# **How To Set Up Multi-Factor Authentication for SSH on Ubuntu 20.04**

### Introduction

SSH uses passwords for authentication by default, and [most SSH hardening instructions recommend using an SSH key instead](https://www.digitalocean.com/community/tutorials/how-to-harden-openssh-on-ubuntu-20-04). However, an SSH key is still only a single factor, though a much more secure factor. The channel is the terminal on your computer sending the data via an encrypted tunnel to the remote machine. But like a hacker can guess a password, they can steal an SSH key, and then in either case, with that single piece of data, an attacker can gain access to your remote systems.

In this tutorial, we’ll set up multi-factor authentication to combat that. **Multi-factor authentication** (MFA) or **Two-factor authentication** (2FA) requires more than one factor to authenticate or log in. This means a bad actor would have to compromise multiple things, like your computer and your phone, to get in. There are several types of factors used in authentication:

1. Something you **know**, like a password or security question
2. Something you **have**, like an authenticator app or security token
3. Something you **are**, like your fingerprint or voice

One common factor is an OATH-TOTP app, like Google Authenticator. OATH-TOTP (Open Authentication Time-Based One-Time Password) is an open protocol that generates a one-time use password, commonly a six-digit number recycled every 30 seconds.

This article will go over how to enable SSH authentication using an OATH-TOTP app in addition to an SSH key. Logging into your server via SSH will require two factors across two channels, thereby making it more secure than a password or SSH key alone. Also, we’ll go over some additional use cases for MFA and some helpful tips and tricks.

And if you are seeking further guidance on securing SSH connections, check out these [tutorials on Hardening OpenSSH](https://www.digitalocean.com/community/tutorials/how-to-harden-openssh-on-ubuntu-20-04) and [Hardening OpenSSH Client](https://www.digitalocean.com/community/tutorials/how-to-harden-openssh-client-on-ubuntu-20-04).

## Prerequisites

To follow this tutorial, you will need:

* [One Ubuntu 20.04 server](https://www.digitalocean.com/products/droplets/) with a sudo non-root user, SSH key, and firewall enabled, which you can set up by following this [Initial Server Setup tutorial](https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-20-04).
* A smartphone or tablet with an OATH-TOTP app installed, like Google Authenticator ([iOS](https://itunes.apple.com/us/app/google-authenticator/id388497605?mt=8), [Android](https://play.google.com/store/apps/details?id=com.google.android.apps.authenticator2&hl=en)).
* Alternatively, you can also use a Linux command line app called ‘oathtool’ to generate an OATH-TOTP code. It’s available in various distribution [repos](https://www.cyberciti.biz/faq/use-oathtool-linux-command-line-for-2-step-verification-2fa/)
* If you want to secure your SSH connection, there are several good steps outlined in this [SSH Essentials](https://www.digitalocean.com/community/tutorials/ssh-essentials-working-with-ssh-servers-clients-and-keys) article, such as whitelisting users, disabling root login, and changing which port SSH uses.

## Step 1 — Installing Google’s PAM

In this step, we’ll install and configure Google’s PAM.

PAM, which stands for *Pluggable Authentication Module*, is an authentication infrastructure used on Linux systems to authenticate a user. Because Google made an OATH-TOTP app, they also made a PAM that generates TOTPs and is fully compatible with any OATH-TOTP app, like Google Authenticator or [Authy](https://www.authy.com/).

First, update Ubuntu’s repository cache:

sudo apt-get update

Copy

Next, install the PAM:

sudo apt-get install libpam-google-authenticator

Copy

With the PAM installed, we’ll use a helper app that comes with the PAM to generate a TOTP key for the user that needs a second factor. This key is generated on a user-by-user basis, not system-wide. This means every user that wants to use a TOTP auth app will need to log in and run the helper app to get their key; you can’t just run it once to enable it for everyone (but there are some tips at the end of this tutorial to set up or require MFA for many users).

Run the initialization app:

google-authenticator

Copy

After you run the command, the app will ask a few questions. The first one asks if authentication tokens should be time-based:

Output

Do you want authentication tokens to be time-based (y/n) y

This PAM allows for time-based or sequential-based tokens. Using *sequential-based tokens* means the code starts at a certain point and then increments the code after every use. Using *time-based tokens* means the code changes after a certain time frame. We’ll stick with time-based because that is what apps like Google Authenticator anticipate, so answer y for yes.

After answering this question, a lot of output will scroll past, including a large QR code. Use your authenticator app on your phone to scan the QR code or manually type in the secret key. If the QR code is too big to scan, you can use the URL above the QR code to get a smaller version. Once it’s added, you’ll see a six-digit code that changes every 30 seconds in your app.

**Note**: Make sure you record the secret key, verification code, and the recovery codes in a safe place, like a password manager. The recovery codes are the only way to regain access if you, for example, lose access to your TOTP app.

The remaining questions inform the PAM on how to function. We’ll go through them one by one:

Output

Do you want me to update your "~/.google\_authenticator" file (y/n) y

This writes the key and options to the .google\_authenticator file. If you say no, the program quits and nothing is written, which means the authenticator won’t work:

Output

Do you want to disallow multiple uses of the same authentication  
token? This restricts you to one login about every 30s, but it increases  
your chances to notice or even prevent man-in-the-middle attacks (y/n) y

By answering yes here, you prevent a replay attack by making each code expire immediately after use. This prevents an attacker from capturing a code you just used and logging in with it:

Output

By default, a new token is generated every 30 seconds by the mobile app.  
In order to compensate for possible time-skew between the client and the server,  
we allow an extra token before and after the current time. This allows for a  
time skew of up to 30 seconds between the authentication server and client. Suppose you  
experience problems with poor time synchronization. In that case, you can increase the window  
from its default size of 3 permitted codes (one previous code, the current  
code, the next code) to 17 permitted codes (the eight previous codes, the current  
code, and the eight next codes). This will permit a time skew of up to 4 minutes  
between client and server.  
Do you want to do so? (y/n) n

Answering yes here allows up to 17 valid codes in a moving four-minute window. By answering no, you limit it to 3 valid codes in a 1:30 minute rolling window. Unless you find issues with the 1:30 minute window, answering no is the more secure choice. You can change this setting later in the .google\_authenticator file stored at the root of your home directory:

Output

If the computer that you are logging into isn't hardened against brute-force  
login attempts, you can enable rate-limiting for the authentication module.  
By default, this limits attackers to no more than three login attempts every 30s.  
Do you want to enable rate-limiting (y/n) y

Rate limiting means a remote attacker can only attempt a certain number of guesses before being forced to wait some time before being able to try again. If you haven’t previously configured rate limiting directly into SSH, doing so now is a great hardening technique.

**Note**: Once you finish this setup, if you want to back up your secret key, you can copy the ~/.google-authenticator file to a trusted location. From there, you can deploy it on additional systems or redeploy it after a backup.

Now that Google’s PAM is installed and configured, the next step is to configure SSH to use your TOTP key. We’ll need to tell SSH about the PAM and then configure SSH to use it.

## Step 2 — Configuring OpenSSH to Use MFA/2FA

Because we’ll be making SSH changes over SSH, it’s important never to close your initial SSH connection. Instead, open a second SSH session to do testing. This is to avoid locking yourself out of your server if there was a mistake in your SSH configuration. Once everything works, then you can safely close any sessions. Another safety precaution is to create a backup of the system files you will be editing, so if something goes wrong, you can\srevert to the original file and start over again with a clean configuration.

To begin, make a backup of the sshd configuration file:

sudo cp /etc/pam.d/sshd /etc/pam.d/sshd.bak

Copy

Now open the file using nano or your favorite text editor:

sudo nano /etc/pam.d/sshd

Copy

Add the following line to the bottom of the file:

/etc/pam.d/sshd

. . .  
# Standard Un\*x password updating.  
@include common-password  
auth required pam\_google\_authenticator.so nullok  
auth required pam\_permit.so

The nullok word at the end of the last line tells the PAM that this authentication method is optional. This allows users without an OATH-TOTP token to still log in just using their SSH key. Once all users have an OATH-TOTP token, you can remove nullok from this line to make MFA mandatory. The second line with pam\_permit.so is required to allow authentication if a user doesn’t use an MFA token to log in. When logging in, each method needs a SUCCESS to allow authentication. If a user doesn’t use the MFA auth tool, utilizing the nullok option returns an IGNORE for the interactive keyboard authentication. pam\_permit.so then returns SUCCESS and allows authentication to proceed.

Save and close the file.

Next, we’ll configure SSH to support this kind of authentication.

First make a backup of the file:

sudo cp /etc/ssh/sshd\_config /etc/ssh/sshd\_config.bak

Copy

Now open the SSH configuration file for editing:

sudo nano /etc/ssh/sshd\_config

Copy

Look for ChallengeResponseAuthentication and set its value to yes:

/etc/ssh/sshd\_config

. . .  
# Change to yes to enable challenge-response passwords (beware issues with  
# some PAM modules and threads)  
ChallengeResponseAuthentication yes  
. . .

Save and close the file, then restart SSH to reload the configuration files. Restarting the sshd service won’t close our current open connections, meaning you won’t risk locking yourself out with this command:

sudo systemctl restart sshd.service

Copy

To test that everything’s working so far, open ANOTHER terminal and try logging in over SSH. It is very important that you keep your current SSH session open and test with an additional session, or you will lock yourself out at some point and will need to use the web console to get yourself back in.

**Note:** If you’ve previously created an SSH key and are using it, you’ll notice you didn’t have to type in your user’s password or the MFA verification code. This is because an SSH key overrides all other authentication options by default. Otherwise, you should have gotten a password and verification code prompt.

Next, to enable an SSH key as one factor and the verification code as a second, we need to tell SSH which factors to use and prevent the SSH key from overriding all other types.

## Step 3 — Making SSH Aware of MFA

MFA is still not working if you are using and SSH key. To make SSH aware of MFA, reopen the sshd configuration file:

sudo nano /etc/ssh/sshd\_config

Copy

Add the following line at the bottom of the file. This tells SSH which authentication methods are required. We tell SSH that users need an SSH key and either a password or a verification code (or all three):

/etc/ssh/sshd\_config

. . .  
AuthenticationMethods publickey,password publickey,keyboard-interactive

Save and close the file.

Next, open the PAM sshd configuration file again:

sudo nano /etc/pam.d/sshd

Copy

Find the line @include common-auth and comment it out by adding a # character as the first character on the line. This tells PAM not to prompt for a password:

/etc/pam.d/sshd

. . .  
# Standard Un\*x authentication.  
#@include common-auth  
. . .

Save and close the file, then restart SSH:

sudo systemctl restart sshd.service

Copy

Now try logging into the server again with a different terminal session/window. Unlike last time, SSH should ask for your verification code. Enter it, and you will complete the login. Even though there is no indication that your SSH key was used, your login attempt used two factors. If you want to verify this, you can add -v (for verbose) after the SSH command.

The -v switch will produce an output like this:

Example SSH output\

. . .  
debug1: Authentications that can continue: publickey  
debug1: Next authentication method: publickey  
debug1: Offering RSA public key: /Users/sammy/.ssh/id\_rsa  
debug1: Server accepts key: pkalg rsa-sha2-512 blen 279  
Authenticated with partial success.  
debug1: Authentications that can continue: password,keyboard-interactive  
debug1: Next authentication method: keyboard-interactive  
Verification code:

Towards the end of the output, you’ll see where SSH uses your SSH key and then asks for the verification code. You can now log in over SSH with an SSH key and a one-time password. If you want to enforce all three authentication types, you can follow the next step.

Congratulations, you’ve successfully added a second factor when logging in remotely to your server over SSH. If this is what you wanted — to use your SSH key and a TOTP token to enable MFA for SSH (for most people, this is the optimal configuration) — then you’re done.

What follows are some tips and tricks for recovery, automated usage, and more.

## Step 4 — Adding a Third Factor (Optional)

In Step 3, we listed the approved types of authentication in the sshd\_config file:

1. publickey (SSH key)
2. password publickey (password)
3. keyboard-interactive (verification code)

Although we listed three different factors, the options we’ve chosen so far, they only allow for an SSH key and the verification code. If you’d like to have all three factors (SSH key, password, and verification code), one quick change will enable all three.

Open the PAM sshd configuration file:

sudo nano /etc/pam.d/sshd

Copy

Locate the line you commented out previously, #@include common-auth, and uncomment the line by removing the # character. Save and close the file. Now once again, restart SSH:

sudo systemctl restart sshd.service

Copy

By enabling the option @include common-auth, PAM will now prompt for a password in addition to checking for an SSH key and asking for a verification code, which we had working previously. Now we can use something we know (password) and two different types of things we have (SSH key and verification code) over two different channels (your computer for the SSH key and your phone for the TOTP token).

Step5- Configure sudo without password for a user

If you wish the “admin” user to execute all commands using sudo with no password edit the configuration file /etc/sudoers using command sudo visudo at the bottom of file add line admin ALL=(ALL) NOPASSWD:ALL .

## Step 6 — Recovering Access to Google MFA (optional)

As with any system that you harden and secure, you become responsible for managing that security. In this case, that means not losing your SSH key or your TOTP secret key and &making sure you have access to your TOTP app. However, sometimes things happen, and you can lose control of the keys or apps you need to get in.

### Losing Your TOTP Secret Key

If you lose your TOTP secret key, you can break the recovery process into a couple of steps. The first is getting back in without knowing the verification code, and the second is finding the secret key or regenerating it for normal MFA login. This often can happen if you get a new phone and don’t transfer over your secrets to a new authenticator app.

To get in after losing the TOTP secret key on a DigitalOcean Droplet, you can\s[use the virtual console](https://www.digitalocean.com/community/tutorials/how-to-use-the-digitalocean-console-to-access-your-droplet) from your dashboard to log in using your username and password. This works because we only protected your user account with MFA for SSH connections. Non-ssh connections, such as a console login, don’t use the Google Authenticator PAM module.

If you’re on a non-Droplet system, then you have two options for regaining access:

1. Console (local/non-ssh) access to the system (typically physical or via something like iDrac)
2. Have a different user that doesn’t have MFA enabled

The second option is the less secure option since the point in using MFA is to harden all SSH connections, but it’s one fail-safe if you lose access to your MFA authenticator app.

Once you’ve logged in, there are two ways to help get the TOTP secret:

1. Recover the existing key
2. Generate a new key

In each user’s home directory, the secret key and Google Authenticator settings are saved in a ~/.google-authenticator file. The very first line of this file is a secret key. A quick way to get the key is to execute the following command, which displays the first line of the google-authenticator file (i.e., the secret key). Then, take that secret key and manually type it into a TOTP app:

* head -n 1 /home/sammy/.google\_authenticator

Copy

Once you’ve recovered your existing key, you can either manually type it into your authenticator app or fill in the relevant details in the URL below and have Google generate a QR code for you to scan. You’ll need to add your username, hostname, the secret key from the .google-authenticator file, and then any name of your choosing for ‘entry-name-in-auth-app’ to easily identify this key versus a different TOTP token:

<https://www.google.com/chart?chs=200x200&chld=M|0&cht=qr&chl=otpauth://totp/username@hostname%3Fsecret%3D16-char-secret%26issuer%3Dentry-name-in-auth-app>

If there is a reason not to use the existing key (for example, being unable to easily share the secret key with the impacted user securely), you can remove the ~/.google-authenticator file outright. This will allow the user to log in again using only a single factor, assuming you haven’t enforced MFA by removing the nullok option. They can then run google-authenticator to generate a new key.

### Losing Access to the TOTP App

If you need to log in to your server but don’t have access to your TOTP app to get your verification code, you can still log in using the recovery codes that were displayed when you first created your secret key, and are the last five lines of the .google-authenticator file. Note that these recovery codes are one-time use. However, for this to work, you need to make your recovery codes available when you don’t have access to the TOTP app.

## Step 7 — Changing Authentication Settings (optional)

If you want to change your MFA settings after the initial configuration, instead of generating a new configuration with the updated settings, you can just edit the ~/.google-authenticator file. Options in this file appear in the following manner:

.google-authenticator layout

<secret key>  
<options>  
<recovery codes>

Options set in this file have a line in the options section; if you answered “no” to a particular option during the initial setup, the program excludes the corresponding.

Here are some changes you can make to this file:

* To enable sequential codes instead of time-based codes, change the line " TOTP\_AUTH to " HOTP\_COUNTER 1.
* To allow multiple uses of a single code, remove the line " DISALLOW\_REUSE.
* To extend the code expiration window to 4 minutes, add the line " WINDOW\_SIZE 17.
* To disable multiple failed logins (rate-limiting), remove the line " RATE\_LIMIT 3 30.
* To change the rate-limiting threshold, find the line " RATE\_LIMIT 3 30 and adjust the numbers. The 3 in the original indicates the number of attempts over a period, and the 30 indicates the time in seconds.
* To disable the use of recovery codes, remove the five eight-digit codes at the bottom of the file.

## Step 8 — Avoiding MFA for Some Accounts (optional)

There may be a situation in which a single user or a few service accounts (i.e., accounts used by applications, not humans) need SSH access without MFA enabled. For example, some applications that use SSH, like some FTP clients, may not support MFA. If an application doesn’t have a way to request the verification code, the request may get stuck until the SSH connection times out.

To control which factors to use for a user, you can edit the /etc/pam.d/sshd file.

To allow MFA for some accounts and SSH only for others, make sure the following settings in /etc/pam.d/sshd are active:

/etc/pam.d/sshd

# PAM configuration for the Secure Shell service  
  
# Standard Un\*x authentication.  
#@include common-auth  
  
. . .  
  
# Standard Un\*x password updating.  
@include common-password  
auth required pam\_google\_authenticator.so nullok

Here, @include common-auth is commented out because passwords need to be disabled. You cannot force MFA if some accounts have MFA disabled, so leave the nullok option on the final line.

After setting this configuration, run google-authenticator as any users that need MFA, and don’t run it for users that will only use SSH keys.

## Step 9 — Automating Setup with Configuration Management (optional)

Many system administrators use [configuration management tools](https://www.digitalocean.com/community/tutorial_series/getting-started-with-configuration-management), like Puppet, Chef, or Ansible, to manage their systems. You can use a system like this to install and set up a secret key whenever a new user creates an account.

google-authenticator supports command-line switches to set all the options in a single, non-interactive command. To see all the options, you can type google-authenticator --help. Below is the command that would set everything up as outlined in Step 1:

* google-authenticator -t -d -f -r 3 -R 30 -w 3

Copy

The options referenced above are as follows:

* -t => Time based counter
* -d => Disallow token reuse
* -f => Force writing the settings to file without prompting the user
* -r => How many attempts to enter the correct code
* -R => How long in seconds a user can attempt to enter the correct code
* -w => How many codes can are valid at a time (this references the 1:30 min - 4 min window of valid codes)

This fully configures the authenticator, saves it to a file, and then outputs the secret key, QR code, and recovery codes. (If you add the flag -q, then there won’t be any output.) If you use this command in an automated fashion, make sure your script captures the secret key and/or recovery codes and makes them available to the user.

## Step 10 — Forcing MFA for All Users (optional)

If you want to force MFA for all users, even on the first login, or prefer not to rely on your users to generate their keys, there’s a quick way to handle this. You can use the same .google-authenticator file for each user, as there’s no user-specific data stored in the file.

To do this, after creating the configuration file, a privileged user needs to copy the file to the root of every home directory and change its permissions to the appropriate user. You can also copy the file to /etc/skel/, automatically copying the file to every new user’s home directory upon creation.

**Warning:** This can be a security risk because everyone is sharing the same second factor. This means that if it’s leaked, it’s as if every user had only one factor. Take this into consideration if you want to use this approach.

Another method to force the creation of a user’s secret key is to use a bash script that:

1. Creates a TOTP token,
2. Prompts them to download the Google Authenticator app and scan the QR code that will be displayed, and
3. Runs the google-authenticator application for them after checking if the .google-authenticator file already exists.

To make sure the script runs when a user logs in, you can name it .bash\_login and place it at the root of their home directory.

## Conclusion

In this tutorial, you added two factors (an SSH key + MFA token) across two channels (your computer + your phone) to your server. You’ve made it very difficult for an outside agent to brute force their way into your machine via SSH and greatly increased the security of your machine.

And remember, if you are seeking further guidance on securing SSH connections, check out these [tutorials on Hardening OpenSSH](https://www.digitalocean.com/community/tutorials/how-to-harden-openssh-on-ubuntu-20-04) and [Hardening OpenSSH Client](https://www.digitalocean.com/community/tutorials/how-to-harden-openssh-client-on-ubuntu-20-04).