How To Set Up Multi-Factor Authentication for SSH on CentOS 7

Published on March 9, 2017

Introduction

An *authentication factor* is a single piece of information used to to prove you have the rights to perform an action, like logging into a system. An *authentication channel* is the way an authentication system delivers a factor to the user or requires the user to reply. Passwords and security tokens are examples of authentication factors; computers and phones are examples of channels.

SSH uses passwords for authentication by default, and most SSH hardening instructions recommend using an SSH key instead. However, this is still only a single factor. If a bad actor has compromised your computer, then they can use your key to compromise your servers as well.

In this tutorial, we'll set up multi-factor authentication to combat that. *Multi-factor authentication* (MFA) requires more than one factor in order to authenticate, or log in. This means a bad actor would have to compromise multiple things, like both your computer and your phone, to get in. The different type of factors are often summarized as:

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 6 digit number that is 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 then require two factors across two channels, thereby making it more secure than a password or SSH key alone. In addition, we'll go over some additional use cases for MFA and some helpful tips and tricks.

Prerequisites

To follow this tutorial, you will need:

- One CentOS 7 server with a sudo non-root user and SSH key, which you can set up by following this Initial Server Setup tutorial.
- A smartphone or tablet with an OATH-TOTP app installed, like Google Authenticator (iOS, Android).

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.

First, we need to add the EPEL (Extra Packages for Enterprise Linux) repo.

```
$ sudo yum install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
```

Copy

Next, install the PAM. You may be prompted to accept the EPEL key if this is the first time using the repo. Once accepted you won't be prompted for again to accept the key.

```
$ sudo yum install 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 you want to add a second factor to. 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 own 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, you'll be asked 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* mean the code starts at a certain point and then increments the code after every use. Using *time-based tokens* mean the code changes randomly after a certain time elapses. 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. At this point, 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 how to function. We'll go through them one by one.

```
Output
Do you want me to update your "/home/sammy/.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 are preventing 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, tokens are good for 30 seconds. 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. If you experience problems with
poor time synchronization, you can increase the window from its default
size of +-1min (window size of 3) to about +-4min (window size of 17 acceptable tokens).
Do you want to do so? (y/n) n
```

Answering yes here allows up to 8 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.

```
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 3 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 blocked. 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

Because we'll be making SSH changes over SSH, it's important to never 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.

To begin we'll edit the `sshd` configuration file. Here, we're using `nano`, which isn't installed on CentOS by default. You can install it with `sudo yum install nano`, or use your favorite alternative text editor.

```
$ sudo nano /etc/pam.d/sshd
```

Copy

Add the following line to the bottom of the file.

/etc/pam.d/sshd

```
. . .
# Used with polkit to reauthorize users in remote sessions
-session   optional     pam_reauthorize.so prepare
auth required pam_google_authenticator.so nullok
```

The `nullok` word at the end of the last line tells the PAM that this authentication method is optional. This allows users without a OATH-TOTP token to still log in using their SSH key. Once all users have an OATH-TOTP token, you can remove `nullok` from this line to make MFA mandatory.

Save and close the file.

Next, we'll configure SSH to support this kind of authentication. Open the SSH configuration file for editing.

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

Copy

Look for `ChallengeResponseAuthentication` lines. Comment out the `no` line and uncomment the `no` line.

/etc/ssh/sshd_config

```
. . .
# Change to no to disable s/key passwords
ChallengeResponseAuthentication yes
#ChallengeResponseAuthentication no
. . .
```

Save and close the file, then restart SSH to reload the configuration files. Restarting the `sshd` service won't close open connections, so 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. 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.

If you want to make sure the what you've done so far works, in your open SSH session navigate to `~/.ssh/` and rename the authorized_keys file, temporarily, and open a new session and log in with our password and verification code.

```
$ cd ~/.ssh
$ mv authorized_keys authorized_keys.bak
```

Copy

Once you've verified that the your TOTP token works rename the 'authorized_keys.bak' file back to what it was.

```
$ mv authorized_keys.bak authorized_keys
```

Copy

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

Step 3 – Making 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. This line tells SSH we need a SSH key and either a password or a verification code (or all three).

/etc/ssh/sshd_config

```
. . .
# Added by DigitalOcean build process
ClientAliveInterval 120
ClientAliveCountMax 2
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 `auth substack password-auth` towards the top of the file. 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

```
. . .
#auth       substack     password-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 session. Unlike last time, SSH should ask for your verification code. Upon entering it, you'll be logged in. Even though you don't see any indication that your SSH key was used, your login attempt used two factors. If you want to verify, you can add `-v` (for verbose) after the SSH command:

```
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 ssh-rsa blen 279
Authenticated with partial success.
debug1: Authentications that can continue: 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 a SSH key and a one-time password. If you want to enforce all three authentication types, you can follow the next step.

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, with 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, `#auth substack password-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 `auth substack password-auth`, PAM will now prompt for a password in addition the 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.

So far, this article has outlined how to enable MFA with an SSH key and a time-based one-time password. If this is all you need, you can end here. However, this isn't the only way to do multi-factor authentication. Below are a couple additional ways of using this PAM module for multi-factor authentication and some tips and tricks for recovery, automated usage, and more.

Tip 1 – Recovering Access

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 an SSH Key or TOTP Secret Key

If you lose your SSH key or TOTP secret key, recovery can be broken up 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.

To get in after losing the secret key that generates the verification code on a DigitalOcean Droplet, you can simply use the virtual console from your dashboard to log in using your username and password.

Otherwise, you'll need an administrative user that has sudo access; make sure not to enable MFA for this user, but use just an SSH key. If you or another user loses their secret key and can't log in, then the administrative user can log in and help recover or regenerate the key for any user using `sudo`.

Once you're 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 `~/.google-authenticator`. 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

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 or the existing key was compromised), 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 'nullok' in the '/etc/pam.d/sshd' file. 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. Note that these recovery codes are one-time use. Once one is used to log in, it cannot be used as a verification code again.

Tip 2 – Changing Authentication Settings

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. This file is laid out in the following manner:

.google-authenticator layout

```
<secret key>
<options>
<recovery codes>
```

Options that are set in this file have a line in the options section; if you answered "no" to a particular option during the initial setup, the corresponding line is excluded from the file.

Here are the 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 threshold of rate limiting, find the line `" RATE_LIMIT 3 30` and adjust the numbers. The `3` in the original indicates the number of attempts over a period of time, and the `30` indicates the period of time in seconds.
- To disable the use of recovery codes, remove the five 8 digit codes at bottom of the file.

Tip 3 – Avoiding MFA for Some Accounts

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.

As long as a couple of options in `/etc/pam.d/sshd` are set correctly, you can control which factors are used on a user-by-user basis.

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

/etc/pam.d/sshd

```
#%PAM-1.0
auth       required     pam_sepermit.so
#auth       substack     password-auth
```

```
. . .

# Used with polkit to reauthorize users in remote sessions
-session   optional     pam_reauthorize.so prepare
auth       required      pam_google_authenticator.so nullok
```

Here, `auth substack password-auth` is commented out because passwords need to be disabled. MFA cannot be forced if some accounts are meant to have MFA disabled, so leave the `nullok` option on the final line.

After setting this configuration, simply run `google-authenticator` as any users that need MFA, and don't run it for users where only SSH keys will be used.

Tip 4 – Automating Setup with Configuration Management

Many system administrators use configuration management tools, like Puppet, Chef, or Ansible, to manage their systems. If you want to use a system like this to install set up a secret key when a new user's account is created, there is a method to do that.

`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
```

Copy

This answers all the questions we answered manually, 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 do use this command in an automated fashion, make sure to capture the secret key and/or recovery codes and make them available to the user.

Tip 5 – Forcing MFA for All Users

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

To do this, after the configuration file is initially created, 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/` so it's automatically copied over to a 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

That said, by having two factors (an SSH key + MFA token) across two channels (your computer + your phone), 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.

Comments

milinddhoke • July 10, 2019

I think line should be uncommented for `ChallengeResponseAuthentication yes`

```
Look for ChallengeResponseAuthentication lines. Comment out the no line and uncomment the no line.
```

Reply

jeremyhagan • January 8, 2019

Thanks for this writeup. I've set this up on AWS, intending to use MFA for named accounts and use the ec2-user account as a failsafe without MFA, however as soon as I add the line "AuthenticationMethods publickey,keyboard-interactive" to sshd_config the ec2-user account can no longer authenticate with "Failed (keyboard-interactive).

This would imply that my "nullok" setting from the pam.d config is absent, but it is not. Any ideas? Connecting with ssh -vvv shows ... debug2: we sent a publickey packet, wait for reply debug3: receive packet: type 51 Authenticated with partial success. debug1: Authentications that can continue: keyboard-interactive debug3: start over, passed a different list keyboard-interactive debug3: preferred publickey,keyboard-interactive,password debug3: authmethod_lookup keyboard-interactive debug3: remaining preferred: password debug3: authmethod_is_enabled keyboard-interactive debug1: Next authentication method: keyboard-interactive debug2: userauth_kbdint debug3: send packet: type 50 debug2: we sent a keyboard-interactive packet, wait for

reply debug3: receive packet: type 51 debug1: Authentications that can continue: keyboard-interactive debug3: userauth_kbdint: disable: no info_req_seen debug2: we did

not send a packet, disable method debug1: No more authentication methods to try. ec2-user@13.55.6.7: Permission denied (keyboard-interactive).

Show replies Reply

Mohakem Khan • February 26, 2018

Thanks for this helpful guide.

I had to make following slight change on the portion for `/etc/ssh/sshd_config`

```
AuthenticationMethods password publickey keyboard-interactive
```

Otherwise kept getting `"Permission denied (publickey)` error while trying to start the session.

`ssh -v` is helpful while trying to setup.

Reply

amitbhatt • February 12, 2018

This setup is working on my RHEL7 box. But I want to allow only root user and any other one user to be able to generate keys for other users and then share the keys with other users so that they can access the server. Can this be done?

Reply

PrestoNL79 • January 12, 2018

Thanks, this was very useful to me! I've now successfully setup MFA.

However, there is something that I would like to change, right now the authentication process checks for SSH key first, Verification code second and user password as last.

Can I change the order, so that it asks for the password first and verification code last?

Show replies Reply

Nearly Normal • June 9, 2017

I set it up this way. It works, but Google Authenticator is annoying because on my two phones and computer it doesn't automatically sync the authentication accounts. Authy does. So I'd like to UNinstall Google Authenticator and enable this with Authy instead. What should I do to uninstall the Google thing? Thank you.

Show replies Reply