**Introduction**

SSH is a secure protocol used as the primary means of connecting to Linux servers remotely. It provides a text-based interface by spawning a remote shell. After connecting, all commands you type in your local terminal are sent to the remote server and executed there.

In this cheat sheet-style guide, we will cover some common ways of connecting with SSH to achieve your objectives. This can be used as a quick reference when you need to know how to do connect to or configure your server in different ways.

How To Use This Guide

* Read the SSH Overview section first if you are unfamiliar with SSH in general or are just getting started.
* Use whichever subsequent sections are applicable to what you are trying to achieve. Most sections are not predicated on any other, so you can use the examples below independently.
* Use the Contents menu on the left side of this page (at wide page widths) or your browser's find function to locate the sections you need.
* Copy and paste the command-line examples given, substituting the values in red with your own values.

SSH Overview

The most common way of connecting to a remote Linux server is through SSH. SSH stands for Secure Shell and provides a safe and secure way of executing commands, making changes, and configuring services remotely. When you connect through SSH, you log in using an account that exists on the remote server.

**How SSH Works**

When you connect through SSH, you will be dropped into a shell session, which is a text-based interface where you can interact with your server. For the duration of your SSH session, any commands that you type into your local terminal are sent through an encrypted SSH tunnel and executed on your server.

The SSH connection is implemented using a client-server model. This means that for an SSH connection to be established, the remote machine must be running a piece of software called an SSH daemon. This software listens for connections on a specific network port, authenticates connection requests, and spawns the appropriate environment if the user provides the correct credentials.

The user's computer must have an SSH client. This is a piece of software that knows how to communicate using the SSH protocol and can be given information about the remote host to connect to, the username to use, and the credentials that should be passed to authenticate. The client can also specify certain details about the connection type they would like to establish.

**How SSH Authenticates Users**

Clients generally authenticate either using passwords (less secure and not recommended) or SSH keys, which are very secure.

Password logins are encrypted and are easy to understand for new users. However, automated bots and malicious users will often repeatedly try to authenticate to accounts that allow password-based logins, which can lead to security compromises. For this reason, we recommend always setting up SSH key-based authentication for most configurations.

SSH keys are a matching set of cryptographic keys which can be used for authentication. Each set contains a public and a private key. The public key can be shared freely without concern, while the private key must be vigilantly guarded and never exposed to anyone.

To authenticate using SSH keys, a user must have an SSH key pair on their local computer. On the remote server, the public key must be copied to a file within the user's home directory at ~/.ssh/authorized\_keys. This file contains a list of public keys, one-per-line, that are authorized to log into this account.

When a client connects to the host, wishing to use SSH key authentication, it will inform the server of this intent and will tell the server which public key to use. The server then check its authorized\_keys file for the public key, generate a random string and encrypts it using the public key. This encrypted message can only be decrypted with the associated private key. The server will send this encrypted message to the client to test whether they actually have the associated private key.

Upon receipt of this message, the client will decrypt it using the private key and combine the random string that is revealed with a previously negotiated session ID. It then generates an MD5 hash of this value and transmits it back to the server. The server already had the original message and the session ID, so it can compare an MD5 hash generated by those values and determine that the client must have the private key.

Now that you know how SSH works, we can begin to discuss some examples to demonstrate different ways of working with SSH

Generating and Working with SSH Keys

This section will cover how to generate SSH keys on a client machine and distribute the public key to servers where they should be used. This is a good section to start with if you have not previously generated keys due to the increased security that it allows for future connections.

**Generating an SSH Key Pair**

Generating a new SSH public and private key pair on your local computer is the first step towards authenticating with a remote server without a password. Unless there is a good reason not to, you should always authenticate using SSH keys.

A number of cryptographic algorithms can be used to generate SSH keys, including RSA, DSA, and ECDSA. RSA keys are generally preferred and are the default key type.

To generate an RSA key pair on your local computer, type:

ssh-keygen

Generating public/private rsa key pair.

Enter file in which to save the key (/home/demo/.ssh/id\_rsa):

This prompt allows you to choose the location to store your RSA private key. Press ENTER to leave this as the default, which will store them in the .ssh hidden directory in your user's home directory. Leaving the default location selected will allow your SSH client to find the keys automatically.

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

The next prompt allows you to enter a passphrase of an arbitrary length to secure your private key. By default, you will have to enter any passphrase you set here every time you use the private key, as an additional security measure. Feel free to press ENTER to leave this blank if you do not want a passphrase. Keep in mind though that this will allow anyone who gains control of your private key to login to your servers.

If you choose to enter a passphrase, nothing will be displayed as you type. This is a security precaution.

Your identification has been saved in /root/.ssh/id\_rsa.

Your public key has been saved in /root/.ssh/id\_rsa.pub.

The key fingerprint is:

8c:e9:7c:fa:bf:c4:e5:9c:c9:b8:60:1f:fe:1c:d3:8a root@here

The key's randomart image is:

+--[ RSA 2048]----+

| |

| |

| |

| + |

| o S . |

| o . \* + |

| o + = O . |

| + = = + |

| ....Eo+ |

+-----------------+

This procedure has generated an RSA SSH key pair, located in the .ssh hidden directory within your user's home directory. These files are:

* ~/.ssh/id\_rsa: The private key. DO NOT SHARE THIS FILE!
* ~/.ssh/id\_rsa.pub: The associated public key. This can be shared freely without consequence.

**Generate an SSH Key Pair with a Larger Number of Bits**

SSH keys are 2048 bits by default. This is generally considered to be good enough for security, but you can specify a greater number of bits for a more hardened key.

To do this, include the -b argument with the number of bits you would like. Most servers support keys with a length of at least 4096 bits. Longer keys may not be accepted for DDOS protection purposes:

ssh-keygen -b 4096

If you had previously created a different key, you will be asked if you wish to overwrite your previous key:

Overwrite (y/n)?

If you choose "yes", your previous key will be overwritten and you will no longer be able to log into servers using that key. Because of this, be sure to overwrite keys with caution.

**Removing or Changing the Passphrase on a Private Key**

If you have generated a passphrase for your private key and wish to change or remove it, you can do so easily.

**Note**: To change or remove the passphrase, you must know the original passphrase. If you have lost the passphrase to the key, there is no recourse and you will have to generate a new key pair.

To change or remove the passphrase, simply type:

ssh-keygen -p

Enter file in which the key is (/root/.ssh/id\_rsa):

You can type the location of the key you wish to modify or press ENTER to accept the default value:

Enter old passphrase:

Enter the old passphrase that you wish to change. You will then be prompted for a new passphrase:

Enter new passphrase (empty for no passphrase):

Enter same passphrase again:

Here, enter your new passphrase or press ENTER to remove the passphrase.

**Displaying the SSH Key Fingerprint**

Each SSH key pair share a single cryptographic "fingerprint" which can be used to uniquely identify the keys. This can be useful in a variety of situations.

To find out the fingerprint of an SSH key, type:

ssh-keygen -l

Enter file in which the key is (/root/.ssh/id\_rsa):

You can press ENTER if that is the correct location of the key, else enter the revised location. You will be given a string which contains the bit-length of the key, the fingerprint, and account and host it was created for, and the algorithm used:

4096 8e:c4:82:47:87:c2:26:4b:68:ff:96:1a:39:62:9e:4e demo@test (RSA)

**Copying your Public SSH Key to a Server with SSH-Copy-ID**

To copy your public key to a server, allowing you to authenticate without a password, a number of approaches can be taken.

If you currently have password-based SSH access configured to your server, and you have the ssh-copy-id utility installed, this is a simple process. The ssh-copy-id tool is included in many Linux distributions' OpenSSH packages, so it very likely may be installed by default.

If you have this option, you can easily transfer your public key by typing:

ssh-copy-id username@remote\_host

This will prompt you for the user account's password on the remote system:

The authenticity of host '111.111.11.111 (111.111.11.111)' can't be established.

ECDSA key fingerprint is fd:fd:d4:f9:77:fe:73:84:e1:55:00:ad:d6:6d:22:fe.

Are you sure you want to continue connecting (yes/no)? yes

/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed

/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install the new keys

demo@111.111.11.111's password:

After typing in the password, the contents of your ~/.ssh/id\_rsa.pub key will be appended to the end of the user account's ~/.ssh/authorized\_keys file:

Number of key(s) added: 1

Now try logging into the machine, with: "ssh 'demo@111.111.11.111'"

and check to make sure that only the key(s) you wanted were added.

You can now log into that account without a password:

ssh username@remote\_host

**Copying your Public SSH Key to a Server Without SSH-Copy-ID**

If you do not have the ssh-copy-id utility available, but still have password-based SSH access to the remote server, you can copy the contents of your public key in a different way.

You can output the contents of the key and pipe it into the ssh command. On the remote side, you can ensure that the ~/.ssh directory exists, and then append the piped contents into the ~/.ssh/authorized\_keys file:

cat ~/.ssh/id\_rsa.pub | ssh username@remote\_host "mkdir -p ~/.ssh && cat >> ~/.ssh/authorized\_keys"

You will be asked to supply the password for the remote account:

The authenticity of host '111.111.11.111 (111.111.11.111)' can't be established.

ECDSA key fingerprint is fd:fd:d4:f9:77:fe:73:84:e1:55:00:ad:d6:6d:22:fe.

Are you sure you want to continue connecting (yes/no)? yes

demo@111.111.11.111's password:

After entering the password, your key will be copied, allowing you to log in without a password:

ssh username@remote\_IP\_host

**Copying your Public SSH Key to a Server Manually**

If you do not have password-based SSH access available, you will have to add your public key to the remote server manually.

On your local machine, you can find the contents of your public key file by typing:

cat ~/.ssh/id\_rsa.pub

ssh-rsa  demo@test

You can copy this value, and manually paste it into the appropriate location on the remote server. You will have to log into the remote server through other means (like the DigitalOcean web console).

On the remote server, create the ~/.ssh directory if it does not already exist:

mkdir -p ~/.ssh

Afterwards, you can create or append the ~/.ssh/authorized\_keys file by typing:

echo public\_key\_string >> ~/.ssh/authorized\_keys

You should now be able to log into the remote server without a password.

Basic Connection Instructions

The following section will cover some of the basics about how to connect to a server with SSH.

**Connecting to a Remote Server**

To connect to a remote server and open a shell session there, you can use the ssh command.

The simplest form assumes that your username on your local machine is the same as that on the remote server. If this is true, you can connect using:

ssh remote\_host

If your username is different on the remoter server, you need to pass the remote user's name like this:

ssh username@remote\_host

Your first time connecting to a new host, you will see a message that looks like this:

The authenticity of host '111.111.11.111 (111.111.11.111)' can't be established.

ECDSA key fingerprint is fd:fd:d4:f9:77:fe:73:84:e1:55:00:ad:d6:6d:22:fe.

Are you sure you want to continue connecting (yes/no)? yes

Type "yes" to accept the authenticity of the remote host.

If you are using password authentication, you will be prompted for the password for the remote account here. If you are using SSH keys, you will be prompted for your private key's passphrase if one is set, otherwise you will be logged in automatically.

**Running a Single Command on a Remote Server**

To run a single command on a remote server instead of spawning a shell session, you can add the command after the connection information, like this:

ssh username@remote\_host command\_to\_run

This will connect to the remote host, authenticate with your credentials, and execute the command you specified. The connection will immediately close afterwards.

**Logging into a Server with a Different Port**

By default the SSH daemon on a server runs on port 22. Your SSH client will assume that this is the case when trying to connect. If your SSH server is listening on a non-standard port (this is demonstrated in a later section), you will have to specify the new port number when connecting with your client.

You can do this by specifying the port number with the -p option:

ssh -p port\_num username@remote\_host

To avoid having to do this every time you log into your remote server, you can create or edit a configuration file in the ~/.ssh directory within the home directory of your local computer.

Edit or create the file now by typing:

nano ~/.ssh/config

In here, you can set host-specific configuration options. To specify your new port, use a format like this:

Host remote\_alias

HostName remote\_host

Port port\_num

This will allow you to log in without specifying the specific port number on the command line.

**Adding your SSH Keys to an SSH Agent to Avoid Typing the Passphrase**

If you have an passphrase on your private SSH key, you will be prompted to enter the passphrase every time you use it to connect to a remote host.

To avoid having to repeatedly do this, you can run an SSH agent. This small utility stores your private key after you have entered the passphrase for the first time. It will be available for the duration of your terminal session, allowing you to connect in the future without re-entering the passphrase.

This is also important if you need to forward your SSH credentials (shown below).

To start the SSH Agent, type the following into your local terminal session:

eval $(ssh-agent)

Agent pid 10891

This will start the agent program and place it into the background. Now, you need to add your private key to the agent, so that it can manage your key:

ssh-add

Enter passphrase for /home/demo/.ssh/id\_rsa:

Identity added: /home/demo/.ssh/id\_rsa (/home/demo/.ssh/id\_rsa)

You will have to enter your passphrase (if one is set). Afterwards, your identity file is added to the agent, allowing you to use your key to sign in without having re-enter the passphrase again.

**Forwarding your SSH Credentials to Use on a Server**

If you wish to be able to connect without a password to one server from within another server, you will need to forward your SSH key information. This will allow you to authenticate to another server through the server you are connected to, using the credentials on your local computer.

To start, you must have your SSH agent started and your SSH key added to the agent (see above). After this is done, you need to connect to your first server using the -A option. This forwards your credentials to the server for this session:

ssh -A username@remote\_host

From here, you can SSH into any other host that your SSH key is authorized to access. You will connect as if your private SSH key were located on this server.

Server-Side Configuration Options

This section contains some common server-side configuration options that can shape the way that your server responds and what types of connections are allowed.

**Disabling Password Authentication**

If you have SSH keys configured, tested, and working properly, it is probably a good idea to disable password authentication. This will prevent any user from signing in with SSH using a password.

To do this, connect to your remote server and open the /etc/ssh/sshd\_config file with root or sudo privileges:

sudo nano /etc/ssh/sshd\_config

Inside of the file, search for the PasswordAuthentication directive. If it is commented out, uncomment it. Set it to "no" to disable password logins:

PasswordAuthentication no

After you have made the change, save and close the file. To implement the changes, you should restart the SSH service.

On Ubuntu/Debian:

sudo service ssh restart

On CentOS/Fedora:

sudo service sshd restart

Now, all accounts on the system will be unable to login with SSH using passwords.

**Changing the Port that the SSH Daemon Runs On**

Some administrators suggest that you change the default port that SSH runs on. This can help decrease the number of authentication attempts your server is subjected to from automated bots.

To change the port that the SSH daemon listens on, you will have to log into your remote server. Open the sshd\_config file on the remote system with root privileges, either by logging in with that user or by using sudo:

sudo nano /etc/ssh/sshd\_config

Once you are inside, you can change the port that SSH runs on by finding the Port 22 specification and modifying it to reflect the port you wish to use. For instance, to change the port to 4444, put this in your file:

#Port 22

Port 4444

Save and close the file when you are finished. To implement the changes, you must restart the SSH daemon.

On Ubuntu/Debian:

sudo service ssh restart

On CentOS/Fedora:

sudo service sshd restart

After the daemon restarts, you will need to authenticate by specifying the port number (demonstrated in an earlier section).

**Limiting the Users Who can Connect Through SSH**

To explicitly limit the user accounts who are able to login through SSH, you can take a few different approaches, each of which involve editing the SSH daemon config file.

On your remote server, open this file now with root or sudo privileges:

sudo nano /etc/ssh/sshd\_config

The first method of specifying the accounts that are allowed to login is using the AllowUsers directive. Search for the AllowUsers directive in the file. If one does not exist, create it anywhere. After the directive, list the user accounts that should be allowed to login through SSH:

AllowUsers user1 user2

Save and close the file. Restart the daemon to implement your changes.

On Ubuntu/Debian:

sudo service ssh restart

On CentOS/Fedora:

sudo service sshd restart

If you are more comfortable with group management, you can use the AllowGroups directive instead. If this is the case, just add a single group that should be allowed SSH access (we will create this group and add members momentarily):

AllowGroups sshmembers

Save and close the file.

Now, you can create a system group (without a home directory) matching the group you specified by typing:

sudo groupadd -r sshmembers

Make sure that you add whatever user accounts you need to this group. This can be done by typing:

sudo usermod -a -G sshmembers user1

sudo usermod -a -G sshmembers user2

Now, restart the SSH daemon to implement your changes.

On Ubuntu/Debian:

sudo service ssh restart

On CentOS/Fedora:

sudo service sshd restart

**Disabling Root Login**

It is often advisable to completely disable root login through SSH after you have set up an SSH user account that has sudo privileges.

To do this, open the SSH daemon configuration file with root or sudo on your remote server.

sudo nano /etc/ssh/sshd\_config

Inside, search for a directive called PermitRootLogin. If it is commented, uncomment it. Change the value to "no":

PermitRootLogin no

Save and close the file. To implement your changes, restart the SSH daemon.

On Ubuntu/Debian:

sudo service ssh restart

On CentOS/Fedora:

sudo service sshd restart

**Allowing Root Access for Specific Commands**

There are some cases where you might want to disable root access generally, but enable it in order to allow certain applications to run correctly. An example of this might be a backup routine.

This can be accomplished through the root user's authorized\_keys file, which contains SSH keys that are authorized to use the account.

Add the key from your local computer that you wish to use for this process (we recommend creating a new key for each automatic process) to the root user's authorized\_keys file on the server. We will demonstrate with the ssh-copy-id command here, but you can use any of the methods of copying keys we discuss in other sections:

ssh-copy-id root@remote\_host

Now, log into the remote server. We will need to adjust the entry in the authorized\_keys file, so open it with root or sudo access:

sudo nano /root/.ssh/authorized\_keys

At the beginning of the line with the key you uploaded, add a command= listing that defines the command that this key is valid for. This should include the full path to the executable, plus any arguments:

command="/path/to/command arg1 arg2" ssh-rsa ...

Save and close the file when you are finished.

Now, open the sshd\_config file with root or sudo privileges:

sudo nano /etc/ssh/sshd\_config

Find the directive PermitRootLogin, and change the value to forced-commands-only. This will only allow SSH key logins to use root when a command has been specified for the key:

PermitRootLogin forced-commands-only

Save and close the file. Restart the SSH daemon to implement your changes.

On Ubuntu/Debian:

sudo service ssh restart

On CentOS/Fedora:

sudo service sshd restart

**Forwarding X Application Displays to the Client**

The SSH daemon can be configured to automatically forward the display of X applications on the server to the client machine. For this to function correctly, the client must have an X windows system configured and enabled.

To enable this functionality, log into your remote server and edit the sshd\_config file as root or with sudo privileges:

sudo nano /etc/ssh/sshd\_config

Search for the X11Forwarding directive. If it is commented out, uncomment it. Create it if necessary and set the value to "yes":

X11Forwarding yes

Save and close the file. Restart your SSH daemon to implement these changes.

On Ubuntu/Debian:

sudo service ssh restart

On CentOS/Fedora:

sudo service sshd restart

To connect to the server and forward an application's display, you have to pass the -X option from the client upon connection:

ssh -X username@remote\_host

Graphical applications started on the server through this session should be displayed on the local computer. The performance might be a bit slow, but it is very helpful in a pinch.

Client-Side Configuration Options

In the next section, we'll focus on some adjustments that you can make on the client side of the connection.

**Defining Server-Specific Connection Information**

On your local computer, you can define individual configurations for some or all of the servers you connect to. These can be stored in the ~/.ssh/config file, which is read by your SSH client each time it is called.

Create or open this file in your text editor on your local computer:

nano ~/.ssh/config

Inside, you can define individual configuration options by introducing each with a Host keyword, followed by an alias. Beneath this and indented, you can define any of the directives found in the ssh\_config man page:

man ssh\_config

An example configuration would be:

Host testhost

HostName example.com

Port 4444

User demo

You could then connect to example.com on port 4444 using the username "demo" by simply typing:

ssh testhost

You can also use wildcards to match more than one host. Keep in mind that later matches can override earlier ones. Because of this, you should put your most general matches at the top. For instance, you could default all connections to not allow X forwarding, with an override for example.com by having this in your file:

Host \*

ForwardX11 no

Host testhost

HostName example.com

ForwardX11 yes

Port 4444

User demo

Save and close the file when you are finished.

**Keeping Connections Alive to Avoid Timeout**

If you find yourself being disconnected from SSH sessions before you are ready, it is possible that your connection is timing out.

You can configure your client to send a packet to the server every so often in order to avoid this situation:

On your local computer, you can configure this for every connection by editing your ~/.ssh/config file. Open it now:

nano ~/.ssh/config

If one does not already exist, at the top of the file, define a section that will match all hosts. Set the ServerAliveInterval to "120" to send a packet to the server every two minutes. This should be enough to notify the server not to close the connection:

Host \*

ServerAliveInterval 120

Save and close the file when you are finished.

**Disabling Host Checking**

By default, whenever you connect to a new server, you will be shown the remote SSH daemon's host key fingerprint.

The authenticity of host '111.111.11.111 (111.111.11.111)' can't be established.

ECDSA key fingerprint is fd:fd:d4:f9:77:fe:73:84:e1:55:00:ad:d6:6d:22:fe.

Are you sure you want to continue connecting (yes/no)? yes

This is configured so that you can verify the authenticity of the host you are attempting to connect to and spot instances where a malicious user may be trying to masquerade as the remote host.

In certain circumstances, you may wish to disable this feature. **Note**: This can be a big security risk, so make sure you know what you are doing if you set your system up like this.

To make the change, the open the ~/.ssh/config file on your local computer:

nano ~/.ssh/config

If one does not already exist, at the top of the file, define a section that will match all hosts. Set the StrictHostKeyChecking directive to "no" to add new hosts automatically to the known\_hosts file. Set the UserKnownHostsFile to /dev/null to not warn on new or changed hosts:

Host \*

StrictHostKeyChecking no

UserKnownHostsFile /dev/null

You can enable the checking on a case-by-case basis by reversing those options for other hosts. The default for StrictHostKeyChecking is "ask":

Host \*

StrictHostKeyChecking no

UserKnownHostsFile /dev/null

Host testhost

HostName example.com

StrictHostKeyChecking ask

UserKnownHostsFile /home/demo/.ssh/known\_hosts

**Multiplexing SSH Over a Single TCP Connection**

There are situations where establishing a new TCP connection can take longer than you would like. If you are making multiple connections to the same machine, you can take advantage of multiplexing.

SSH multiplexing re-uses the same TCP connection for multiple SSH sessions. This removes some of the work necessary to establish a new session, possibly speeding things up. Limiting the number of connections may also be helpful for other reasons.

To set up multiplexing, you can manually set up the connections, or you can configure your client to automatically use multiplexing when available. We will demonstrate the second option here.

To configure multiplexing, edit your SSH client's configuration file on your local machine:

nano ~/.ssh/config

If you do not already have a wildcard host definition at the top of the file, add one now (as Host \*). We will be setting the ControlMaster, ControlPath, and ControlPersist values to establish our multiplexing configuration.

The ControlMaster should be set to "auto" in able to automatically allow multiplexing if possible. The ControlPath will establish the path to control socket. The first session will create this socket and subsequent sessions will be able to find it because it is labeled by username, host, and port.

Setting the ControlPersist option to "1" will allow the initial master connection to be backgrounded. The "1" specifies that the TCP connection should automatically terminate one second after the last SSH session is closed:

Host \*

ControlMaster auto

ControlPath ~/.ssh/multiplex/%r@%h:%p

ControlPersist 1

Save and close the file when you are finished. Now, we need to actually create the directory we specified in the control path:

mkdir ~/.ssh/multiplex

Now, any sessions that are established with the same machine will attempt to use the existing socket and TCP connection. When the last session exists, the connection will be torn down after one second.

If for some reason you need to bypass the multiplexing configuration temporarily, you can do so by passing the -S flag with "none":

ssh -S none username@remote\_host

Setting Up SSH Tunnels

Tunneling other traffic through a secure SSH tunnel is an excellent way to work around restrictive firewall settings. It is also a great way to encrypt otherwise unencrypted network traffic.

**Configuring Local Tunneling to a Server**

SSH connections can be used to tunnel traffic from ports on the local host to ports on a remote host.

A local connection is a way of accessing a network location from your local computer through your remote host. First, an SSH connection is established to your remote host. On the remote server, a connection is made to an external (or internal) network address provided by the user and traffic to this location is tunneled to your local computer on a specified port.

This is often used to tunnel to a less restricted networking environment by bypassing a firewall. Another common use is to access a "localhost-only" web interface from a remote location.

To establish a local tunnel to your remote server, you need to use the -L parameter when connecting and you must supply three pieces of additional information:

* The local port where you wish to access the tunneled connection.
* The host that you want your remote host to connect to.
* The port that you want your remote host to connect on.

These are given, in the order above (separated by colons), as arguments to the -L flag. We will also use the -f flag, which causes SSH to go into the background before executing and the -N flag, which does not open a shell or execute a program on the remote side.

For instance, to connect to example.com on port 80 on your remote host, making the connection available on your local machine on port 8888, you could type:

ssh -f -N -L 8888:example.com:80 username@remote\_host

Now, if you point your local web browser to 127.0.0.1:8888, you should see whatever content is at example.com on port 80.

A more general guide to the syntax is:

ssh -L your\_port:site\_or\_IP\_to\_access:site\_port username@host

Since the connection is in the background, you will have to find its PID to kill it. You can do so by searching for the port you forwarded:

ps aux | grep 8888

1001 5965 0.0 0.0 48168 1136 ? Ss 12:28 0:00 ssh -f -N -L 8888:example.com:80 username@remote\_host

1001 6113 0.0 0.0 13648 952 pts/2 S+ 12:37 0:00 grep --colour=auto 8888

You can then kill the process by targeting the PID, which is the number in the second column of the line that matches your SSH command:

kill 5965

Another option is to start the connection *without* the -f flag. This will keep the connection in the foreground, preventing you from using the terminal window for the duration of the forwarding. The benefit of this is that you can easily kill the tunnel by typing "CTRL-C".

**Configuring Remote Tunneling to a Server**

SSH connections can be used to tunnel traffic from ports on the local host to ports on a remote host.

In a remote tunnel, a connection is made to a remote host. During the creation of the tunnel, a *remote* port is specified. This port, on the remote host, will then be tunneled to a host and port combination that is connected to from the local computer. This will allow the remote computer to access a host through your local computer.

This can be useful if you need to allow access to an internal network that is locked down to external connections. If the firewall allows connections *out* of the network, this will allow you to connect out to a remote machine and tunnel traffic from that machine to a location on the internal network.

To establish a remote tunnel to your remote server, you need to use the -R parameter when connecting and you must supply three pieces of additional information:

* The port where the remote host can access the tunneled connection.
* The host that you want your local computer to connect to.
* The port that you want your local computer to connect to.

These are given, in the order above (separated by colons), as arguments to the -R flag. We will also use the -f flag, which causes SSH to go into the background before executing and the -N flag, which does not open a shell or execute a program on the remote side.

For instance, to connect to example.com on port 80 on our local computer, making the connection available on our remote host on port 8888, you could type:

ssh -f -N -R 8888:example.com:80 username@remote\_host

Now, on the remote host, opening a web browser to 127.0.0.1:8888 would allow you to see whatever content is at example.com on port 80.

A more general guide to the syntax is:

ssh -R remote\_port:site\_or\_IP\_to\_access:site\_port username@host

Since the connection is in the background, you will have to find its PID to kill it. You can do so by searching for the port you forwarded:

ps aux | grep 8888

1001 5965 0.0 0.0 48168 1136 ? Ss 12:28 0:00 ssh -f -N -R 8888:example.com:80 username@remote\_host

1001 6113 0.0 0.0 13648 952 pts/2 S+ 12:37 0:00 grep --colour=auto 8888

You can then kill the process by targeting the PID, which is the number in the second column, of the line that matches your SSH command:

kill 5965

Another option is to start the connection *without* the -f flag. This will keep the connection in the foreground, preventing you from using the terminal window for the duration of the forwarding. The benefit of this is that you can easily kill the tunnel by typing "CTRL-C".

**Configuring Dynamic Tunneling to a Remote Server**

SSH connections can be used to tunnel traffic from ports on the local host to ports on a remote host.

A dynamic tunnel is similar to a local tunnel in that it allows the local computer to connect to other resources *through* a remote host. A dynamic tunnel does this by simply specifying a single local port. Applications that wish to take advantage of this port for tunneling must be able to communicate using the SOCKS protocol so that the packets can be correctly redirected at the other side of the tunnel.

Traffic that is passed to this local port will be sent to the remote host. From there, the SOCKS protocol will be interpreted to establish a connection to the desired end location. This set up allows a SOCKS-capable application to connect to any number of locations through the remote server, without multiple static tunnels.

To establish the connection, we will pass the -D flag along with the local port where we wish to access the tunnel. We will also use the -f flag, which causes SSH to go into the background before executing and the -N flag, which does not open a shell or execute a program on the remote side.

For instance, to establish a tunnel on port "7777", you can type:

ssh -f -N -D 7777 username@remote\_host

From here, you can start pointing your SOCKS-aware application (like a web browser), to the port you selected. The application will send its information into a socket associated with the port.

The method of directing traffic to the SOCKS port will differ depending on application. For instance, in Firefox, the general location is Preferences > Advanced > Settings > Manual proxy configurations. In Chrome, you can start the application with the --proxy-server= flag set. You will want to use the localhost interface and the port you forwarded.

Since the connection is in the background, you will have to find its PID to kill it. You can do so by searching for the port you forwarded:

ps aux | grep 8888

1001 5965 0.0 0.0 48168 1136 ? Ss 12:28 0:00 ssh -f -N -D 7777 username@remote\_host

1001 6113 0.0 0.0 13648 952 pts/2 S+ 12:37 0:00 grep --colour=auto 8888

You can then kill the process by targeting the PID, which is the number in the second column, of the line that matches your SSH command:

kill 5965

Another option is to start the connection *without* the -f flag. This will keep the connection in the foreground, preventing you from using the terminal window for the duration of the forwarding. The benefit of this is that you can easily kill the tunnel by typing "CTRL-C".

Using SSH Escape Codes to Control Connections

Even after establishing an SSH session, it is possible to exercise control over the connection from within the terminal. We can do this with something called SSH escape codes, which allow us to interact with our local SSH software from within a session.

**Forcing a Disconnect from the Client-Side (How to Exit Out of a Stuck or Frozen Session)**

One of the most useful feature of OpenSSH that goes largely unnoticed is the ability to control certain aspects of the session from within.

These commands can be executed starting with the ~ control character within an SSH session. Control commands will only be interpreted if they are the first thing that is typed after a newline, so always press ENTER one or two times prior to using one.

One of the most useful controls is the ability to initiate a disconnect from the client. SSH connections are typically closed by the server, but this can be a problem if the server is suffering from issues or if the connection has been broken. By using a client-side disconnect, the connection can be cleanly closed from the client.

To close a connection from the client, use the control character (~), with a dot. If your connection is having problems, you will likely be in what appears to be a stuck terminal session. Type the commands despite the lack of feedback to perform a client-side disconnect:

[ENTER]

~.

The connection should immediately close, returning you to your local shell session.

**Placing an SSH Session into the Background**

One of the most useful feature of OpenSSH that goes largely unnoticed is the ability to control certain aspects of the session from within the connection.

These commands can be executed starting with the ~ control character from within an SSH connection. Control commands will only be interpreted if they are the first thing that is typed after a newline, so always press ENTER one or two times prior to using one.

One capability that this provides is to put an SSH session into the background. To do this, we need to supply the control character (~) and then execute the conventional keyboard shortcut to background a task (CTRL-z):

[ENTER]

~[CTRL-z]

This will place the connection into the background, returning you to your local shell session. To return to your SSH session, you can use the conventional job control mechanisms.

You can immediately re-activate your most recent backgrounded task by typing:

fg

If you have multiple backgrounded tasks, you can see the available jobs by typing:

jobs

[1]+ Stopped ssh username@some\_host

[2] Stopped ssh username@another\_host

You can then bring any of the tasks to the foreground by using the index in the first column with a percentage sign:

fg %2

**Changing Port Forwarding Options on an Existing SSH Connection**

One of the most useful feature of OpenSSH that goes largely unnoticed is the ability to control certain aspects of the session from within the connection.

These commands can be executed starting with the ~ control character from within an SSH connection. Control commands will only be interpreted if they are the first thing that is typed after a newline, so always press ENTER one or two times prior to using one.

One thing that this allows is for a user to alter the port forwarding configuration after the connection has already been established. This allows you to create or tear down port forwarding rules on-the-fly.

These capabilities are part of the SSH command line interface, which can be accessed during a session by using the control character (~) and "C":

[ENTER]

~C

ssh>

You will be given an SSH command prompt, which has a very limited set of valid commands. To see the available options, you can type -h from this prompt. If nothing is returned, you may have to increase the verbosity of your SSH output by using ~v a few times:

[ENTER]

~v

~v

~v

~C

-h

Commands:

-L[bind\_address:]port:host:hostport Request local forward

-R[bind\_address:]port:host:hostport Request remote forward

-D[bind\_address:]port Request dynamic forward

-KL[bind\_address:]port Cancel local forward

-KR[bind\_address:]port Cancel remote forward

-KD[bind\_address:]port Cancel dynamic forward

As you can see, you can easily implement any of the forwarding options using the appropriate options (see the forwarding section for more information). You can also destroy a tunnel with the associated "kill" command specified with a "K" before the forwarding type letter. For instance, to kill a local forward (-L), you could use the -KL command. You will only need to provide the port for this.

So, to set up a local port forward, you may type:

[ENTER]

~C

-L 8888:127.0.0.1:80

Port 8888 on your local computer will now be able to communicate with the web server on the host you are connecting to. When you are finished, you can tear down that forward by typing:

[ENTER]

~C

-KL 8888

Conclusion

The above instructions should cover the majority of the information most users will need about SSH on a day-to-day basis. If you have other tips or wish to share your favorite configurations and methods, feel free to use the comments below.