Hashicorp Vault: Create Multi Factor Authentication for your Application

And test it with less than 50 line of Python code

[](https://medium.com/@tjabraha?source=post_page-----6994df363e3----------------------)

[Tjabraha](https://medium.com/@tjabraha?source=post_page-----6994df363e3----------------------)

Follow

Draft · 14 min read

**Basic MFA concepts**

So first things first. I am not a security engineer nor a software engineer. I am a humble DevOps dude that has used Hashicorp’s Vault product for a variety of different DevOps Deployment pipelines. This guide is intended to for folks who are interested in discovering some very basic concepts related to MFA(specifically Time Based One Time Passwords) and using security as a service products like Hashicorp Vault.

Why use MFA? Well there are a myriad of reasons, but the most common explanation is that relying on just usernames and passwords for authentication to your application are inherently risky. There are many different articles here on Medium and other blog sites that outline the reasons why you should use some form of MFA. You can peruse these at your leisure:

[*What is 2-Factor Authentication and Why Should You Care?*](https://medium.com/hackernoon/what-is-2-factor-authentication-and-why-you-should-care-e8af5808d499)

[*Multi-Factor Authentication (MFA)*](https://medium.com/bugbountywriteup/multi-factor-authentication-mfa-add5009eeb04)

[*How does Multi-Factor Authentication work?*](https://www.onelogin.com/learn/what-is-mfa)

The point to all this is that MFA is a good thing and you should use it as part of your user authentication process. While its not bulletproof, it has become a standard at most companies, and can save you a lot of heartache when dealing with malicious entities and attackers.

**So how how should I implement MFA on my app?**

You could just code your own process using [TOTP](https://en.wikipedia.org/wiki/Time-based_One-time_Password_algorithm). Whats TOTP? Well in a nutshell it uses an equation that creates a one time password using a combination of a cryptographic hash based on the current time from your server’s system clock. Another device uses this hash as well as the local time from its system clock and the same cryptographic hash you created on your server. Since time is a relatively universal attribute, and both devices are using the same hash key, the one time password on the second device should match the one time password created on your server. And since its based on the time and a hash key, if either one of those are different on either the server or the second device, then the passwords wont match.

Here is the equation for how to do that:

https://miro.medium.com/max/60/1*BarjSbRqeX1GI2zAv53VMw.jpeg?q=20

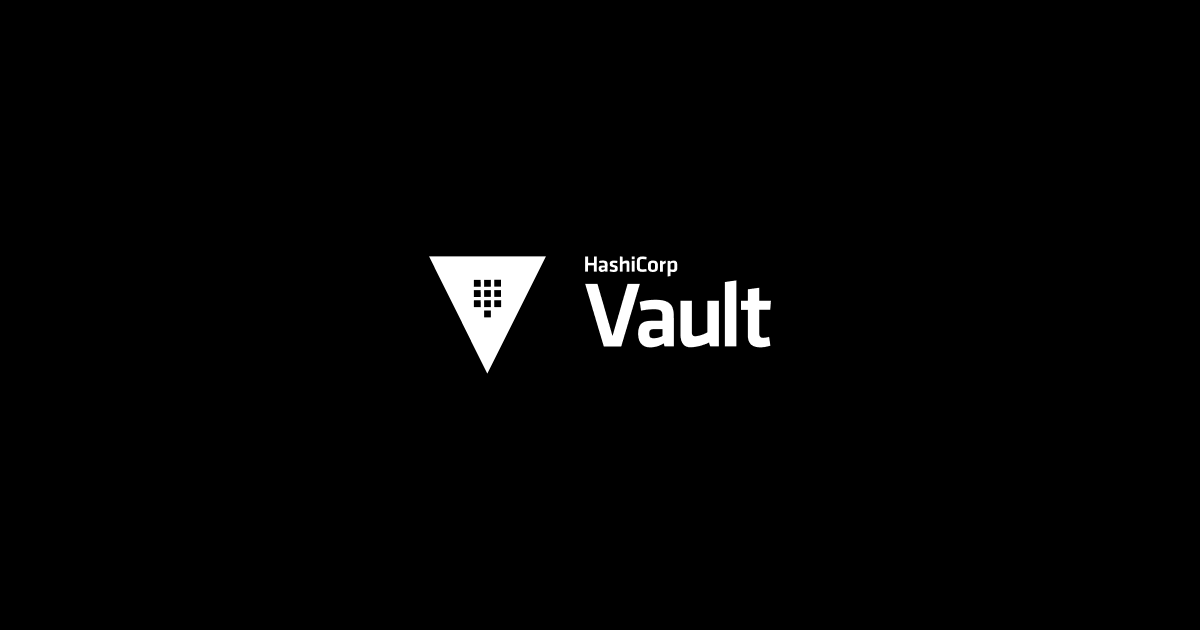
https://miro.medium.com/max/191/1*BarjSbRqeX1GI2zAv53VMw.jpeg

Sourced from wikipedia article

If you’re like me, and hate math , you probably would hate your life if you had to figure that out. And if you’re concerned with coding this and maintaining the tech debt on your app, you probably hope there is an easier way to do it.

This is where Hashicorp Vault comes in to the picture.

https://miro.medium.com/max/60/1*oImZxVJi_YOkf99JH6uxoA.png?q=20



Vault Logo

**What is Hashicorp Vault?**

Vault is security as a service product. It is open source and free to use.

The Hashicorp company describes it as a secrets management product for keeping passwords, encryption keys, and other secrets centrally located, utilizing tight controls to access those items. It does do that and it does it really well. At first glance, you might think its just another password manager, but its way more than that. Vault is api driven, and access to each of those endpoints can be managed and controlled. This allows users, applications, CI/CD pipelines or devices to access these secrets based on their level of access. Since it is centralized and access based, sensitive secrets can be versioned and changed at any time without having to locate and change these secrets across servers, application, or source code repos.

However Vault offers many other features as well. These other services don’t get as much attention. And that’s a shame. Because similar to its secret management service, Vault’s other security services offer a wide range of benefits. It can handle hashing algorithms, SSL certificate management, data randomization, and create encryption keys. And since it is api driven each of these services and access to them can be tightly controlled.

We will be using Vault’s TOTP service for the remainder of this tutorial.

**Vault MFA Tutorial**

Looks you’re still with me, so lets get started on the tutorial. For this tutorial you will need the following:

1. A computer (preferably one based on a Linux like OS. I am using a Mac)
2. A mobile phone
3. The Vault Application Binary
4. The Authy mobile phone app
5. Python 3 and the [requests library](https://requests.readthedocs.io/en/master/)

**Step 1 — Install Vault, Enable a Secrets Engine, and Create a Secret**

First lets download Vault. Go to the Vault download page and download the Vault binary for your operating system : [https://www.vaultproject.io/downloads/](https://www.vaultproject.io/downloads).

Once its downloaded, locate the Vault binary and add it to your Operating systems path. On linux like operating systems you may need to use sudo:

sudo cp vault /usr/local/bin

Once you’ve copied it to your path, check to see that its working properly:

vault --version  
Vault v1.4.0

It doesn’t have to be the exact version listed above, we are just checking to make sure the Vault binary is working

Next lets start up Vault in ‘dev mode’. From the command line enter the following:

vault server -dev

You should see output similar to this:

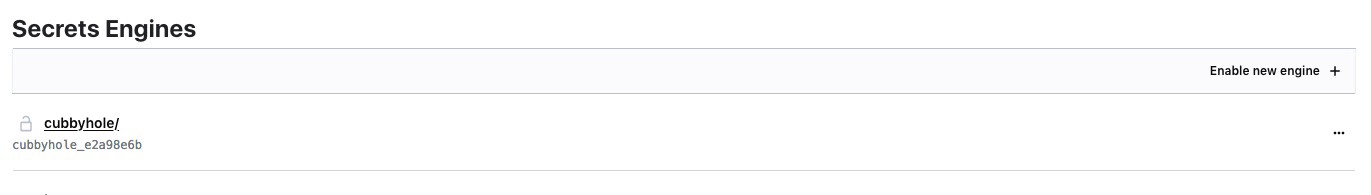
==> Vault server configuration:  
  
 Api Address: http://127.0.0.1:8200  
 Cgo: disabled  
 Cluster Address: https://127.0.0.1:8201  
 Listener 1: tcp (addr: "127.0.0.1:8200", cluster address: "127.0.0.1:8201", max\_request\_duration: "1m30s", max\_request\_size: "33554432", tls: "disabled")  
 Log Level: info  
 Mlock: supported: false, enabled: false  
 Recovery Mode: false  
 Storage: inmem  
 Version: Vault v1.4.0  
  
WARNING! dev mode is enabled! In this mode, Vault runs entirely in-memory  
and starts unsealed with a single unseal key. The root token is already  
authenticated to the CLI, so you can immediately begin using Vault.  
  
You may need to set the following environment variable:  
  
 $ export VAULT\_ADDR='http://127.0.0.1:8200'  
  
The unseal key and root token are displayed below in case you want to  
seal/unseal the Vault or re-authenticate.  
  
Unseal Key: Xqqc6UYEKP2QwfXesAPNSR6NogvynHW4I/3mzG7qkgw=  
***Root Token: s.7osR7zASDqNNfhvFxxZPbSZz***  
 **Development mode should NOT be used in production installations!**  
  
==> Vault server started! Log data will stream in below:  
  
2020-04-13T13:48:12.338-0500 [INFO] proxy environment: http\_proxy= https\_proxy= no\_proxy=

**Make sure you copy the Root Token from the output and keep it somewhere handy for the remainder of the exercise.** (Your root token will be different than whats listed above)

*Starting Vault in dev mode starts Vault completely in memory so its important not to stop the process until after you complete this exercise, otherwise you will have to start over with your work.*

To test that your Vault instance is up and running you can go to the UI and login with the root token by accessing [http://127.0.0.1:8200](http://127.0.0.1:8200/) using your browser. Enter in the root token to login.

https://miro.medium.com/max/60/1*NDqYvq0fTLlpNBZO83Op5A.jpeg?q=20



You should see this after logging in

Ok, Vault is up and running! Now lets start by entering in some passwords for your app.

Click Enable new Engine and select the KV (it stands for Key-Value by the way) option on the next page:

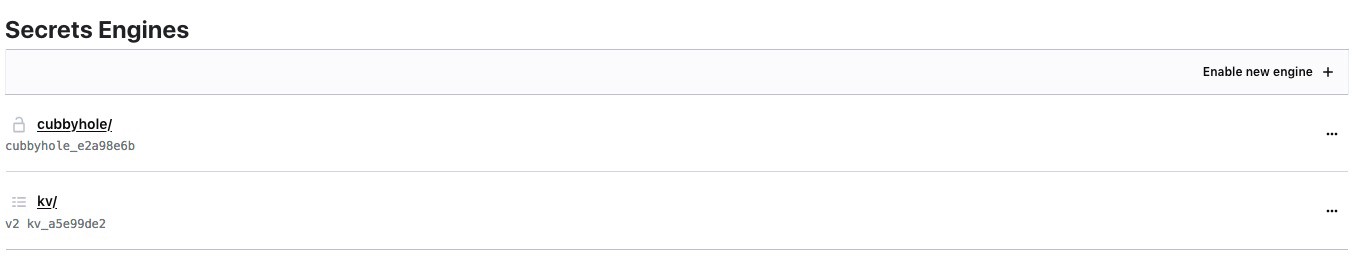
https://miro.medium.com/max/44/1*86CWghe787zvB0DRf0n26w.jpeg?q=20



Select KV and hit Next

Accept the defaults on the next page and click Enable Engine. You should now see the KV engine in your secrets engine list:

https://miro.medium.com/max/60/1*_l7w36MicmG4oEPO4_lEhA.jpeg?q=20



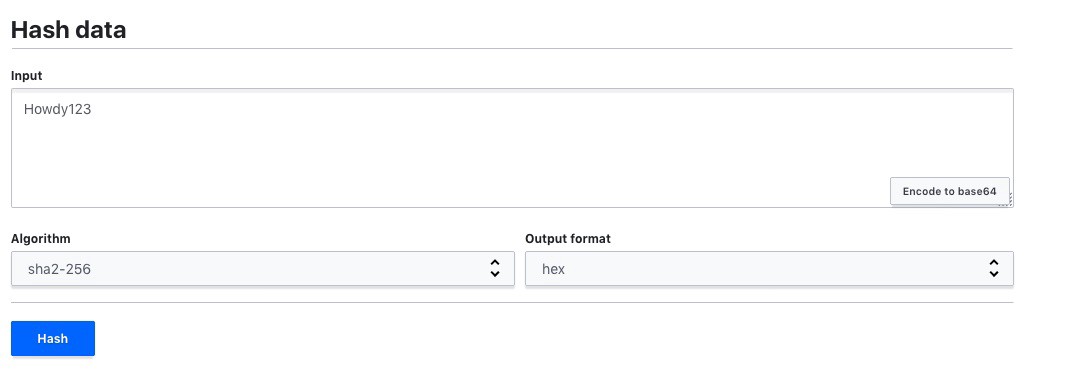
kv/ engine has now been enabled

Now lets create a hashed password. We can do this from the Vault UI by using the hashing tool.

On the top menu bar click Tools, then click Hash. Enter in a terrible user password like **Howdy123** into the text box. (While its recommended to base64 encode your passwords before hashing, don’t worry about that for now). Keep the hash algorithm sha2–256 and change the output to **hex**

Make sure your screen looks **exactly** like this:

https://miro.medium.com/max/60/1*x1wKYIIPOttLxYnecxFkWg.jpeg?q=20



Click Hash when done

It will produce the hash on the next screen:

0ee78709ab2a4a9b404b7676ff0abe3bbb3b7f44f2940b4e88d7cfa694fa9fa9

Copy the hash somewhere handy since that will be the app password for our user.

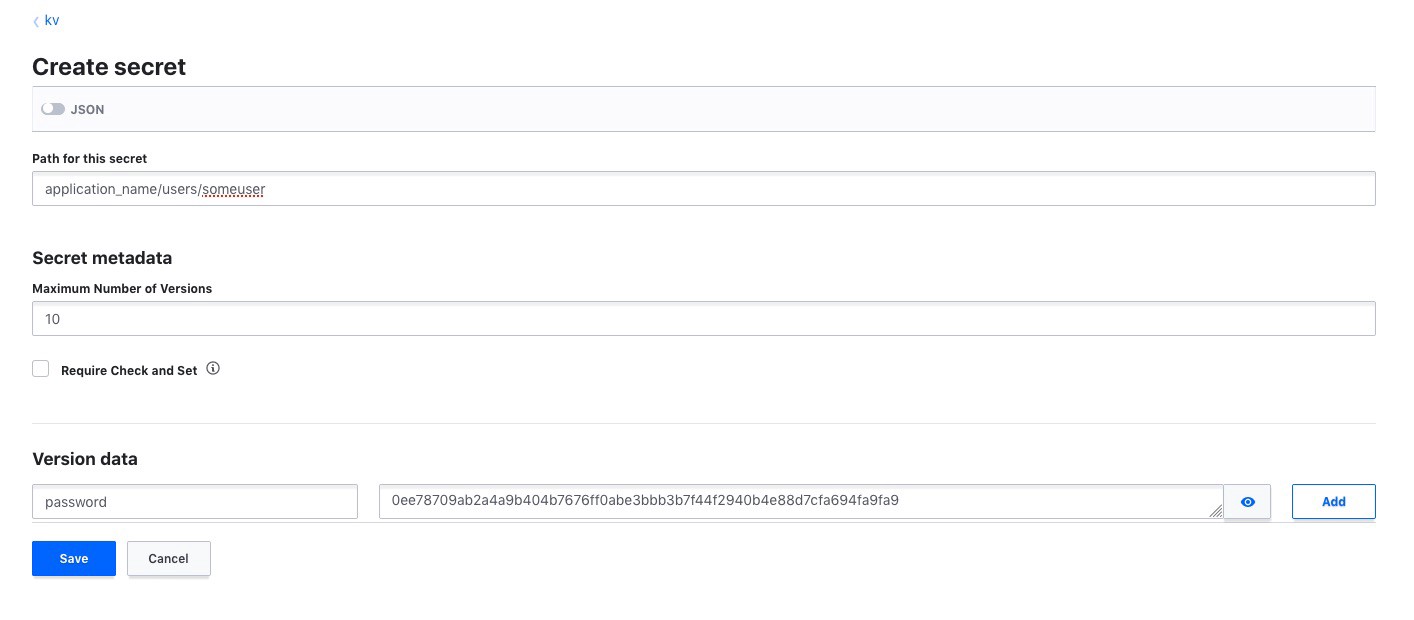
Now go back to the kv engine by selecting Secrets from the top menu and then k/v. Click Create Secret.

In the path field enter the path exactly as listed here:

application\_name/users/someuser

In the version data field enter a *key* named Password and set the *value*to the password hash we just generated. The screen should look like this:

https://miro.medium.com/max/60/1*t3jyazbrsXMbijALCuFsWQ.jpeg?q=20



Click Save

**Take-away Concepts for this Section**

Now that we have created an application password, its a good idea to review what we did here and why. We have simulated a fake user password for an application. The path we entered up above (application\_name/users/someuser) is an api endpoint that we created. I chose generic terms like *application\_name* but that would preferably be the name of your specific app name. We used the*/users/* path to create the endpoint to give us an indication that we are entering user data. The final part of the path is *someuser*. This is the unique user name for the user. For this user we entered in a *key* called password and put in a *value* equal to the hex hash of the password Howdy123.

So for your app the path could be similar to this f*inancial\_app/users/joeuser1*. For another user for the same app you would create another path called f*inancial\_app/users/janeuser2.*Since Vault is completely based on the path to a specific api endpoint, these paths should be thoroughly thought out and planned. This specific path strategy may not work for your organization, but for this example its the simplest way to go.

The other thing to note is that most usernames and passwords for apps are kept in databases and are hashed and encrypted. While putting all application usernames and passwords in Vault is definitely more secure, there are various pros and cons about keeping passwords in databases vs Vault. Since we aren’t going to spend time on setting up a database for this tutorial, we are just using Vault.

The final point I want to make is that Vault has many security tool endpoints and the hash function is one of them. When we get to the code section of this tutorial you will see this in action. Instead of having to hash passwords with your code every time a new user is created, you can simply call the hash endpoint, hash your data, and then place it wherever you like. This makes it much easier to maintain from a code perspective since all you are doing is calling a service. So when your security engineer tells you need to go to sha2–512 hash, you don’t need to hunt through your code to change your hash algorithm and test it. It’s literally a one word change when your using Vault.

**Step 2 — Enable TOTP Engine, Create a barcode for a user, Capture with Authy**

Next lets enable the TOTP engine. From the Vault UI select Secrets/Secret Engine and click Enable new Engine. On the next screen, select TOTP:

https://miro.medium.com/max/50/1*xkGaOnWRT4QRb401cNGt0A.jpeg?q=20



Click Next

Accept the defaults on the next page and Enable the Engine.

Now we are going to create a unique TOTP barcode for our user.

To do this you must use the command line. Open a new terminal prompt and enter in the following command:

export VAULT\_ADDR='http://127.0.0.1:8200'

Then to test that you can connect to Vault from the command line, enter in the following command:

vault status

The output should look like this:

Key Value  
--- -----  
Seal Type shamir  
Initialized true  
Sealed false  
Total Shares 1  
Threshold 1  
Version 1.4.0  
Cluster Name vault-cluster-351952cf  
Cluster ID 9c251179-ee0f-9bdf-7010-f0152ba29522  
HA Enabled false

Now that we have verified that Vault is working from the command line Copy and paste the following command exactly as it appears below (without the comments):

vault write --field=barcode totp/keys/someuser generate=true issuer=vault account\_name=someuser@test.com | base64 -D > someuser@test.png# Please note, use the base64 decode option for your os. I am using # a mac so it requires a -D. For many Linux OS' the command is   
# base64 -d

This command generates the barcode that is needed for the authenticator app. There’s a lot of things going on here but a simple explanation is that we are creating a unique hashed value based on the user we created in the last section (*someuser*). Vault automatically generates the barcode data in base64 so we need to do it using the base64 -D command. After that we are just piping the output to a file called *someuser@test.png.*

Locate the *someuser@test.png*and open it with your favorite image editor. It should look something like this:

https://miro.medium.com/max/60/1*2vwAwbN9E2OjCJaqB-mtOg.png?q=20

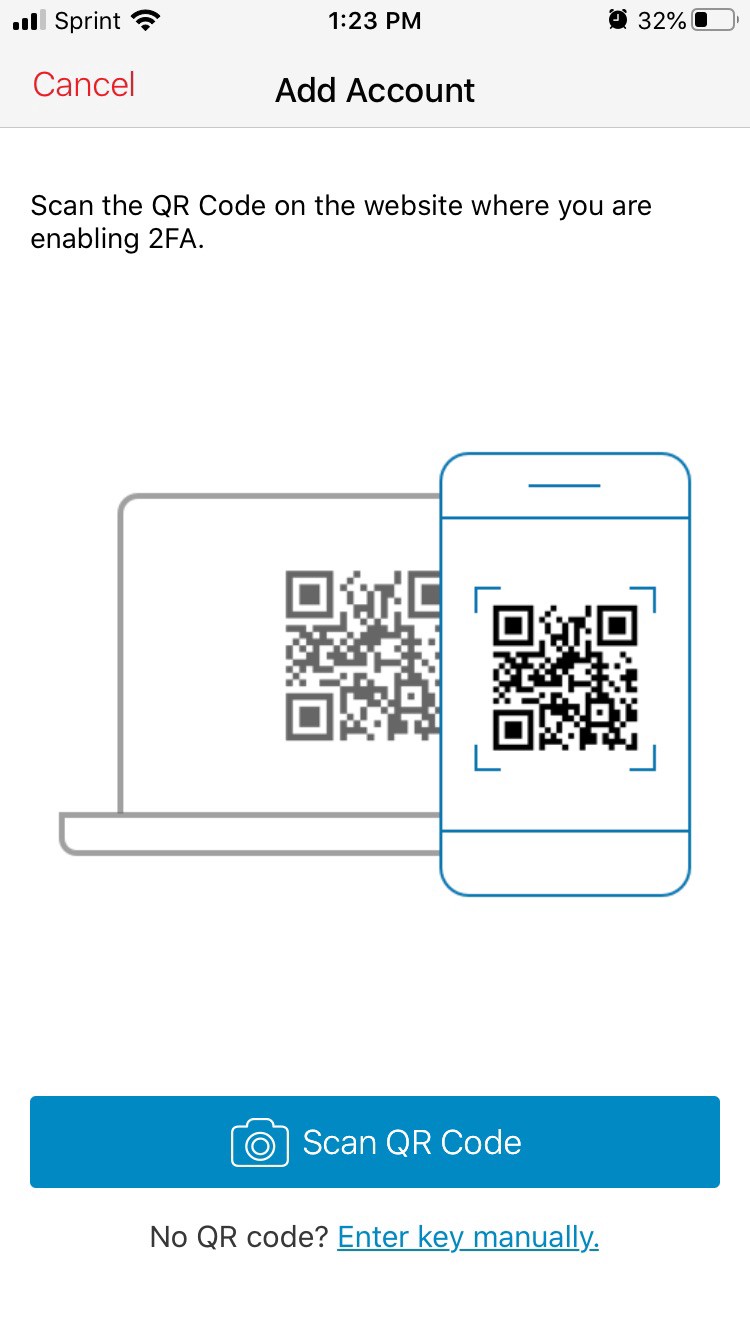


Don’t use this one, it won’t work since the hash key will not match the one create by your Vault Instance

Now we need to read it using out Authenticator app, Authy. (I am fairly confident this will work with most authentication apps like Google Authenticator, but I have not tested it lately). If you don’t have Authy yet, you can install it from your phones app store.

Open Authy on your phone and complete any of the setup prompts. When everything is complete, click the Add Account button on the screen. It will prompt you scan the QR code:

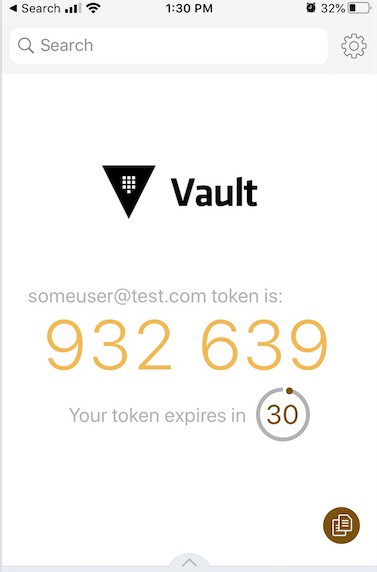
https://miro.medium.com/max/34/1*CSZ-GyYRW5YrZCAbd038CA.jpeg?q=20



You may be prompted to give Authy permission to use your phone’s camera. Hit yes if asked

Point your camera at you desktop screen at the barcode file in your image editor. Authy should immediately recognize it and will create a screen that looks similar to this:

https://miro.medium.com/max/40/1*Wmzo_3tCOITADtBjGgnrYg.jpeg?q=20



Your 6 digit code will be different

After 30 seconds the code will change. Now lets test if the codes between Vault and Authy match. Once your code switches to the next 30 second interval, go to your computer’s command line enter this command:

vault read totp/code/someuser

The output should look something like this:

Key Value  
--- -----  
code 595459# your six digit code will be different

The code that is produced should match exactly what is on your phone’s screen. To test this again you can wait till the 30 seconds interval switches on your phone again and then rerun the same command. This is an indication that both Vault and Authy are syncing to the current time using their respective system clocks, and are producing the same codes.

Vault also gives you convenient command to check if the code from your phone matches the one in Vault. To test it, enter in the following command:

vault write totp/code/someuser code=<the\_code\_from\_your\_phone>

The output will look like this:

Key Value  
--- -----  
valid true

We have successfully setup the TOTP tokens.

**Take-away Concepts for this Section**

We did some important steps in this section. The first thing we did is setup an endpoint for the app userfor our TOTP engine. For simplicity’s sake, we just named the endpoint *totp/code/someuser*which is equal to the login name of the user. Vault generated the unique hash key for the user and provided the barcode for the authenticator. All this was done with one command and without any complicated math.

We did this by using the Vault Command Line Tool, but **ideally all of this would be done via code when a new user is setup for your app**. Note that we used the vault write command which translates to a standard http POST or PUT command. Since Vault is developed using API driven code, every action can be done using standard http methodologies. You will see this when you get to the code example but as an example, we can run the last command that verifies the code (*vault write totp/code/someuser code=<the\_code\_from\_your\_phone>*) using a simple curl command:

curl -X PUT -H "X-Vault-Token: <your root token for vault>" -H "X-Vault-Request: true" -d '{"code":"<the code from your phone>"}' http://127.0.0.1:8200/v1/totp/code/someuser

The output generated from this looks like this:

{"request\_id":"140696e4-b628-b66a-ff74-3c940e536b10","lease\_id":"","renewable":false,"lease\_duration":0,"data":{"valid":true},"wrap\_info":null,"warnings":null,"auth":null}

(When using http methods the output will always generate json unless you specify some other output like yaml).

**Step 3 — Testing the MFA Login with some Python Test Code**

So now we’ve got MFA working with Vault. Let’s move on to some basic test code! I emphasize basic here because as I mentioned, I am not a software developer, and my goal here is to make the code as simple as possible so that you can convert it whatever code you are comfortable with. I’ve kept the code to only 50 lines (including comments).

Here it is:

Or get the code directly from Github: <https://github.com/tjabraha/vault_mfa_example>

Some notes about this code:

1. It requires Python3. Install it on your computer if you don’t have it already.
2. And the excellent [requests library](https://requests.readthedocs.io/en/master/) to be installed. All other libraries (json and getpass) are part of the Python3 standard library. I recommend using a virtual environment of some sort if you are a Python user and are concerned about the Python package versions on your computer.
3. You will need to replace the value for the vault\_root\_token variable with your Vault root token.
4. The password will display on the command line. I have left that in for testing purposes. To prevent the password from being displayed, follow the instructions in the Python file.

Let’s get started:

1. Copy the embedded code into a file called login.py.
2. Enter the root token for your Vault dev instance into this variable in the file and save it:

vault\_root\_token = '<enter your Vault root token>' # Enter in the root token for your Vault Dev Instance

3. Install the requests library

pip3 install requests

4. Execute the code

python3 login.py

5. Follow the prompts in the program.

Username: someuser  
Password: Howdy123  
Enter the MFA Token: <the authy code from your phone>

6. If everything was entered correctly, you will get a Login Succeeded message.

So how does this code work?

1. The first thing it does is prompt the user for their login information. The first thing it asks for is the username which is *someuser,*since that is the only user we have setup. Then it asks for the password which is *Howdy123.*
2. After that it assigns all the variables needed for the Vault api. This includes the api urls to get to the hashed secret we created in the first section of the tutorial as well as the api url endpoint for the hash algorithm and the TOTP endpoint
3. It then grabs the password input from the user and uses the Vault api endpoint to take the raw password input and hash it with the the sha2–256 algorithm and loads it into a variable in your systems memory
4. It then pulls the value from the hashed password we set in the first section of the tutorial by referencing the Vault api endpoint of the secret. (Remember we converted the password *Howdy123*through the hash algorithm and saved that hex value in the secret path for Vault.)
5. It then parses the json from the Vault api, grabbing only the values we are concerned with.
6. It then does a string compare between the value entered into the secrets path with the value returned from the sha2–256 algorithm. (hex = *0ee78709ab2a4a9b404b7676ff0abe3bbb3b7f44f2940b4e88d7cfa694fa9fa9)*. If the two values match, the main logic of the program will then ask for the MFA token from Authy on your mobile phone, else it will fail and respond that the login failed.
7. When you enter the valid Authy token from your phone, it will run another check to validate that the token is valid.
8. If the TOTP token from your phone matches the TOTP code from Vault, then it will authenticate the user and indicate that your user has been logged in.

**Take-away Concepts for this Section**

This code demonstrates how to use the Vault API and running authentication against both the password secret that was entered as well as by authenticating against the TOTP token. It should be noted that the code relies on the Vault root token. ***This is a terrible, terrible practice for anything other than development work. In a real world production scenario, you should use some other form of authentication method.***If you are interested in using Vault for your app or organization, you should use some other form of [authentication method](https://www.vaultproject.io/docs/auth)to authenticate to Vault. However, for this example, it’s the easiest way to demonstrate the abilities of Vault Multi Factor Authentication.