Understanding and Using SSH

A guide to how the secure shell (ssh) works, why it's important, and how to effectively use it.

This was written by Moshe Jacobson and given on Tue Sep 18 2001 to the Linux Users Group at Georgia Tech.

Table of Contents

1 How SSH Works

1.1 How SSH negotiates an encrypted session

The server has a 1024bit public/private host key pair that remains constant, as well as a 768bit public/private server key pair that changes hourly.

When a client connects, the server sends its public host and server keys to it. To determine if the host is legitimate, the client keeps a cache of the host keys of all the servers it has ever connected to. If the public key sent by the server is indeed the same as the one in the cache, all is good. If not, the client is warned: someone may be trying to intercept the connection.

The client then generates a random 256-bit key, which it encrypts with both of the public keys sent from the server, and sends back this encrypted key. The server unencrypts the 256-bit key with its own private host key. Now both sides have a common key unknown to anyone else, which will be used to encrypt the rest of the traffic in the ssh session, with a cipher such as Blowfish or 3DES.

1.2 Password authentication

Once the encrypted session has begun as described above, the password can be sent from the client to the server with no worry that it will be sniffed; it is never sent in plaintext since the session is already encrypted.

The advantage of password authentication is that it is very secure, prone only to keylogging. Because of this weakness, however, you should try not to type any passwords on systems where root may be running a keylogger. This, of course, is a precaution that applies much more broadly than to just ssh.

1.3 RSA key authentication

A more secure method of authentication is through the use of RSA keys. The basic principle is as follows. Each user generates a 1024bit public/private key pair for himself. Generally, each user has a different key for every system he is on, but frequently, if he has multiple trusted hosts, it can be more convenient to use the same key on all the systems to simplify things.

Any host to which the user wants to connect must be aware of his public RSA key, as the server uses it during the authentication process. The user must place his public key living on the originating client machine, into his own authorized_keys file on the server.

When he wants to connect to that server, ssh will first negotiate an encrypted session, then send the server the client's public key. The server checks that the public key is in the user's authorized_keys. If so, the server sends the client a challenge (a random number encrypted with the user's public key). If the client can then send back the random number unencrypted, it has just proven that it has the private key (there is no other way to unencrypt the challenge number), and is therefore authentic.

The user's private key is a very sensitive piece of data - with it, anyone can connect to any host on which the corresponding public key is in the authorized_keys. Therefore, the user's private key is never written to disk unencrypted.

The private key is encrypted using a passphrase made up by the user. The passphrase should generally be around 20 characters in length to prevent brute force cracking attempts against it. With a safe passphrase, there is little risk in exposing the private key.

The advantage of RSA based authentication is that a password is never sent across the line, even in encrypted form. Additionally, you have to remember only one passphrase rather than a password for each system you log into. However, because of the neccessity for a long passphrase on a private key, it is becomes very inconvenient to type it every time you connect to another host.

1.4 Authentication agents

To ease the use of RSA keys, one may use an authentication agent such as ssh-agent. ssh-agent is simply a daemon that caches a user's unencrypted private key(s), to save the user from having to type his passphrase every time he wants to connect to another host.

Every key that the user wishes to be cached must be added to the agent's cache by running ssh-add and typing in the passphrase for the key being added. Multiple keys may be added to a single agent. The agent will automatically determine which key (if any) can be used to authenticate you on a host.

When run, ssh-agent creates a named socket file under /tmp, whose permissions are set to allow only you to read from and write to it. It places the name of this file into an exported environment variable, $SSH_AUTH_SOCK. ssh client processes that have this agent running as an ancestor process will have this variable available to them to inform them how to communicate with the agent. Communication with the agent is done by reading from and writing to the named socket in $SSH_AUTH_SOCK.

Once the ssh client has contacted the running ssh-agent, it attempts to authenticate with the server in the same method as mentioned before. However, the user does not have to type the passphrase for the private key, as it is already available from the agent in its unencrypted form.

Because the user's private keys are stored unencrypted in the memory of the client machine when using ssh-agent, care should be taken that keys are not added to ssh-agent on systems where root is untrusted, as root has access to shared memory, and can steal the user's unencrypted key and access any hosts to which the added keys grant access.

A drawback of using RSA authentication is that once a key is used to log into a remote host, they key is no longer available for further authentication, as it is not available anywhere in the host machine's filesystem or memory. To circumvent this problem, an arguably unsafe trick called agent forwarding is used.

1.5 Agent forwarding

Ssh has the ability to "forward" the ssh-agent it is using, in order to make it available to the user after he has connected to the remote machine. Say You have a key on system A that authenticates you on hosts B and C. The key is not available to you on host B though. With forwarding, you can ssh into host B and still have a pseudo-agent available to you on B, which forwards requests back to the real agent running on A. When you connect to B, the $SSH_AUTH_SOCK is set to point to this pseudo-agent, and is exported in your environment on B. Now you can authenticate yourself on C from B without typing a single password or passphrase.

Because agent forwarding provides access to the agent running on system A, it is now possible for root on system B to connect to the agent on A by way of the pseudo-agent created for forwarding, and authenticate himself with the user's keys. He can then log into to any system allowing one of the user's added keys.

Therefore, agent forwarding should never be used when connecting to a system on which root is untrusted. If the user must connect to multiple untrusted systems, he should disconnect from one before connecting to another - forwarded agents can be hijacked and passwords can be keylogged.

1.6 A Safe Solution

* Do not forward your agent to untrusted hosts. If you must go through an untrusted host B to get to host C, you must then trust root on B with access to C, period.
* If you're not sure what to do, use password authentication. It requires typing a password, but is safer: If someone sniffs your password, they only have access to the one system to which you're connecting, rather than all the systems on which your key is authorized. (You are using different passwords on each system, right?)

2. Using Secure Shell

2.1. Using the client

To connect to a remote system using ssh, invoke ssh with the following form: `ssh [user@]host [command]`

If your username on the remote system is the same as on the local system, you may omit the user@, and your local username will be used.

If you would like to execute a command on the remote system and immediately disconnect (rather than executing a shell and entering an interactive session), you can specify the command to run after the hostname specification. Remember that the program may not be in the search path on the remote system, so it is always a good idea to give the full path to the program if you're not sure.

So for example, if I wanted to ssh to acme.gatech.edu and list the contents of /bin, it would look something like this:

```
[ jehsom@jolt ~]$ /usr/bin/ssh gte741e@acme.gatech.edu ls ~/bin
gte741e@acme.gatech.edu's password:
myindent
vncviewer.exe
[ jehsom@jolt ~]$
```

Notice that I escaped the ~ in ~/bin. Otherwise the local shell would have expanded it to /home/jehsom, which doesn't exist on acme. The way I prefer to pass a command to ssh is to pass the whole thing in quotes:

```
/usr/bin/ssh gte741e@acme.gatech.edu "ls -l ~/bin"
```

Special shell characters such as ~ (as well as many others) will be passed straight through to the remote system and handled by the remote shell. This is what you want.

2.2. X11 Forwarding

One of the best features of ssh is its automatic X forwarding. If you have properly set your DISPLAY environment variable on the local machine (this will be the case if you're running in a terminal window under X), then ssh will automatically negotiate the forwarding of the display to the remote server. This means that when you connect to the remote machine, you may start graphical X programs, and have them display their interfaces on the local X server. All X11 traffic is encrypted, of course, which is an added benefit to the usual method of using xhost and exporting your DISPLAY.

2.3. SSH Client Configuration Options

Your ~/.ssh/config file contains all the client functionality options. You can restrict settings to specific hosts by preceeding them with a "Host hostname" line. Put all your global options first, because every line following a Host declaration will apply only to that host.

Some interesting options (again, ripped from man ssh(1)) are:

* Cipher - Specifies the cipher to use for encrypting the session in protocol version 1. Currently, "blowfish" and "3des" are supported. The default is "3des".
* EscapeChar - Sets the escape character (default: '~'). The escape character can also be set on the command line. The argument should be a single character, '^' followed by a letter, or "none" to disable the escape character entirely (making the connection transparent for binary data).
* ForwardAgent - Specifies whether the connection to the authentication agent (if any) will be forwarded to the remote machine. The argument must be "yes" or "no". The default is "no".

* ForwardX11 - Specifies whether X11 connections will be automatically redirected over the secure channel and DISPLAY set. The argument must be "yes" or "no". The default is "no".

* Protocol - Specifies the protocol versions ssh should support in order of preference. The possible values are "1" and "2". Multiple versions must be comma-separated. The default is "1,2". This means that ssh tries version 1 and falls back to version 2 if version 1 is not available.

* User - Specifies the user to log in as. This can be useful if you have a different user name on different machines. This saves the trouble of having to remember to give the user name on the command line. Usually this is used on a host-specific basis.

2.4. Using RSA Authentication

I'll try to break this up into several steps.

* Generate your own personal RSA key pair on the client machine using the 'ssh-keygen' command with no parameters. It will prompt you for a passphrase which you will type whenever you use this key. The new keyfiles are placed ~/.ssh.

* For security reasons, you MUST chmod -R go-rwx ~/.ssh, or you will not be able to connect to this host using RSA keys. This is because ssh sees a potential security breach and disallows access using the keys.

* ssh into your account on the server machine (you will obviously still be using password authentication at this point). Make the ~/.ssh directory and chmod it as above.

* cat your ~/.ssh/identity.pub on the client machine. You will see a very long line that should wrap when it gets to the edge of the screen. Copy and paste this line into the ~/.ssh/authorized_keys file on the remote system. This is one of perhaps many keys that you may put in the remote authorized_keys file. Make sure you chmod ug-rwx authorized_keys.

* Now when you ssh from the client to the server, you will be prompted for the passphrase for your new RSA key. Type it in and you will be automatically logged into the remote system. If you mistype, you will be prompted for the regular password.

2.5. Using ssh-agent to simplify things

Since it is a bit tedious to type your passphrase every time you wish to use your key, you can use ssh-agent to hold your keys for you, and provide them to ssh whenever you need to log into a remote system.

If you normally log into your system on a text console and then start X, you'll want to put the following in your login script (~/.bash_profile if you are using bash):

```
[ -z "$SSH_AUTH_SOCK" ] && eval `ssh-agent`
ssh-add
```

This way, if you have no agent running when you log in, ssh-agent will be run and you will be prompted for your key's passphrase. Now any subprocesses will inherit the connection to the agent. You can start X, and any shell sessions you start will have the agent available to them.

If you normally log into your system via a graphical console, place the following lines at the top of your ~/.xinitrc:

```
eval `ssh-agent`
ssh-askpass> &-
# Now execute your window manager with one of these commands:
#gnome-panel
#startkde
#wmaker
```

Now when you log in, you will be graphically prompted to enter your ssh passphrase before your window manager starts. All shell sessions you create will now have the $SSH_AUTH_SOCK variable set, and ssh will use the agent to authorize you.

2.6. SCP - Secure Copy

A very handy extension to ssh is scp. It allows you to copy files between systems quickly and securely, while keeping a simple interface.

You will most commonly use scp in one of the following two ways:

```
scp [-r] [user@]host:[path] path
scp [-r] path [user@]host:[path]
```

The first copies a remote file or directory to the local machine at the given path location. The second copies from local to remote.

If the user@ is not given, the local username will be assumed on the remote side. You must have a colon after the hostname, though. If it is omitted, you will create a file called user@host in your current directory. Whoops!

The remote path specification is relative to your home directory. You can begin the path with a '/' to make it absolute, though. If it is omitted, the files are copied to or from your home directory.

If the first path spec is a directory, you'll probably want to use the -r option, which will send the entire directory tree, recursively. If it is a file, the -r is unneccessary. Only one file will be copied.

NOTE: If the first path spec is a directory and you're using -r, DO NOT follow your path spec with a trailing '/', or else the files inside the directory will be copied straight into the destination path, instead of making a path and placing the files within. For example:

```
scp ~/bin acme:
```

will copy the entire ~/bin directory tree to acme, and place it in my home directory. I will end up with a ~/bin/ path on acme that looks exactly like the one here. However,

```
scp -r ~/bin/ acme:
```

will recursively copy all files and directories in ~/bin into my home directory on acme. So my home directory there will look like my bin directory here.

2.7. Starting up the server

On SysV-ish systems (Redhat, Mandrake, etc), you can usually start the ssh server by running the following command as root:

```
/etc/rc.d/init.d/sshd start
```

If your host keys have not been generated yet, it should automatically take care of that. If the server does not start up automatically on bootup, your distribution should have a tool to "turn it on". On Redhat and Mandrake, it is chkconfig:

```
/sbin/chkconfig --level 35 sshd on
```

On BSD-ish systems (Slackware, most unixes, *BSD, etc), there is a little more complexity involved. Let's assume you compiled and installed ssh under /usr/local (If not, just strip off the /usr/local from my examples). You will first have to generate your host keys:

```
# Generate the ssh1 key
ssh-keygen -f /usr/local/etc/ssh/ssh_host_key -N ""
# Generate the ssh2 keys
ssh-keygen -f /usr/local/etc/ssh/ssh_host_dsa_key -N "" -t dsa
ssh-keygen -f /usr/local/etc/ssh/ssh_host_rsa_key -N "" -t rsa
```

Now you may start the server:
/usr/local/sbin/sshd

You may want to add that line to your startup scripts so sshd will start when your system starts up.

2.8. Server Configuration

The server configuration file has a lot of interesting settings. Edit the file, usually /etc/ssh/sshd_config. You will probably be interested only in the following options (descriptions brutally ripped from the sshd(1) manpage):

* PermitRootLogin
  Specifies whether root can login using ssh(1). The argument must be "yes", "without-password", "forced-commands-only" or "no". The default is "yes". If this option is set to "no" root is not allowed to login.

* PasswordAuthentication
  Specifies whether password authentication is allowed. The default is "yes". Note that this option applies to both protocol versions 1 and 2.

* RSAAuthentication
  Specifies whether pure RSA authentication is allowed. The default is "yes". Note that this option applies to protocol version 1 only.

* X11Forwarding
  Specifies whether X11 forwarding is permitted. The default is "no". Note that disabling X11 forwarding does not improve security in any way, as users can always install their own forwarders.

Remember to killall -HUP sshd after making changes, so they will take effect.
From http://www.lugatgt.org/articles/using_ssh/

image:rdf newsfeed / //static.linuxhowtos.org/data/rdf.png (null)
|
image:rss newsfeed / //static.linuxhowtos.org/data/rss.png (null)
|
image:Atom newsfeed / //static.linuxhowtos.org/data/atom.png (null)
- Powered by
image:LeopardCMS / //static.linuxhowtos.org/data/leopardcms.png (null)
- Running on
image:Gentoo / //static.linuxhowtos.org/data/gentoo.png (null)
-

image:Valid XHTML1.1 / //static.linuxhowtos.org/data/xhtml.png (null)
:
image:Valid CSS / //static.linuxhowtos.org/data/css.png (null)
:
image:buttonmaker / //static.linuxhowtos.org/data/buttonmaker.png (null)
- Level Triple-A Conformance to Web Content Accessibility Guidelines 1.0 -
- Copyright and legal notices -
Time to create this page: ms
<!--
image:system status display / /status/output.jpg (null)
-->