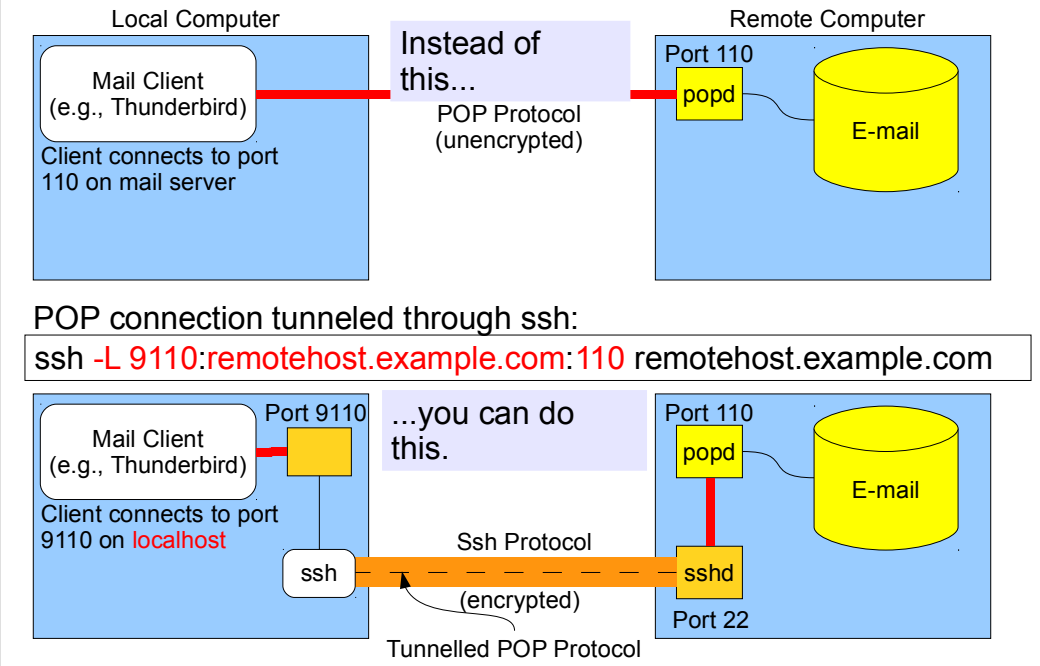
Part 4: Tunnels



Many older network protocols are still unencrypted. Ssh provides us with a way to wrap these legacy protocols in a layer of encryption, by creating an ssh tunnel between two computers.

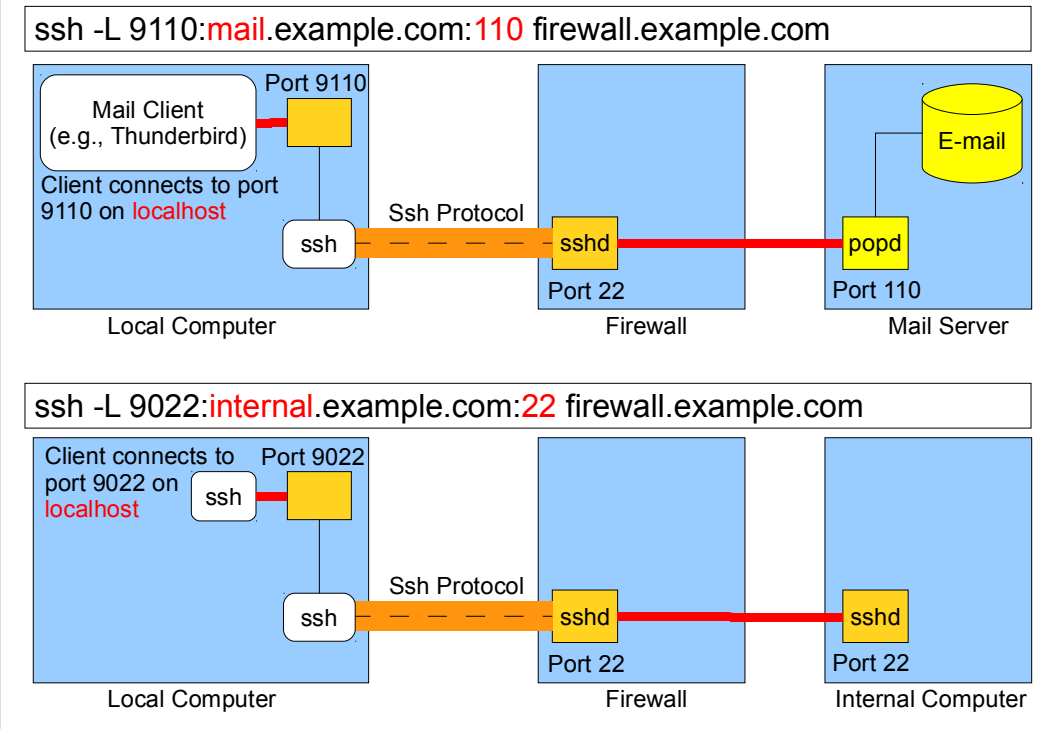
Creating Tunnels with ssh:

You can use ssh to wrap encryption around legacy protocols that aren't natively encrypted. For example:



We've already talked about the special provisions in ssh that make it easy to create an encrypted tunnel for X connections. Here we see that this functionality isn't limited to X. It can be extended to any protocol.

Tunneling through Firewalls:



The choice of local port number is arbitrary, as long as the local port isn't already in use by something else. Remember that ports below 1024 are “privileged” ports, so only the root user can use them.

In the second case, after you've set up the tunnel you'd get to the internal remote computer by typing:

```
ssh -p 9022 localhost
```

This may cause your ssh client to complain that the host key has changed (since it may already know a different key for “localhost”). In this case, you can add a section at the bottom of your `~/.ssh/config` file like this:

```
Host sneakyssh
  Hostname localhost
  HostKeyAlias sneakyssh
  Port 9022
  CheckHostIP no
```

You can then just type “ssh sneakyssh” to connect to the internal remote computer, after you've set up the tunnel.

Part 5: The ssh Server



sshd Configuration Files:

The ssh server (“sshd”) is primarily configured through the file `/etc/ssh/sshd_config`. Here's a typical `sshd_config`:

```
# Comment lines look like this.
Protocol                2
SyslogFacility          AUTHPRIV
PasswordAuthentication  yes
ChallengeResponseAuthentication no
GSSAPIAuthentication    yes
GSSAPICleanupCredentials yes
UsePAM                  yes
X11Forwarding           yes
Subsystem sftp /usr/libexec/openssh/sftp-server
```

The “**X11Forwarding**” option controls whether the server will allow ssh clients to establish a forwarded, tunneled X connection when they connect. The default is “no”, but Linux distributions often set this value to “yes”.

See “`man sshd_config`” for many more options.

Part 6: Transporting Files Securely



The ssh protocol also lets us transfer files securely over the network.

The “scp” Command:

You can also use the ssh protocol to securely copy files across the network. One way to do this is through the “scp” command, which is analogous to the “cp” command for copying files locally.

Copying files locally:

```
cp file.dat otherfile.dat  
cp file.dat /other/directory/
```

Copying files to a remote computer:

```
scp file.dat pc.example.com:otherfile.dat  
scp file.dat pc.example.com:/other/directory/
```

In the first example, the file ends up in the user's home directory on the remote computer.

Copying files from a remote computer to the local computer:

```
scp pc.example.com:file.dat otherfile.dat
```

If the username is different on the remote computer, do this:

```
scp file.dat elvis@pc.example.com:otherfile.dat
```

Whenever you specify a directory into which you want to move or copy a file, it's good practice to append the optional slash at the end of the directory name. This is true with mv, cp, scp or anything else. If you don't append the slash, and mis-type the directory name, the target will be interpreted as a file name, not a directory. This may cause you file to end up someplace unexpected, perhaps even overwriting an existing file. If you mis-type the name, but append a slash, you'll get an error message saying that no such directory exists.

Copying Multiple Files with scp:

The following command would copy all files ending in “.txt” to the user's home directory on the remote computer:

```
scp *.txt pc.example.com:
```

Copy *.txt from a remote computer to the current directory locally:

```
scp pc.example.com:'*.txt' .
```

Notice that here we need to put quotes around '*.txt' so that the local shell won't expand it before its passed to scp. (We want to send the string '*.txt' literally to the remote machine, where the shell there will expand it.)

Recursively copy the directory tree “stuff” to the remote computer:

```
scp -r stuff/ pc.example.com:stuff/
```

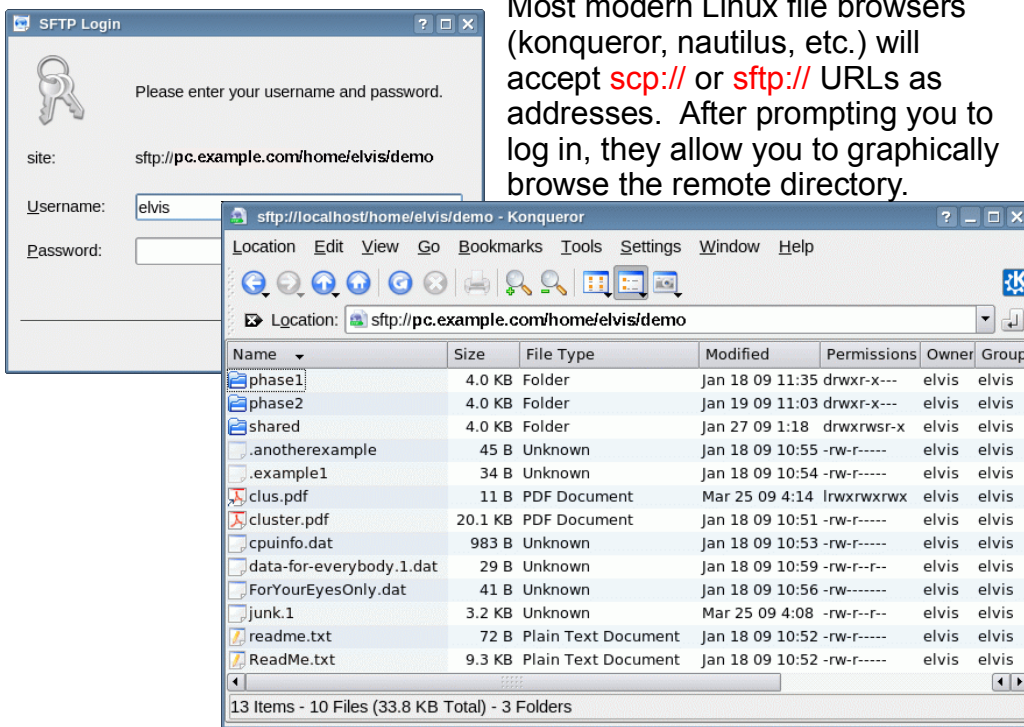
The “sftp” Command:

Alternatively, you can use the “sftp” command. This also copies files securely across the network using the ssh protocol, but it gives you an ftp-like user interface:

```
~/demo> sftp pc.example.com
Connecting to pc.example.com...
elvis@pc.example.com's password:
sftp> cd demo
sftp> dir
Fortran.dat  ReadMe.txt  clus.pdf
cluster.pdf  cpuinfo.dat  data-1.dat
phase1      phase2      readme.txt
shared
sftp> get readme.txt
Fetching /home/elvis/demo/readme.txt to readme.txt
/home/elvis/demo/readme.txt 100% 72 0.1KB/s 00:00
sftp> put junk.1
Uploading junk.1 to /home/elvis/demo/junk.1
junk.1 100% 3301 3.2KB/s 00:00
sftp> quit
```

Graphical scp/sftp Clients:

Most modern Linux file browsers (konqueror, nautilus, etc.) will accept **scp://** or **sftp://** URLs as addresses. After prompting you to log in, they allow you to graphically browse the remote directory.



The image shows two overlapping windows. The top window is titled 'SFTP Login' and contains a key icon, the text 'Please enter your username and password.', and a 'site:' label with the value 'sftp://pc.example.com/home/elvis/demo'. Below this are fields for 'Username:' (containing 'elvis') and 'Password:'. The bottom window is titled 'sftp://localhost/home/elvis/demo - Konqueror' and shows a file browser interface. The location bar contains 'sftp://pc.example.com/home/elvis/demo'. Below the location bar is a table listing files and folders.

Name	Size	File Type	Modified	Permissions	Owner	Group
phase1	4.0 KB	Folder	Jan 18 09 11:35	drwxr-x---	elvis	elvis
phase2	4.0 KB	Folder	Jan 19 09 11:03	drwxr-x---	elvis	elvis
shared	4.0 KB	Folder	Jan 27 09 1:18	drwxrwsr-x	elvis	elvis
.anotherexample	45 B	Unknown	Jan 18 09 10:55	-rw-r-----	elvis	elvis
.example1	34 B	Unknown	Jan 18 09 10:54	-rw-r-----	elvis	elvis
clus.pdf	11 B	PDF Document	Mar 25 09 4:14	lrwxrwxrwx	elvis	elvis
cluster.pdf	20.1 KB	PDF Document	Jan 18 09 10:51	-rw-r-----	elvis	elvis
cpuinfo.dat	983 B	Unknown	Jan 18 09 10:53	-rw-r-----	elvis	elvis
data-for-everybody.1.dat	29 B	Unknown	Jan 18 09 10:59	-rw-r--r--	elvis	elvis
ForYourEyesOnly.dat	41 B	Unknown	Jan 18 09 10:56	-rw-----	elvis	elvis
junk.1	3.2 KB	Unknown	Mar 25 09 4:08	-rw-r--r--	elvis	elvis
readme.txt	72 B	Plain Text Document	Jan 18 09 10:52	-rw-r-----	elvis	elvis
ReadMe.txt	9.3 KB	Plain Text Document	Jan 18 09 10:52	-rw-r-----	elvis	elvis

13 Items - 10 Files (33.8 KB Total) - 3 Folders

In the Windows world, WinSCP is a free tool that can add similar functionality to Windows Explorer.

Part 7: Using rsync



Now let's take a look at another tool. Rsync can be used to “synchronize” two directories. When copying files from one computer to another, rsync (by default) uses the ssh protocol. Rsync is an incredibly useful and versatile program that you should become familiar with.

The “rsync” Command:

One of the most useful commands that works in conjunction with ssh (and one of the most useful commands of any kind) is “rsync”.

Rsync can be used to synchronize a local directory with a remote directory, or vice versa. By default, each time you run rsync it will only copy files that are new, or have been changed.

Synchronize a remote directory with the local directory “demo”:

```
rsync -av demo/ pc.example.com:demo/
```

The “-a” switch tells rsync to copy the entire tree, and to preserve the properties of the files. The “-v” switch tells rsync to be verbose as it tells you what it's doing. This will, among other things, cause it to show the name of each file that's copied.

Synchronize a local directory with a remote directory:

```
rsync -av pc.example.com:demo/ demo/
```

If the remote user name is different, do this:

```
rsync -av demo/ elvis@pc.example.com:demo/
```

See “man rsync” for many more options.

Note that rsync is directional: it copies things from the source directory to the target directory. If “file.dat” is different in the two places, the source directory's version will overwrite the target directory's version, even if the version in the target directory is newer.

In the example above, any extra files in the target directory would be left alone. If you want rsync to make the target an exact copy of the source, with no extra files, add the “--delete” option. Before you do this, it's a good idea to run the command with the addition of the “-n” option, which will tell rsync to show you what it **would** do, without actually doing it.

Local-to-Local Copying with rsync:

You can also use rsync to copy files to a different location on the local computer. Rsync does a very good job of preserving ownerships, permissions and other file properties, so it's the best tool for moving files from place to place locally.

```
rsync -av /here/files/ /there/files/
```

In general, rsync takes the names of a “source” and a “target”. Each of these names has the form:

```
username@host:path
```

If the source or target is on the local computer, you can omit the `username@host:` part

Using Trailing Slashes:

Notice the difference between these two commands:

```
rsync -av /here/mydir/ /here/otherdir/
```

```
rsync -av /here/mydir /here/otherdir
```

In the first case, the **contents** of “mydir” would be copied into the directory “otherdir”. This is usually what you want to do.

In the second case rsync interprets this as a command to copy the **directory** “mydir” into the directory “otherdir”. Afterward, we'd find a new directory called “/here/otherdir/mydir”. This might not be what you want.

In general, if the source includes a trailing slash, rsync copies the contents. If not, it copies the directory.

Deleting Unwanted files in the Target:

If the target directory already includes some files, but you want to make it into an exact copy of the source directory, you can use the “--delete” qualifier:

```
rsync -av -delete /here/mydir/ /here/otherdir/
```

This will cause rsync to delete any files in “otherdir” that aren’t present in “mydir”. You obviously want to use this with care!

One way to be cautious is by using the “-n” qualifier:

```
rsync -n -av -delete /here/mydir/ /here/otherdir/
```

This will cause rsync to show you what it **would do**, without actually doing anything. Once you’ve done a successful trial run, you can repeat the command without the “-n” switch.

Finally, you might want to add the “--delete-delay” switch. This causes any deletions to happen **after** files have been copied, and may prevent problems in cases where rsync is interrupted unexpectedly.

Keeping Backup Copies of Deletia:

If you want to be even more cautious, you can retain copies of any files that have been deleted or modified in the target directory. To do this, use the qualifiers “-b” and “--backup-dir”:

```
rsync -av -delete -b --backup-dir=/deletia/ \  
/here/ /there/
```

In this example, any files that are deleted or modified in “/there” will be preserved in the backup directory “/deletia”.

You might even think of keeping multiple deletia directories, organized by the times at which rsync was run:

```
NOW=`date +%y%m%d%H%M%S`
```

This gives a string like
“120409113939”,
based on the current date and time.

```
rsync -av -delete \  
-b --backup-dir=/deletia/$NOW/ \  
/here/ /there/
```



Thanks!