Introduction

Linux servers are often administered remotely using SSH by connecting to an OpenSSH server, which is the default SSH server software used within Ubuntu, Debian, CentOS, FreeBSD, and most other Linux/BSD-based systems.

OpenSSH server is the server side of SSH, also known as SSH daemon or `sshd`. You can connect to an OpenSSH server using the OpenSSH client—the `ssh` command. You can learn more about the SSH client-server model in SSH Essentials: Working with SSH Servers, Clients, and Keys. Properly securing your OpenSSH server is very important, as it acts as the front door or entry into your server.

In this tutorial, you will harden your OpenSSH server by using different configuration options to ensure that remote access to your server is as secure as possible.

Prerequisites

To complete this tutorial, you will need:

•An Ubuntu 18.04 server set up by following the Initial Server Setup with Ubuntu 18.04, including a sudo non-root user.
Once you have this ready, log in to your server as your non-root user to begin.

Step 1 — General Hardening

In this first step, you will implement some initial hardening configurations to improve the overall security of your SSH server.

The exact hardening configuration that is most suitable for your own server depends heavily on your own threat model and risk threshold. However, the configuration you'll use in this step is a general secure configuration that will suit the majority of servers.

Many of the hardening configurations for OpenSSH you implement using the standard OpenSSH server configuration file, which is located at `/etc/ssh/sshd_config`. Before continuing with this tutorial, it is recommended to take a backup of your existing configuration file, so that you can restore it in the unlikely event that something goes wrong.

Take a backup of the file using the following command:

```
sudo cp /etc/ssh/sshd_config /etc/ssh/sshd_config.bak
```

This will save a backup copy of the file to `/etc/ssh/sshd_config.bak`.

Before editing your configuration file, you can review the options that are currently set. To do this, run the following command:

```
sudo sshd -T
```

This will run OpenSSH server in extended test mode, which will validate the full configuration file and print out the effective configuration values.

You can now open the configuration file using your favorite text editor to begin implementing the initial hardening measures:

```
sudo nano /etc/ssh/sshd_config
```

Note: The OpenSSH server configuration file includes many default options and configurations. Depending on your existing server configuration, some of the recommended hardening options may already have been set.

When editing your configuration file, some options may be commented out by default using a single hash character (#) at the start of the line. In order to edit these options, or have the commented option be recognized, you'll need to uncomment them by removing the hash.

Firstly, disable logging in via SSH as the **root** user by setting the following option:

sshd_config

```
PermitRootLogin no
```

This is massively beneficial, as it will prevent a potential attacker from logging in directly as root. It also encourages good operational security practices, such as operating as a non-privileged user and using `sudo` to escalate privileges only when absolutely needed.

Next, you can limit the maximum number of authentication attempts for a particular login session by configuring the following:

sshd_config

```
MaxAuthTries 3
```

A standard value of `3` is acceptable for most setups, but you may wish to set this higher or lower depending on your own risk threshold.

If required, you can also set a reduced login grace period, which is the amount of time a user has to complete authentication after initially connecting to your SSH server:

sshd_config

```
LoginGraceTime 20
```

The configuration file specifies this value in seconds.

Setting this to a lower value helps to prevent certain denial-of-service attacks where multiple authentication sessions are kept open for a prolonged period of time.

If you have configured SSH keys for authentication, rather than using passwords, disable SSH password authentication to prevent leaked user passwords from allowing an attacker to log in:

sshd_config

```
PasswordAuthentication no
```

As a further hardening measure related to passwords, you may also wish to disable authentication with empty passwords. This will prevent logins if a user's password is set to a blank or empty value:

sshd_config

```
PermitEmptyPasswords no
```

In the majority of use cases, SSH will be configured with public key authentication as the only in-use authentication method. However, OpenSSH server also supports many other authentication methods, some of which are enabled by default. If these are not required, you can disable them to further reduce the attack surface of your SSH server:

sshd_config

```
ChallengeResponseAuthentication no
KerberosAuthentication no
GSSAPIAuthentication no
```

If you'd like to know more about some of the additional authentication methods available within SSH, you may wish to review these resources:

•Challenge Response Authentication
•Kerberos Authentication
•GSSAPI Authentication

X11 forwarding allows for the display of remote graphical applications over an SSH connection, but this is rarely used in practice. It is recommended to disable it if it isn't needed on your server:

sshd_config

```
X11Forwarding no
```

OpenSSH server allows connecting clients to pass custom environment variables, that is, to set a `$PATH` or to configure terminal settings. However, like X11 forwarding, these are not commonly used, so can be disabled in most cases:

sshd_config

```
PermitUserEnvironment no
```

If you decide to configure this option, you should also make sure to comment out any references to `AcceptEnv` by adding a hash (`#`) to the beginning of the line.

Next, you can disable several miscellaneous options related to tunneling and forwarding if you won't be using these on your server:

sshd_config

```
AllowAgentForwarding no
AllowTcpForwarding no
PermitTunnel no
```

Finally, you can disable the verbose SSH banner that is enabled by default, as it shows various information about your system, such as the operating system version:

sshd_config

```
DebianBanner no
```

Note that this option most likely won't already be present in the configuration file, so you may need to add it manually. Save and exit the file once you're done.

Now validate the syntax of your new configuration by running `sshd` in test mode:

```
sudo sshd -t
```

If your configuration file has a valid syntax, there will be no output. In the event of a syntax error, there will be an output describing the issue.

Once you're satisfied with your configuration file, you can reload `sshd` to apply the new settings:

```
sudo service sshd reload
```

In this step, you completed some general hardening of your OpenSSH server configuration file. Next, you'll implement an IP address allowlist to further restrict who can log in to your server.

Step 2 — Implementing an IP Address Allowlist

You can use IP address allowlists to limit the users who are authorized to log in to your server on a per-IP address basis. In this step, you will configure an IP allowlist for your OpenSSH server.

In many cases, you will only be logging on to your server from a small number of known, trusted IP addresses. For example, your home internet connection, a corporate VPN appliance, or a static jump box or bastion host in a data center.

By implementing an IP address allowlist, you can ensure that people will only be able to log in from one of the pre-approved IP addresses, greatly reducing the risk of a breach in the event that your private keys and/or passwords are leaked.

Note: Please take care in identifying the correct IP addresses to add to your allowlist, and ensure that these are not floating or dynamic addresses that may regularly change, for example as is often seen with consumer internet service providers.

You can identify the IP address that you're currently connecting to your server with by using the `w` command:

```
w
```

This will output something similar to the following:

```
Output
 14:11:48 up 2 days, 12:25,  1 user,  load average: 0.00, 0.00, 0.00
        USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
```

```
your_username     pts/0    203.0.113.1     12:24    1.00s  0.20s  0.00s w
```

Locate your user account in the list and take a note of the connecting IP address. Here we use the example IP of `203.0.113.1`
In order to begin implementing your IP address allowlist, open the OpenSSH server configuration file in your favorite text editor:

```
sudo nano /etc/ssh/sshd_config
```

You can implement IP address allowlists using the `AllowUsers` configuration directive, which restricts user authentications based on username and/or IP address.

Your own system setup and requirements will determine which specific configuration is the most appropriate. The following examples will help you to identify the most suitable one:

•Restrict all users to a specific IP address:

```
AllowUsers *@203.0.113.1
```

•Restrict all users to a specific IP address range using Classless Inter-Domain Routing (CIDR) notation:

```
AllowUsers *@203.0.113.0/24
```

•Restrict all users to a specific IP address range (using wildcards):

```
AllowUsers *@203.0.113.*
```

•Restrict all users to multiple specific IP addresses and ranges:

```
AllowUsers *@203.0.113.1 *@203.0.113.2 *@192.0.2.0/24 *@172.16.*.1
```

•Disallow all users except for named users from specific IP addresses:

```
AllowUsers sammy@203.0.113.1 alex@203.0.113.2
```

•Restrict a specific user to a specific IP address, while continuing to allow all other users to log in without restrictions:

```
Match User ashley
  AllowUsers ashley@203.0.113.1
```

Warning: Within an OpenSSH configuration file, all configurations under a `Match` block will only apply to connections that match the criteria, regardless of indentation or line breaks. This means that you must be careful and ensure that configurations intended to apply globally are not accidentally put within a `Match` block. It is recommended to put all `Match` blocks at the bottom/end of your configuration file to help avoid this.

Once you have finalized your configuration, add it to the bottom of your OpenSSH server configuration file:

sshd_config

```
AllowUsers *@203.0.113.1
```

Save and close the file, and then proceed to test your configuration syntax:

```
sudo sshd -t
```

If no errors are reported, you can reload OpenSSH server to apply your configuration:

```
sudo service sshd reload
```

In this step, you implemented an IP address allowlist on your OpenSSH server. Next, you will restrict the shell of a user to limit the commands that they are allowed to use.

Step 3 — Restricting the Shell of a User

In this step, you'll look at the various options for restricting the shell of an SSH user.

In addition to providing remote shell access, SSH is also great for transferring files and other data, for example, via SFTP. However, you may not always want to grant full shell access to users when they only need to be able to carry out file transfers.

There are multiple configurations within OpenSSH server that you can use to restrict the shell environment of particular users. For instance, in this tutorial, we will use these to create SFTP-only users.

Firstly, you can use the `/usr/sbin/nologin` shell to disable interactive logins for certain user accounts, while still allowing non-interactive sessions to function, like file transfers, tunneling, and so on.

To create a new user with the `nologin` shell, use the following command:

```
sudo adduser --shell /usr/sbin/nologin alex
```

Alternatively, you can change the shell of an existing user to be `nologin`:

```
sudo usermod --shell /usr/sbin/nologin sammy
```

If you then attempt to interactively log in as one of these users, the request will be rejected:

```
sudo su alex
```

This will output something similar to the following message:

```
Output
This account is currently not available.
```

Despite the rejection message on interactive logins, other actions such as file transfers will still be allowed.

Next, you should combine your usage of the `nologin` shell with some additional configuration options to further restrict the relevant user accounts.

Begin by opening the OpenSSH server configuration file in your favorite text editor again:

```
sudo nano /etc/ssh/sshd_config
```

There are two configuration options that you can implement together to create a tightly restricted SFTP-only user account: `ForceCommand internal-sftp` and `ChrootDirectory`.

The `ForceCommand` option within OpenSSH server forces a user to execute a specific command upon login. This can be useful for certain machine-to-machine communications, or to forcefully launch a particular program.

However, in this case, the `internal-sftp` command is particularly useful. This is a special function of OpenSSH server that launches a basic in-place SFTP daemon that doesn’t require any supporting system files or configuration.

This should ideally be combined with the `ChrootDirectory` option, which will override/change the perceived root directory for a particular user, essentially restricting them to a specific directory on the system.

Add the following configuration section to your OpenSSH server configuration file for this:

sshd_config

```
Match User alex
  ForceCommand internal-sftp
  ChrootDirectory /home/alex/
```

Warning: As noted in Step 2, within an OpenSSH configuration file, all configurations under a `Match` block will only apply to connections that match the criteria, regardless of indentation or line breaks. This means that you must be careful and ensure that configurations intended to apply globally are not accidentally put within a `Match` block. It is recommended to put all `Match` blocks at the bottom/end of your configuration file to help avoid this.

Save and close your configuration file, and then test your configuration again:

```
sudo sshd -t
```

If there are no errors, you can then apply your configuration:

```
sudo service sshd reload
```

This has created a robust configuration for the `alex` user, where interactive logins are disabled, and all SFTP activity is restricted to the home directory of the user. From the perspective of the user, the root of the system, that is, `/`, is their home directory, and they will not be able to traverse up the file system to access other areas.
You've implemented the `nologin` shell for a user and then created a configuration to restrict SFTP access to a specific directory.

Step 4 — Advanced Hardening

In this final step, you will implement various additional hardening measures to make access to your SSH server as secure as possible.

A lesser-known feature of OpenSSH server is the ability to impose restrictions on a per-key basis, that is restrictions that apply only to specific public keys present in the `.ssh/authorized_keys` file. This is particularly useful to control access for machine-to-machine sessions, as well as providing the ability for non-sudo users to control the restrictions for their own user account.

You can apply most of these restrictions at the system or user level too, however it is still advantageous to implement them at the key-level as well, to provide defence-in-depth and an additional failsafe in the event of accidental system-wide configuration errors.

Note: You can only implement these additional security configurations if you're using SSH public-key authentication. If you're only using password authentication, or have a more complex setup such as an SSH certificate authority, unfortunately these will not be usable.

Begin by opening your `.ssh/authorized_keys` file in your favorite text editor:

```
nano ~/.ssh/authorized_keys
```

Note: Since these configurations apply on a per-key basis, you'll need to edit each individual key within each individual `authorized_keys` file that you want them to apply to, for all users on your system. Usually you will only need to edit one key/file, but this is worth considering if you have a complex multi-user system.

Once you've opened your `authorized_keys` file, you will see that each line contains an SSH public key, which will most likely begin with something like `ssh-rsa AAAB...`. Additional configuration options can be added to the beginning of the line, and these will only apply to successful authentications against that specific public key.

The following restriction options are available:

- `no-agent-forwarding`: Disable SSH agent forwarding.
- `no-port-forwarding`: Disable SSH port forwarding.
- `no-pty`: Disable the ability to allocate a tty (i.e. start a shell).
- `no-user-rc`: Prevent execution of the `~/.ssh/rc` file.
- `no-X11-forwarding`: Disable X11 display forwarding.

You can apply these to disable specific SSH features for specific keys. For example, to disable agent forwarding and X11 forwarding for a key, you would use the following configuration:

~/.ssh/authorized_keys

```
no-agent-forwarding,no-X11-forwarding ssh-rsa AAAB...
```

By default, these configurations work using an “allow by default, block by exception” methodology; however, it is also possible to use “block by default, allow by exception,” which is generally preferable for ensuring security.

You can do this by using the `restrict` option, which will implicitly deny all SSH features for the specific key, requiring them to be explicitly re-enabled only where absolutely needed. You can re-enable features using the same configuration options described earlier in this tutorial, but without the `no-` prefix.

For example, to disable all SSH features for a particular key, apart from X11 display forwarding, you can use the following configuration:

~/.ssh/authorized_keys

```
restrict,X11-forwarding ssh-rsa AAAB...
```

You may also wish to consider using the `command` option, which is very similar to the `ForceCommand` option described in Step 3. This doesn’t provide a direct benefit if you’re already using `ForceCommand`, but it is good defense-in-depth to have it in place, just in the unlikely event that your main OpenSSH server configuration file is overwritten, edited, and so on.

For example, to force users authenticating against a specific key to execute a specific command upon login, you can add the following configuration:

~/.ssh/authorized_keys

```
command="top" ssh-rsa AAAB...
```

Warning: The `command` configuration option acts purely as a defense-in-depth method, and should not be solely relied on to restrict the activities of an SSH user, as there are potentially ways to override or bypass it depending on your

environment. Instead, you should use the configuration in tandem with the other controls described in this article.

Finally, to best use the per-key restrictions for the SFTP-only user that you created in Step 3, you can use the following configuration:

~/.ssh/authorized_keys

```
restrict,command="false" ssh-rsa AAAB...
```

The `restrict` option will disable all interactive access, and the `command="false"` option acts as a second line of defense in the event that the `ForceCommand` option or `nologin` shell were to fail.

Save and close the file to apply the configuration. This will take effect immediately for all new logins, so you don't need to reload OpenSSH manually.

In this final step, you implemented some additional advanced hardening measures for OpenSSH server by using the custom options within your `.ssh/authorized_keys` file(s).

Conclusion

In this article, you reviewed your OpenSSH server configuration and implemented various hardening measures to help secure your server.

This will have reduced the overall attack surface of your server by disabling unused features and locking down the access of specific users.

You may wish to review the manual pages for OpenSSH server and its associated configuration file, to identify any potential further tweaks that you want to make.