OpenSSH Server

Introduction

OpenSSH is a powerful collection of tools for the remote control of, and transfer of data between, networked computers. You will also learn about some of the configuration settings possible with the OpenSSH server application and how to change them on your Ubuntu system.

OpenSSH is a freely available version of the Secure Shell (SSH) protocol family of tools for remotely controlling, or transferring files between, computers. Traditional tools used to accomplish these functions, such as telnet or rcp, are insecure and transmit the user's password in cleartext when used. OpenSSH provides a server daemon and client tools to facilitate secure, encrypted remote control and file transfer operations, effectively replacing the legacy tools.

The OpenSSH server component, sshd, listens continuously for client connections from any of the client tools. When a connection request occurs, sshd sets up the correct connection depending on the type of client tool connecting. For example, if the remote computer is connecting with the ssh client application, the OpenSSH server sets up a remote control session after authentication. If a remote user connects to an OpenSSH server with scp, the OpenSSH server daemon initiates a secure copy of files between the server and client after authentication. OpenSSH can use many authentication methods, including plain password, public key, and Kerberos tickets.

Installation

Installation of the OpenSSH client and server applications is simple. To install the OpenSSH client applications on your Ubuntu system, use this command at a terminal prompt:

```
sudo apt install openssh-client
```

To install the OpenSSH server application, and related support files, use this command at a terminal prompt:

```
sudo apt install openssh-server
```

Configuration

You may configure the default behavior of the OpenSSH server application, sshd, by editing the file `/etc/ssh/sshd_config`. For information about the configuration directives used in this file, you may view the appropriate manual page with the following command, issued at a terminal prompt:

```
man sshd_config
```

There are many directives in the sshd configuration file controlling such things as communication settings, and authentication modes. The following are examples of configuration directives that can be changed by editing the `/etc/ssh/sshd_config` file.

> **Tip**
>
> Prior to editing the configuration file, you should make a copy of the original file and protect it from writing so you will have the original settings as a reference and to reuse as necessary.
>
> Copy the `/etc/ssh/sshd_config` file and protect it from writing with the following commands, issued at a terminal prompt:
>
> ```
> sudo cp /etc/ssh/sshd_config /etc/ssh/sshd_config.original
> sudo chmod a-w /etc/ssh/sshd_config.original
> ```

Furthermore since losing an ssh server might mean losing your way to reach a server, check the configuration after changing it and before restarting the server:

```
sudo sshd -t -f /etc/ssh/sshd_config
```

The following is an *example* of a configuration directive you may change:

- To make your OpenSSH server display the contents of the `/etc/issue.net` file as a pre-login banner, simply add or modify this line in the `/etc/ssh/sshd_config` file:

  Banner /etc/issue.net

After making changes to the `/etc/ssh/sshd_config` file, save the file, and restart the sshd server application to effect the changes using the following command at a terminal prompt:

```
sudo systemctl restart sshd.service
```

> **Warning**
>
> Many other configuration directives for sshd are available to change the server application's behavior to fit your needs. Be advised, however, if your only method of access to a server is ssh, and you make a mistake in configuring sshd via the `/etc/ssh/sshd_config` file, you may find you are locked out of the server upon restarting it. Additionally, if an incorrect configuration directive is supplied, the sshd server may refuse to start, so be extra careful when editing this file on a remote server.

SSH Keys

SSH allow authentication between two hosts without the need of a password. SSH key authentication uses a *private key* and a *public key*.

To generate the keys, from a terminal prompt enter:

```
ssh-keygen -t rsa
```

This will generate the keys using the *RSA Algorithm*. At the time of this writing, the generated keys will have 3072 bits. You can modify the number of bits by using the `-b` option. For example, to generate keys with 4096 bits, you can do:

```
ssh-keygen -t rsa -b 4096
```

During the process you will be prompted for a password. Simply hit *Enter* when prompted to create the key.

By default the *public* key is saved in the file `~/.ssh/id_rsa.pub`, while `~/.ssh/id_rsa` is the *private* key. Now copy the `id_rsa.pub` file to the remote host and append it to `~/.ssh/authorized_keys` by entering:

```
ssh-copy-id username@remotehost
```

Finally, double check the permissions on the `authorized_keys` file, only the authenticated user should have read and write permissions. If the permissions are not correct change them by:

```
chmod 600 .ssh/authorized_keys
```

You should now be able to SSH to the host without being prompted for a password.

Import keys from public keyservers

These days many users have already ssh keys registered with services like launchpad or github. Those can be easily imported with:

```
ssh-import-id <username-on-remote-service>
```

The prefix `lp:` is implied and means fetching from launchpad, the alternative `gh:` will make the tool fetch from github instead.

Two factor authentication with U2F/FIDO

OpenSSH 8.2 added support for U2F/FIDO hardware authentication devices <https://www.openssh.com/txt/release-8.2> . These devices are used to provide an extra layer of security on top of the existing key-based authentication, as the hardware token needs to be present to finish the authentication.

It's very simple to use and setup. The only extra step is generate a new keypair that can be used with the hardware device. For that, there are two key types that can be used: `ecdsa-sk` and `ed25519-sk`. The former has broader hardware support, while the latter might need a more recent device.

Once the keypair is generated, it can be used as you would normally use any other type of key in openssh. The only requirement is that in order to use the private key, the U2F device has to be present on the host.

For example, plug the U2F device in and generate a keypair to use with it:

```
$ ssh-keygen -t ecdsa-sk
Generating public/private ecdsa-sk key pair.
You may need to touch your authenticator to authorize key generation. <-- touch device
Enter file in which to save the key (/home/ubuntu/.ssh/id_ecdsa_sk):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ubuntu/.ssh/id_ecdsa_sk
Your public key has been saved in /home/ubuntu/.ssh/id_ecdsa_sk.pub
The key fingerprint is:
SHA256:V9PQ1MqaU8FODXdHqDiH9Mxb8XK3o5aVYDQLVl9IFRo ubuntu@focal
```

Now just transfer the public part to the server to `~/.ssh/authorized_keys` and you are ready to go:

```
$ ssh -i .ssh/id_ecdsa_sk ubuntu@focal.server
Confirm user presence for key ECDSA-SK SHA256:V9PQ1MqaU8FODXdHqDiH9Mxb8XK3o5aVYDQLVl9IFRo <-- touch device
Welcome to Ubuntu Focal Fossa (GNU/Linux 5.4.0-21-generic x86_64)
(...)
ubuntu@focal.server:~$
```

FIDO2 resident keys

FIDO2 private keys consist of two parts: a "key handle" part stored in the private key file on disk, and a per-device key that is unique to each FIDO2 token and that cannot be exported from the token hardware. These are combined by the hardware at authentication time to derive the real key that is used to sign authentication challenges.

For tokens that are required to move between computers, it can be cumbersome to have to move the private key file first. To avoid this, tokens implementing the newer FIDO2 standard support *resident keys*, where it is possible to retrieve the key handle part of the key from the hardware.

Using resident keys increases the likelihood of an attacker being able to use a stolen token device. For this reason, tokens normally enforce PIN authentication before allowing download of keys, and users should set a PIN on their tokens before creating any resident keys. This is done via the hardware token management software.

OpenSSH allows resident keys to be generated using the ssh-keygen`-O resident` flag at key generation time:

```
$ ssh-keygen -t ecdsa-sk -O resident -O application=ssh:mykeyname
Generating public/private ecdsa-sk key pair.
You may need to touch your authenticator to authorize key generation.
Enter PIN for authenticator:
Enter file in which to save the key (/home/ubuntu/.ssh/id_ecdsa_sk): mytoken
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in mytoken
(...)
```

This will produce a public/private key pair as usual, but it will be possible to retrieve the private key part (the key handle) from the token later. This is done by running:

```
$ ssh-keygen -K
Enter PIN for authenticator:
You may need to touch your authenticator to authorize key download.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Saved ECDSA-SK key ssh:mytoken to id_ecdsa_sk_rk_mytoken
```

It will use the part after `ssh:` from the *application* parameter from before as part of the key filenames:

```
$ l id_ecdsa_sk_rk_mytoken*
-rw------- 1 ubuntu ubuntu 598 out  4 18:49 id_ecdsa_sk_rk_mytoken
-rw-r--r-- 1 ubuntu ubuntu 228 out  4 18:49 id_ecdsa_sk_rk_mytoken.pub
```

If you set a passphrase when extracting the keys from the hardware token, and later use these keys, you will be prompted for both the key passphrase, and the hardware key PIN, and you will also have to touch the token:

```
$ ssh -i ./id_ecdsa_sk_rk_mytoken ubuntu@focal.server
Enter passphrase for key './id_ecdsa_sk_rk_mytoken':
Confirm user presence for key ECDSA-SK
SHA256:t+l26IgTXeURY6e36wtrq7wVYJtDVZrO+iuobs1CvVQ
User presence confirmed
(...)
```

It is also possible to download and add resident keys directly to ssh-agent by running

```
$ ssh-add -K
```

In this case no file is written, and the public key can be printed by running `ssh-add -L`.

> NOTE
> If you used the `-O verify-required` option when generating the keys, or if that option is set on the SSH server via `/etc/ssh/sshd_config`'s `PubkeyAuthOptions verify-required`, then using the agent currently in Ubuntu 22.04 LTS won't work.

Two factor authentication with TOTP/HOTP

For the best two factor authentication (2FA) security, we recommend using hardware authentication devices that support U2F/FIDO. See the previous section for details. However, if this is not possible or practical to implement in your case, TOTP/HOTP based 2FA is an improvement over no two factor at all. Smartphone apps to support this type of 2FA are common, such as Google Authenticator.

Background

The configuration presented here makes public key authentication the first factor, the TOTP/HOTP code the second factor, and makes password authentication unavailable. Apart from the usual setup steps required for public key authentication, all configuration and setup takes place on the server. No changes are required at the client end; the 2FA prompt appears in place of the password prompt.

The two supported methods are HOTP and TOTP. Generally, TOTP is preferable if the 2FA device supports it.

HOTP <https://en.wikipedia.org/wiki/HMAC-based_one-time_password> is based on a sequence predictable only to those who share a secret. The user must take an action to cause the client to generate the next code in the sequence, and this response is sent to the server. The server also generates the next code, and if it matches the one supplied by the user, then the user has proven to the server that they share the secret. A downside of this approach is that if the user generates codes without the server following along, such as in the case of a typo, then the sequence generators can fall "out of sync". Servers compensate by allowing a gap in the sequence and considering a few subsequent codes to also be valid; if this mechanism is used, then the server "skips ahead" to sync back up. But to remain secure, this can only go so far before the server must refuse. When HOTP falls out of sync like this, it must be reset using some out of band method, such as authenticating using a second backup key in order to reset the secret for the first one.

TOTP <https://en.wikipedia.org/wiki/Time-based_one-time_password> avoids this downside of HOTP by using the current timezone independent date and time to determine the appropriate position in the sequence. However, this results in additional requirements and a different failure mode. Both devices must have the ability to tell the time, which is not practical for a USB 2FA token with no battery, for example. And both the server and client must agree on the correct time. If their clocks are skewed, then they will disagree on their current position in the sequence. Servers compensate for clock skew by allowing a few codes either side to also be valid. But like HOTP, they can only go so far before the server must refuse. One advantage of TOTP over HOTP is that correcting for this condition involves ensuring the clocks are correct at both ends; an out-of-band authentication to reset unfortunate users' secrets is not required. When using a modern smartphone app, for example, the requirement to keep the clock correct isn't usually a problem since this is typically done automatically at both ends by default.

> Note
>
> It is not recommended to configure U2F/FIDO at the same time as TOTP/HOTP. This combination has not been tested, and using the configuration presented here, TOTP/HOTP would become mandatory for everyone, whether or not they are also using U2F/FIDO.

Install software

From a terminal prompt, install the `google-authenticator` PAM module:

```
sudo apt update
sudo apt install libpam-google-authenticator
```

> Note
>
> The `libpam-google-authenticator` package is in Ubuntu's universe archive component, which receives best-effort community support only.

Configure users

Since public key authentication with TOTP/HOTP 2FA is about to be configured to be mandatory for users, each user who wishes to continue using ssh must first set up public key authentication and then configure their 2FA keys by running the user setup tool. If this isn't done first, users will not be able to do it later over ssh, since at that point they won't have public key authentication and/or 2FA configured to authenticate with.

Configure users' key-based authentication

To set up key-based authentication, see "SSH Keys" above. Once this is done, it can be tested independently of subsequent 2FA configuration. At this stage, user authentication should work with keys only, requiring the supply of the private key passphrase only if it was configured. If configured correctly, the user should not be prompted for their password.

Configure users' TOTP/HOTP 2FA secrets

Each user needs to run the setup tool to configure 2FA. This will ask some questions, generate a key, and display a QR code for the user to import the secret into their smartphone app, such as the Google Authenticator app on Android. The tool creates the file `~/.google-authenticator`, which contains a shared secret, emergency passcodes and per-user configuration.

As a user that needs 2FA configured, from a terminal prompt run the following command:

```
google-authenticator
```

Follow the prompts, scanning the QR code into your 2FA app as directed.

It's important to plan for the eventuality that the 2FA device gets lost or damaged. Will this lock the user out of their account? In mitigation, it's worth each user considering doing one or more of the following:

- Use the 2FA device's backup or cloud sync facility if it has one.
- Write down the backup codes printed by the setup tool.
- Take a photo of the QR code.
- (TOTP only) Scan the QR code on multiple 2FA devices. This only works for TOTP, since multiple HOTP 2FA devices will not be able to stay in sync.
- Ensure that the user has a different authentication path to be able to rerun the setup tool if required.

Of course, any of these backup steps also negate any benefit of 2FA should someone else get access to the backup, so the steps taken to protect any backup should be considered carefully.

Configure the ssh server

Once all users are configured, configure sshd itself by editing `/etc/ssh/sshd_config`. Depending on your installation, some of these settings may be configured already, but not necessarily with the values required for this configuration. Check for and adjust existing occurences of these configuration directives, or add new ones, as required:

```
KbdInteractiveAuthentication yes
PasswordAuthentication no
AuthenticationMethods publickey,keyboard-interactive
```

> Note
>
> On Ubuntu 20.04 "Focal Fossa" and earlier, use `ChallengeResponseAuthentication yes` instead of `KbdInteractiveAUthentication yes`.

Restart the `ssh` service to pick up configuration changes:

```
sudo systemctl try-reload-or-restart ssh
```

Edit `/etc/pam.d/sshd` and replace the line:

```
@include common-auth
```

with:

```
auth required pam_google_authenticator.so
```

Changes to PAM configuration have immediate effect, and no separate reloading command is required.

Log in using 2FA

Now when you log in using ssh, in addition to the normal public key authentication, you will be prompted for your TOTP or HOTP code:

```
$ ssh jammy.server
Enter passphrase for key 'id_rsa':
(ubuntu@jammy.server) Verification code:
Welcome to Ubuntu Jammy Jellyfish...
(...)
ubuntu@jammy.server:~$
```

Special cases

On Ubuntu, the following settings are default in `/etc/ssh/sshd_config`, but if you have overridden them, note that they are required for this configuration to work correctly and must be restored as follows:

```
UsePAM yes
PubkeyAuthentication yes
```

Remember to run `sudo systemctl try-reload-or-restart ssh` for any changes make to sshd configuration to take effect.

References

- Ubuntu Wiki SSH <https://help.ubuntu.com/community/SSH> page.
- OpenSSH Website <http://www.openssh.org/>
- OpenSSH 8.2 release notes <https://www.openssh.com/txt/release-8.2>
- Advanced OpenSSH Wiki Page <https://wiki.ubuntu.com/AdvancedOpenSSH>
- Yubikey documentation for OpenSSH FIDO/FIDO2 usage <https://developers.yubico.com/SSH/Securing_SSH_with_FIDO2.html>
- Wikipedia on TOTP <https://en.wikipedia.org/wiki/Time-based_one-time_password>
- Wikipedia on HOTP <https://en.wikipedia.org/wiki/HMAC-based_one-time_password>

Previous NFS Next OpenVPN

This page was last modified 10 months ago. Help improve this document in the forum <https://discourse.ubuntu.com/t/openssh-server/11896> .