authenticator

9-11 minutes

---

authenticator is a CLI analog to the Google Authenticator phone app, or the LastPass Authenticator phone app. It is a TOTP/HOTP client that can generate the numeric codes needed for authentication with sites that support Two-Factor Authentication (TFA) or Multi-Factor Authentication (MFA).

- Benefit
- Authentication secret (READ THIS)
- System Requirements
- Installation
- Usage
- Implementation details
- Development
- License

Benefit

The benefit of using authenticator over a phone app is that this CLI utility can run anywhere Python 3.5 can run from a command line interface (e.g., a terminal window), and the database of accounts and secrets is a platform-independent passphrase-protected encrypted file that can be backed up and can be copied to multiple systems without fear of bad actors gaining access to the second factor authentication.

Another benefit is that authenticator can act as a backup in case you loose your phone or tablet (running Google Authenticator) or Google breaks the app or withdraws it.

Personally, I use both Google Authenticator on my iPhone and iPad, and run authenticator on several different computer systems. I keep a copy of the accounts file in a variety of places. If my phone bricks, is lost or stolen, I can still get access to my TFA-protected accounts if I can access any of those other computers, or any other computer on which I can install and run authenticator and access a copy of my accounts file.

Authentication secret (READ THIS)

TFA/MFA clients that produce a ‘one time’ numeric code require a secret that they share with the server validating the authentication. Typically this secret is generated by the server and captured by the client in the form of a QR code that can be scanned by the client app on your

phone.

But this CLI utility doesn't have the ability to capture or process QR images. It needs the secret in the form of a text string. If the server cannot provide a text string equivalent of the shared secret then you cannot use authenticator to generate one-time codes for that account.

Each account will use a different secret, a different text string. If you are already using a phone app to generate one-time TFA codes for some accounts then you'll need to generate new secrets for those accounts (and update the info in the phone app) so that you can capture the secret as a text string.

GMail TFA example

If you enable Two-Factor Authentication for GMail, you'll go through a setup process that first configures your account to send your phone a code by text or voice. At the end of the setup you'll have an opportunity to add Google Authenticator as an alternative code generator. Select that and scan the QR code with your phone's Authenticator app (Google Authenticator, LastPass Authenticator, et cetera). Then click the link labeled "CAN'T SCAN IT" under the QR code … that will give you a 32-character secret string that you'll use to add configure this account in authenticator.

It is important that you capture the QR code and *then* click the link for the secret string secret code. If you capture the secret and click "back" to get the QR code then a new secret will be generated and the new QR won't match the previously captured secret string. (With Google Authenticator you actually don't need the QR code; you can also provide the secret as a text string using the 'manual entry' option when adding an account to the app. LastPass Authenticator doesn't have that option.)

You add the account and secret to authenticator like so:

```
$ authenticator add Google:example@gmail.com
Enter passphrase:
Enter shared secret: xj6p kokw ipvk usc6 bveu sz3b csir xhbu
OK
```

You then generate codes like so (use Ctrl-C to stop the generation):

```
$ authenticator generate
Enter passphrase:
Google:example@gmail.com 162534 (expires in 12 seconds)

Google:example@gmail.com 162534 (expires in 7 seconds)

Google:example@gmail.com 162534 (expires in 2 seconds)

Google:example@gmail.com 996752 (expires in 27 seconds)

Google:example@gmail.com 996752 (expires in 22 seconds)
^C
```

System Requirements

This requires Python 3.5 or later.

It has been tested on OS X 11.9.5, Windows 10, and Ubuntu 14.04. As none of those systems come with Python 3.5 out of the box, you'll need to install that yourself.

And I recommend setting up a Python 3.5 virtual environment in which to install authenticator.

Installation

Installation is simple:

```
pip install authenticator

authenticator --help
```

Usage

Add an account

To add a new account, do something like:

```
authenticator add Google:example@gmail.com
```

You can use any string there as the name. I recommend the format 'vendor:userid', where *vendor* is some string indicating the organization or server that will check your TFA credentials, and *userid* is the user account id that is being authenticated.

You'll get prompted for the passphrase to unlock the file in which all the account secrets are stored. And then you'll get prompted for a secret string.

Generate current codes

To get the current code for all the accounts, do:

```
authenticator generate
```

You'll get prompted for the passphrase, and then the program will start generating the current passcode for all the accounts. It will continue to generate current codes every 5 seconds until you stop it with ctrl-C.

Other commands and options

There's a lot more, just enter authenticator --help for a list of all the commands and something

like authenticator add --help for detailed help on a specific command.

```
$ authenticator --help
usage: authenticator [-h] [--version] [--data ALTDATAFILE]
                     {add,delete,del,generate,gen,info,list,set}
...

Run or manage HOTP/TOTP calculations

optional arguments:
  -h, --help            show this help message and exit
  --version             show the software version
  --data ALTDATAFILE    Specify the path to an alternate data file

Sub-commands:
  Valid Sub-Commands

  {add,delete,del,generate,gen,info,list,set}
                        Sub-command Help
    add                 add a HOTP/TOTP configuration
    delete (del)        delete a HOTP/TOTP configuration
    generate (gen)      generate passwords for one or more HOTP/
TOTP
                        configurations
    info                show information about this software and
your data
    list                list HOTP/TOTP configurations
    set                 set HOPT configuration values
```

```
$ authenticator add --help
usage: authenticator add [-h] [--counter COUNTER] [--length
PASSWORDLENGTH]
                         [--period PERIOD]
                         clientIdToAdd

Add a new HOTP/TOTP configuration to the data file.

positional arguments:
  clientIdToAdd         a unique identifier for the HOTP/TOTP
configuration

optional arguments:
  -h, --help            show this help message and exit
  --counter COUNTER     initial counter value for a counter-based
HOTP
                        calculation (no default)
  --length PASSWORDLENGTH
                        length of the generated password (default:
6)
```

```
  --period PERIOD       length of the time period in seconds for a
time-based
                        HOTP calculation (default: 30)
```

Implementation details

This is a simple attempt to implement the “Pseudocode for Time OTP”and “Pseudocode for Event/Counter OTP” given in the Wikipedia article on Google Authenticator. That pseudocode is reproduced here …

Pseudocode for Time OTP

```
function GoogleAuthenticatorCode(string secret)
    key := base32decode(secret)
    message := current Unix time ÷ 30
    hash := HMAC-SHA1(key, message)
    offset := last nibble of hash
    //4 bytes starting at the offset
    truncatedHash := hash[offset..offset+3]
    //remove the most significant bit
    Set the first bit of truncatedHash to zero
    code := truncatedHash mod 1000000
    pad code with 0 until length of code is 6
    return code
```

Pseudocode for Event/Counter OTP

```
function GoogleAuthenticatorCode(string secret)
    key := base32decode(secret)
    message := counter encoded on 8 bytes
    hash := HMAC-SHA1(key, message)
    offset := last nibble of hash
    //4 bytes starting at the offset
    truncatedHash := hash[offset..offset+3]
    //remove the most significant bit
    Set the first bit of truncatedHash to zero
    code := truncatedHash mod 1000000
    pad code with 0 until length of code is 6
    return code
```

I’ve validated the pseudocode and this implementation against RFC6238 (TOTP), RFC4226 (HOTP) and RFC4648 (Base32 encoding).

Dependencies

This implementation requires:

- Python 3.5 or later
- cryptography 1.4
- iso8601 0.1.11

Development

To setup the development environment on OS X, clone the repo from GitHub, and then cd in Terminal to the root of the cloned repository and do:

1. dev/venv/make-venv.sh
2. . dev/venv/activate-project.src
3. dev/venv/provision-venv.sh
4. dev/lint.sh
5. dev/runtests.sh

To setup the development environment on OS X, clone the repo from GitHub, and then cd in Terminal to the root of the cloned repository and do:

1. dev/venv/make-venv.ps1
2. dev/venv/activate-project.ps1
3. dev/venv/provision-venv.ps1
4. dev/lint.ps1
5. dev/runtests.ps1

You can find out more about why the virtual environment is setup and managed that way by looking at these blog posts:

- Using Virtual Environments - Python I
- Using Virtual Environments - Python II

I build the distribution using dev/build/make-package.sh.

License