How To Configure Custom Connection Options for your SSH Client

By: Justin Ellingwood

Introduction

SSH, or secure shell, is the most common way of connecting to Linux hosts for remote administration. Although the basics of connecting to a single host are often rather straight forward, this can become unwieldy and a much more complicated task when you begin working with a large number of remote systems.

Fortunately, OpenSSH allows you to provide customized client-side connection options. These can be saved to a configuration file that can be used to define per-host values. This can help keep the different connection options you use for each host separated and organized, and can keep you from having to provide extensive options on the command line whenever you need to connect.

In this guide, we'll cover the basics of the SSH client configuration file, and go over some common options.

Prerequisites

To complete this guide, you will need a working knowledge of SSH and some of the options that you can provide when connecting. You may also wish to configure SSH key-based authentication for some of your users or hosts, at the very least for testing purposes.

The SSH Config File Structure and Interpretation Algorithm

Each user on your local system can maintain a client-side SSH configuration file. These can contain any options that you would use on the command line to specify connection parameters, allowing you to store your common connection items and process them automatically on connection. It is always possible to override the values defined in the configuration file at the time of the connection through normal flags to the `ssh` command.

The Location of the SSH Client Config File

The client-side configuration file is called `config` and it is located in your user's home directory within the `.ssh` configuration directory. Often, this file is not created by default, so you may need to create it yourself:

```
touch ~/.ssh/config
```

Configuration File Structure

The `config` file is organized by hosts. Each host definition can define connection options for the specific matching host. Wildcards are also available to allow for options that should have a broader scope.

Each of the sections starts with a header defining the hosts that should match the configuration options that will follow. The specific configuration items for that matching host are then defined below. Only items that differ from the default values need to be specified, as the host will inherit the defaults for any undefined items. A section is defined from the `Host` header to the following `Host` header.

Typically, for organizational purposes and readability, the options being set for each host are indented. This is not a hard requirement, but a useful convention that allows for easier interpretation at a glance.

The general format will look something like this:

```
Host firsthost
    SSH_OPTION_1 custom_value
    SSH_OPTION_2 custom_value
    SSH_OPTION_3 custom_value

Host secondhost
    ANOTHER_OPTION custom_value

Host *host
    ANOTHER_OPTION custom_value

Host *
    CHANGE_DEFAULT custom_value
```

Here, we have four sections that will be applied on each connection attempt depending on whether the host in question matches.

Interpretation Algorithm

It is very important to understand the way that SSH will interpret the file to apply the configuration values defined within. This has large implications when using wildcards and the `Host *` generic host definition.

SSH will match the hostname given on the command line with each of the `Host` headers that define configuration sections. It will do this from the top of the file

downwards, so order is incredibly important.

This is a good time to point out that the patterns in the `Host` definition do not have to match the actual host that you will be connecting with. You can essentially use these definitions to set up aliases for hosts that can be used in lieu of the actual host name.

For example, consider this definition:

```
Host devel
    HostName devel.example.com
    User tom
```

This host allows us to connect as `tom@devel.example.com` by typing this on the command line:

```
ssh devel
```

With this in mind, we can now discuss the way in which SSH applies each configuration option as it moves down the file. It starts at the top and checks each `Host` definition to see if it matches the value given on the command line.

When the first matching `Host` definition is found, each of the associated SSH options are applied to the upcoming connection. The interpretation does not end here though.

SSH then moves down the file, checking to see if other `Host` definitions also match. If another definition is found that matches the current hostname given on the command line, it will consider the SSH options associated with the new section. It will then apply any SSH options defined for the new section that have not already been defined by previous sections.

This last point is extremely important to internalize. SSH will interpret each of the `Host` sections that match the hostname given on the command line, in order. During this process, it will always use the *first* value given for each option. There is no way to override a value that has already been given by a previously matched section.

This means that your `config` file should follow the simple rule of having the most specific configurations at the top. More general definitions should come later on in order to apply options that were not defined by the previous matching sections.

Let's look again at the mock-up `config` file we used in the last section:

```
Host firsthost
    SSH_OPTION_1 custom_value
    SSH_OPTION_2 custom_value
    SSH_OPTION_3 custom_value
```

```
Host secondhost
    ANOTHER_OPTION custom_value

Host *host
    ANOTHER_OPTION custom_value

Host *
    CHANGE_DEFAULT custom_value
```

Here, we can see that the first two sections are defined by literal hostnames (or aliases), meaning that they do not use any wildcards. If we connect using `ssh firsthost`, the very first section will be the first to be applied. This will set `SSH_OPTION_1`, `SSH_OPTION_2`, and `SSH_OPTION_3` for this connection.

It will check the second section and find that it does not match and move on. It will then find the third section and find that it matches. It will check `ANOTHER_OPTION` to see if it already has a value for that from previous sections. Finding that it doesn't, it will apply the value from this section. It will then match the last section since the `Host *` definition matches every connection. Since it doesn't have a value for the mock `CHANGE_DEFAULT` option from other sections, it will take the value from this section. The connection is then made with the options collected from this process.

Let's try this again, pretending to call `ssh secondhost` from the command line.

Again, it will start at the first section and check whether it matches. Since this matches only a connection to `firsthost`, it will skip this section. It will move on to the second section. Upon finding that this section matches the request, it will collect the value of `ANOTHER_OPTION` for this connection.

SSH then looks at the third definition and find that the wildcard matches the current connection. It will then check whether it already has a value for `ANOTHER_OPTION`. Since this option was defined in the second section, which was already matched, the value from the third section is dropped and has no effect.

SSH then checks the fourth section and applies the options within that have not been defined by previously matched sections. It then attempts the connection using the values it has gathered.

Basic Connection Options

Now that you have an idea about the general format you should use when designing your configuration file, let's discuss some common options and the format to use to specify them on the command line.

The first ones we will cover are the basic information necessary to connect to a

remote host. Namely, the hostname, username, and port that the SSH daemon is running on.

To connect as a user named `apollo` to a host called `example.com` that runs its SSH daemon on port `4567` from the command line, we could give the variable information in a variety of ways. The most common would probably be:

```
ssh -p 4567 apollo@example.com
```

However, we could also use the full option names with the `-o` flag, like this:

```
ssh -o "User=apollo" -o "Port=4567" -o "HostName=example.com" anything
```

Here, we have set all of the options we wish to use with the `-o` flag. We have even specified the host as "anything" as an alias just as we could in the config file as we described above. The actual hostname is taken from the `HostName` option that we are setting.

The capitalized option names that we are using in the second form are the same that we must use in our `config` file. You can find a full list of available options by typing:

```
man ssh_config
```

To set these in our `config` file, we first must decide which hosts we want these options to be used for. Since we are discussing options that are specific to the host in question, we should probably use a literal host match.

We also have an opportunity at this point to assign an alias for this connection. Let's take advantage of that so that we do not have to type the entire hostname each time. We will use the alias "home" to refer to this connection and the associated options:

```
Host home
```

Now, we can define the connection details for this host. We can use the second format we used above to inform us as to what we should put in this section.

```
Host home
    HostName example.com
    User apollo
    Port 4567
```

We define options using a key-value system. Each pair should be on a separate line. Keys can be separated from their associated values either by white space, or by an equal sign with optional white space. Thus, these are all identical as interpreted by our SSH client:

```
Port 4567
Port=4567
Port = 4567
```

The only difference is that depending on the option and value, using the equal sign with no spaces can allow you to specify an option on the command line without quoting. Since we are focusing on our `config` file, this is entirely up to your preferences.

So far, the configuration we have designed is incredibly simple. In its entirety, it looks like this:

```
Host home
    HostName example.com
    User apollo
    Port 4567
```

What if we use the same username on both our work and home computers? We could add redundant options with our section defining the work machine like this:

```
Host home
    HostName example.com
    User apollo
    Port 4567

Host work
    HostName company.com
    User apollo
```

This works, but we are repeating values. This is only a single option, so it is not a huge deal, but sometimes we want to share a large number of options. The best way of doing that is to break the shared options out into separate sections.

If we use the username "apollo" on all of the machines that we connect to, we could place this into our generic "Host" definition marked by a single `*` that matches every connection. Remember that the more generic sections should go further towards the bottom:

```
Host home
    HostName example.com
    Port 4567

Host work
    HostName company.com

Host *
    User apollo
```

This clears up the issue of repetition in our configuration and will work if "apollo" is

your default username for the majority of new systems you connect to.

What if there are some systems that do not use this username? There are a few different ways that you can approach this, depending on how widely the username is shared.

If the "apollo" username is used on *almost* all of your hosts, it's probably best to leave it in the generic `Host *` section. This will apply to any hosts that have not received a username from sections above. For our anomalous machines that use a different username, we can override the default by providing an alternative. This will take precedence as long as it is defined before the generic section:

```
Host home
    HostName example.com
    Port 4567

Host work
    HostName company.com

Host oddity
    HostName weird.com
    User zeus

Host *
    User apollo
```

For the `oddity` host, SSH will connect using the username "zeus". All other connections will not receive their username until they hit the generic `Host *` definition.

What happens if the "apollo" username is shared by a few connections, but isn't common enough to use as a default value? If we are willing to rename the aliases that we are using to have a more common format, we can use a wildcard to apply additional options to just these two hosts.

We can change the `home` alias to something like `hapollo` and the work connection to something like `wapollo`. This way, both hosts share the `apollo` portion of their alias, allowing us to target it with a different section using wildcards:

```
Host hapollo
    HostName example.com
    Port 4567

Host wapollo
```

```
    HostName company.com

Host *apollo
    User apollo

Host *
    User diffdefault
```

Here, we have moved the shared `User` definition to a host section that matches SSH connections trying to connect to hosts that end in `apollo`. Any connection not ending in `apollo` (and without its own `Host` section defining a `User`) will receive the username `diffdefault`.

Note that we have retained the ordering from most specific to least specific in our file. It is best to think of less specific Host sections as fallbacks as opposed to defaults due to the order in which the file is interpreted.

Common SSH Configuration Options

So far, we have discussed some of the basic options necessary to establish a connection. We have covered these options:

- `HostName`: The actual hostname that should be used to establish the connection. This replaces any alias defined in the `Host` header. This option is *not* necessary if the `Host` definition specifies the actual valid hostname to connect to.
- `User`: The username to be used for the connection.
- `Port`: The port that the remote SSH daemon is running on. This option is only necessary if the remote SSH instance is not running on the default port `22`.

There are many other useful options worth exploring. We will discuss some of the more common options, separated according to function.

General Tweaks and Connection Items

Some other tweaks that you may wish to configure on a broad level, perhaps in the `Host *` section, are below.

- **`ServerAliveInterval`**: This option can be configured to let SSH know when to send a packet to test for a response from the sever. This can be useful if your connection is unreliable and you want to know if it is still available.
- **`LogLevel`**: This configures the level of detail in which SSH will log on the client-side. This can be used for turning off logging in certain situations or increasing the verbosity when trying to debug. From least to most verbose, the levels are QUIET,

FATAL, ERROR, INFO, VERBOSE, DEBUG1, DEBUG2, and DEBUG3.

- **`StrictHostKeyChecking`**: This option configures whether ssh SSH will ever automatically add hosts to the `~/.ssh/known_hosts` file. By default, this will be set to "ask" meaning that it will warn you if the Host Key received from the remote server does not match the one found in the `known_hosts` file. If you are constantly connecting to a large number of ephemeral hosts, you may want to turn this to "no". SSH will then automatically add any hosts to the file. This can have security implications, so think carefully before enabling it.
- **`UserKnownHostsFile`**: This option specifies the location where SSH will store the information about hosts it has connected to. Usually you do not have to worry about this setting, but you may wish to set this to `/dev/null` if you have turned off strict host checking above.
- **`VisualHostKey`**: This option can tell SSH to display an ASCII representation of the remote host's key upon connection. Turning this on can be an easy way to get familiar with your host's key, allowing you to easily recognize it if you have to connect from a different computer sometime in the future.
- **`Compression`**: Turning compression on can be helpful for very slow connections. Most users will not need this.

  With the above configuration items in mind, we could make a number of useful configuration tweaks.

  For instance, if we are creating and destroying hosts very quickly at a cloud provider, something like this may be useful:

```
Host home
    VisualHostKey yes

Host cloud*
    StrictHostKeyChecking no
    UserKnownHostsFile /dev/null
    LogLevel QUIET

Host *
    StrictHostKeyChecking ask
    UserKnownHostsFile ~/.ssh/known_hosts
    LogLevel INFO
    ServerAliveInterval 120
```

  This will turn on your visual host key for your home connection, allowing you to become familiar with it so you can recognize if it changes or when connecting from a

different machine. We have also set up any host that begins with cloud* to not check hosts and not log failures. For other hosts, we have sane fallback values.

Connection Forwarding

One common use of SSH is forwarding connections, either allowing a local connection to tunnel through the remote host, or allowing the remote machine access to tunnel through the local machine. SSH can also do dynamic forwarding using protocols like SOCKS5 which include the forwarding information for the remote host.

The options that control this behavior are:

- **`LocalForward`**: This option is used to specify a connection that will forward a local port's traffic to the remote machine, tunneling it out into the remote network. The first argument should be the local port you wish to direct traffic to and the second argument should be the address and port that you wish to direct that traffic to on the remote end.
- **`RemoteForward`**: This option is used to define a remote port where traffic can be directed to in order to tunnel out of the local machine. The first argument should be the remote port where traffic will be directed on the remote system. The second argument should be the address and port to point the traffic to when it arrives on the local system.
- **`DynamicForward`**: This is used to configure a local port that can be used with a dynamic forwarding protocol like SOCKS5. Traffic using the dynamic forwarding protocol can then be directed at this port on the local machine and on the remote end, it will be routed according to the included values.

These options can be used to forward ports in both directions, as you can see here:

```
# This will allow us to use port 8080 on the local machine
# in order to access example.com at port 80 from the remote machine
Host local_to_remote
    LocalForward 8080 example.com:80

# This will allow us to offer access to internal.com at port 443
# to the remote machine through port 7777 on the other side
Host remote_to_local
    RemoteForward 7777 internal.com:443
```

Other Forwarding

Along with connection forwarding, SSH allows other types of forwarding as well.

We can forward any SSH keys stored in an agent on our local machine, allowing us to

connect from the remote system as using credentials stored on our local system. We can also start applications on a remote system and forward the graphical display to our local system using X11 forwarding.

These are the directives that are associated with these capabilities:

- **`ForwardAgent`**: This option allows authentication keys stored on our local machine to be forwarded onto the system you are connecting to. This can allow you to hop from host-to-host using your home keys.
- **`ForwardX11`**: If you want to be able to forward a graphical screen of an application running on the remote system, you can turn this option on.

These both are "yes" or "no" options.

Specifying Keys

If you have SSH keys configured for your hosts, these options can help you manage which keys to use for each host.

- **`IdentityFile`**: This option can be used to specify the location of the key to use for each host. If your keys are in the default locations, each will be tried and you will not need to adjust this. If you have a number of keys, each devoted to different purposes, this can be used to specify the exact path where the correct key can be found.
- **`IdentitiesOnly`**: This option can be used to force SSH to only rely on the identities provided in the `config` file. This may be necessary if an SSH agent has alternative keys in memory that are not valid for the host in question.

These options are especially useful if you have to keep track of a large number of keys for different hosts and use one or more SSH agents to assist.

Multiplexing SSH Over a Single TCP Connection

SSH has the ability to use a single TCP connection for multiple SSH connections to the same host machine. This can be useful if it takes awhile to establish a TCP handshake to the remote end as it removes this overhead from additional SSH connections.

The following options can be used to configure multiplexing with SSH:

- **`ControlMaster`**: This option tells SSH whether to allow multiplexing when possible. Generally, if you wish to use this option, you should set it to "auto" in either the host section that is slow to connect or in the generic `Host *` section.
- **`ControlPath`**: This option is used to specify the socket file that is used to control the connections. It should be to a location on the filesystem. Generally, this is given using

SSH variables to easily label the socket by host. To name the socket based on username, remote host, and port, you can use `/path/to/socket/%r@%h:%p`.

- **`ControlPersist`**: This option establishes the amount of time in seconds that the TCP connection should remain open after the final SSH connection has been closed. Setting this to a high number will allow you to open new connections after closing the first, but you can usually set this to something low like "1" to avoid keeping an unused TCP connection open.

  Generally, you can set this up using a section that looks something like this:

  ```
  Host *
      ControlMaster auto
      ControlPath ~/.ssh/multiplex/%r@%h:%p
      ControlPersist 1
  ```

  Afterwards, you should make sure that the directory is created:

  ```
  mkdir -p ~/.ssh/multiplex
  ```

  If you wish to not use multiplexing for a specific connection, you can select no multiplexing on the command line like this:

  ```
  ssh -S none user@host
  ```

Conclusion

By now, it should be clear that you can heavily customize the options you use to connect to remote hosts. As long as you keep in mind the way that SSH will interpret the values, you can establish rich sets of specific values with reasonable fall backs.