Exploring age v 1.0

Posted on September 6, 2021

About one year ago I learned and wrote about age, a "simple, modern and secure encryption tool (and Go library) with small explicit keys, no config options, and UNIX-style composability." Basically it's a new tool for encrypting files, and it just hit version 1.0.0. Exciting!

If you're like me and have struggled to learn and use PGP to encrypt files, you might find age worth checking out. (Note that for encrypted chat, you're better off with something like Signal or Wire.)

Mostly for my future self, I wrote out the basic usages below. Maybe you'll find it useful as well.

Installing age on Ubuntu 20.04

If I were running macOS, Ubuntu 21.04+ or a number of other Linux distros, I could use a package manager to install age.

However, I'm still on Ubuntu 20.04, so I need to build it from source. Luckily, I already have Go v 1.16 installed, so following the README, I ran:

```
git clone https://filippo.io/age && cd age
go build -o . filippo.io/age/cmd/...
```

That `go build` command created two binaries for us, `age` and `age-keygen`, both of which we'll use later.

Since I wanted access to these binaries anywhere on my system, I needed to move them to somewhere in my PATH.

After some deliberation, I chose `~/.local/bin`. To actually move the binaries, I ran:

```
mv age ~/.local/bin
mv age-keygen ~/.local/bin
```

You should now be able to run `age` from any directory in your file system. We're ready to try it out!

Using age to encrypt a file with a passphrase

Let's say we have a file called secrets.txt that we want to encrypt, such that only someone who knows a certain passphrase can decrypt it. We can just run:

```
age -p secrets.txt > secrets_encrypted.txt
```

Your terminal with then prompt you to enter your chosen the passphrase (if you don't provide one, age will generate a nice long one for you). I left the passphrase field blank so that age would generate a passphrase for me. Here's what I got:

```
$ age -p secrets.txt > secrets_encrypted.txt
Using an autogenerated passphrase:
    skull-tribe-mirror-help-logic-maple-syrup-round-click-danger
```

Age created a new file called "secrets_encrypted.txt" that is encrypted and thus unreadable. The original secrets.txt is still in tact.

Decrypting file with a passphrase

Decrypting this file is simple. Just run

```
age -d secrets.txt.age > secrets_decrypted.txt
```

Running this command prompts you to enter the passphrase we set when encrypting the file. If you enter the passphrase correctly, age decrypts the file into a new file called secrets_decrypted.txt. Again, the original "secrets.txt.age" file is left in tact.

Using a public/private key-pair

Encrypting and decrypting files with secret passphrases can be useful, but one downside is that you have to securely get that passphrase to the other party. We can avoid this issue by using a public/private key-pair, which age also supports.

When we installed age, we got two executables: `age`, which we've been using, and `age-keygen`, which we haven't used yet. `age-keygen`, as you might have guessed, generates key-pairs for us. Let's make a key-pair now.

```
age-keygen -o key.txt
```

Running this command will print our new *public* key.

```
Public key: age19gxvanpygn3mw50l8achn5r9ml7p3rwntgq0c95w77kwp83kzqjsqxkzhd
```

More importantly, this command created a file called `key.txt` that has both this public key and our private or *secret* key. You should not share your secret key with anyone, but you can check the file now to verify that it has both a "public key" and an "AGE-SECRET-KEY"

Encrypting a file for a public key

Let's say someone else wants to encrypt a file for us, the holder of the key above. Let's say the file they want to encrypt for us is secrets.txt. They'd run

```
age -r age19gxvanpygn3mw50l8achn5r9ml7p3rwntgq0c95w77kwp83kzqjsqxkzhd secrets.txt >
secrets.txt.age
```

Here, we're telling age that we want to encrypt this file for a particular *recipient* (`-r`), namely the owner of this specific public key (remember, *public* keys can be shared safely out in the open). Then we tell age which file we want to act on, in this case "secrets.txt", the file we wish to encrypt. And lastly we `>` to an encrypted output file, which age will create for us.

Once we hit enter, age creates a new, encrypted file called `secrets.txt.age`.

Encrypting for a key-pair file you have access to (example: archiving for your future self)

If you want to encrypt a file for a key-pair file that you have access to, it looks like you'd run this:

```
age --encrypt -i ~/age/archive_key.txt secrets.txt > secrets.txt.age
```

Decrypting a file with a secret key

Now let's say we receive a file that's been encrypted for us, using our public key. We want to decrypt it. We'll run:

```
age --decrypt -i key.txt secrets.txt.age > secrets_decrypted.txt
```

Which will decrypt the file to `secrets_decrypted.txt`. Note the `-i` flag (which stands for "identity") pointing to the key-pair file (which contains the secret key). Here we used the `--decrypt` flag for decryption, but we could have also used `-d`, which we're already familiar with.

Should you create long-term, persistent age keys?

I've gleaned from the creator of age, Filippo Valsorda, that he's encouraging age users to create keys on a per-project basis, in order to compartmentalize security. (Of course, there's nothing stopping you from generating a key file somewhere central like `~/age/key.txt` and using that for everything.)

Let me know if you understand otherwise.

Can you encrypt/decrypt *text* rather than a file?

What if you just want to encrypt a line or two of text, rather than a file? Age seems to be primarily built for encrypting and decrypting files, but we can use UNIX pipes to encrypt short strings of text.

Specifically, you can "pipe" in an echo command (note that in this case, age requires that you encrypt using a public key, rather than a passphrase):

```
$ echo "secret message from echo" | age -a -r
age19gxvanpygn3mw50l8achn5r9ml7p3rwntgq0c95w77kwp83kzqjsqxkzhd
```

which prints the encrypted text to the terminal:

```
-----BEGIN AGE ENCRYPTED FILE-----
YWdlLWVuY3J5cHRpb24ub3JnL3YxCi0+IFgyNTUxOSBVNFB2QVdBSXNYc2pnQzFO
b2xEQjVwVitzK3cxMlo5bnQ3OFJnKzlYUzB3CjhLdFRkQlBCS0pwTHpQSVdSV24y
bU8wRHE1RGI1UmxwOVRaSzhvRHc2ODgKLS0tIGk3aG56aU03bHRSbE1UaWQxVUx6
WVQ2a25tTWtpZFR3cTBUVU1yYkU1dUEKyqs9c4eHWjA/sZvbVPhfFMSwxooYIx0F
pRQr+t+MK5iEwUdZsa4tk+IQznL6GN4tXDIL75bGlqtKIsinWKbl3R0=
-----END AGE ENCRYPTED FILE-----
```

The `-a` flag stands for “armor” and it tells age to encrypt to a PEM encoded format, which is easier to copy and paste and move around in general.

Having this block of text printed to the terminal screen isn’t super helpful. We already know how to print it to a file. But this time, how about we pipe it directly to the clipboard? If you’re using Linux and have `xsel` installed, you’d do this:

```
echo "secret message from echo" | age -a -r
age180d9ut0ff3zzkq6umq588p7zlqqetuf8nhxxfhsysmg4hjyt55lsjraysp | xsel --clipboard
```

(On Mac, try `| pbcopy` at the end, though I haven’t tested this. Other systems may have yet other tools for this – another popular one is called xclip.)

To decrypt an encrypted message currently in your clipboard with your stored key, you could run:

```
xsel --clipboard | age -d -i key.txt
```

which should print the decrypted message to the terminal screen.

An actual use-case: Encrypting a directory before storing on the cloud

I’ve got a small directory of very important documents which I want to back-up somewhere, in this case on Dropbox. However, I want it to encrypt it before I upload it to Dropbox. Let’s use age!

Step 1: Compressing with tar

It’s a directory (rather than a single file), so the first thing I’m going to do is put it in a tar ball.

```
tar -czvf important_documents.tar.gz important_documents/
```

Running this command creates important_documents.tar.gz for us. This single, compressed file will be easier for us to encrypt. (If you need more compression, try `tar -cjvf important_documents.tar.bz2 important_documents/`.)

Step 2: Encrypting with age

Now we’re ready to encrypt. As we learned above:

```
age -p important_documents.tar.gz > important_documents.tar.gz.age
```

Enter a new passphrase twice. Age will create an encrypted file called `important_documents.tar.gz.age`.

We can safely upload this `important_documents.tar.gz.age` file to Dropbox or another unencrypted cloud provider (I do this through the website, but I'm sure there's a way to do it through the command line...).

Step 3: Decrypting and decompressing

We of course need to be able to restore these files. First we decrypt:

```
age -d important_documents.tar.gz.age > important_documents.tar.gz
```

Enter your passphrase to decrypt to `important_documents.tar.gz`. Then we extract the files from the tar ball:

```
tar -xzvf important_documents.tar.gz
```

All together (one-liners)

I think these work.

```
# compress and encrypt
tar -czv important_documents/ | age -p > important_documents.tar.gz.age
# decrypt and extract
age -d important_documents.tar.gz.age | tar -xzv
```

Bottle

I've now written a shell script that wraps around some of these tar and age commands to make them (even) easier to use. I call it Bottle.

How this would look using symmetrical GPG encryption

Just for my notes, here's how we could do something similar with GPG:

Encrypting:

```
gpg -c important_documents.tar.gz
```

Decrypting:

```
gpg --output important_documents.tar.gz --decrypt important_documents.tar.gz.gpg
```

More age features

This guide is by no means exhaustive of what you can already do with age. Age definitely has more features than I've outlined here, which you can read about on the project's README, including some nifty things with SSH keys.

Also, here's a blog post with some helpful shell scripts for using age. (h/t Aaron Toponce)

To look forward to

There's an interoperable Rust port of age called rage, currently at version 0.6.0. I've tried it and it seems to work well, though the syntax of the commands is now a bit different than age v1.0.

I'm also excited to test this Yubikey plugin for age, which is written in Rust.

← PREVIOUS POST

NEXT POST →