AN OPINIONATED YUBIKEY SET-UP GUIDE

This article will show you how to set up a brand new YubiKey 5 with the following applications:

- OpenPGP: used for encrypting and signing PGP (aka GPG) messages, as well as authenticating with SSH or WireGuard.
- PIV: used for additional encryption and signing keys (and signature-based authentication) through the PKCS #11 smartcard interface.
- FIDO (both U2F and FIDO2 flavors): used for browser-integrated "passwordless" authentication (aka Passkeys) and 2FA (2-Factor Authentication) with WebAuthn; or for local terminal/desktop log-in and sudo authentication via PAM (Pluggable Authentication Modules); or for other signature-based authentication like SSH.
- OATH: used with the Yubico Authenticator app for 2FA with TOTP (Time-based One-Time Password) or HOTP (HMAC-based One-Time Password).
- OTP: used for YubiCloud two-factor authentication; or for one or two static passwords.

(We won't cover the YubiKey's YubiHSM Auth application, which is used to store management credentials for a YubiHSM, since it's relevant only if you manage YubiHSM devices.)

PRELIMINARIES

THREAT MODEL

This guide assumes the primary reason why you want to use a YubiKey is that you fear at some point a remote adversary who's targeting you specifically will gain root access on your local computer. With root on your computer, the adversary will be able to log your keystrokes and sift through your computer's memory to identify and exfiltrate all the passwords and private keys you use regularly, as well as steal credentials stored on your computer's hard drives (or remotely accessible from your computer).

Stealing your passwords, private keys, and other credentials would allow the adversary to use those credentials to log into any publicly-accessible web application or server at a later time from any other computer in the world. She would also be able to decrypt any files she had stolen previously (or steals later) that had been encrypted for your private keys. And she would be able to sign any messages, certificates, or other files at any time from any computer with the keys she had stolen.

By using a YubiKey to store your private keys, and by using your YubiKey for passwordless authentication or 2FA to various websites, you can limit an adversary's use of your private keys and other credentials. Only while the adversary maintains her access on your local computer will she be able to use your private keys or be able to log into websites or other servers using your credentials.

Also, as long as you always keep your YubiKeys on your person whenever you're away from your computer (or keep them in a secure location to which only you have access, like a personal safe), you can prevent an adversary from using your private keys and website credentials while you're away from your computer — even for an adversary who has physical access to your computer. (Of course, an adversary with physical access can still do bad things to your computer, like set up permanent remote access for herself.)

This guide assumes that even if you store your credentials on a YubiKey, a remote adversary with access to your computer will be able to trick you into touching your YubiKey whenever she needs to use them. Therefore, the YubiKey's touch requirements provide only a "defence in depth" benefit, forcing the adversary go to the trouble of ensuring that you're at your computer and are expecting to have to touch your YubiKey whenever she needs to use a private key or other credential stored on your YubiKey.

OTHER ASSUMPTIONS

Some other opinions that pervade this guide:

- ECC (Elliptic Curve Cryptography) is preferable to RSA (factoring large numbers). ECC keys are smaller, and ECC operations run faster and use less power on most hardware. Furthermore, all the curves available on the YubiKey are at least as strong or stronger than RSA-2048 against classical-computing attacks. (On the other hand, quantum computers may be more likely to crack even the strongest of the YubiKey's ECC keys before they are able to crack RSA-2048 keys.)
- All private keys should be generated on the YubiKey, instead of generating the keys elsewhere and importing them into the YubiKey later. This guarantees that only the YubiKey holder can use the YubiKey's private keys. (See the Backing Up section for alternatives to backing up your keys.)

- All applications on the YubiKey that can be protected by PIN or passphrase should be. This prevents a common thief who steals your YubiKey from using any of the credentials stored on it — she must not only steal your YubiKey, but she must also spend a lot of effort observing you using it beforehand in order to steal your PINs, too.

YUBIKEY MANAGER

This guide will show you how to use the YubiKey Manager CLI (aka ykman) to set up each YubiKey application — see the YubiKey Manager Installation page for installation options. If you don't use a package manager to install the ykman CLI, you most likely will have to install the pcsc-lite daemon (aka pcscd) separately.

On a Debian-based Linux distro (like Ubuntu and friends), if you **don't** use a package manager to install the ykman CLI, you'll have to install these dependencies in order to build and run the Python `yubikey-manager` package manually:

```
$ sudo apt install libpcsclite-dev pcscd swig
...
$ pipx install yubikey-manager
```

And on a Fedora-based distro (like RHEL and friends), if you **don't** use a package manager to install the ykman CLI, you'll have to install these dependencies for the Python `yubikey-manager` package:

```
$ sudo dnf install pcsc-lite pcsc-lite-devel swig
...
$ pipx install yubikey-manager
```

Otherwise, on most Linux distributions, if you **do** use the distro's package manager to install the ykman CLI, the package manager will automatically install the necessary dependencies for you; like on Debian:

```
$ sudo apt install yubikey-manager
```

Or on Fedora:

```
$ sudo dnf install yubikey-manager
```

Once you have the ykman CLI installed, plug your YubiKey into your computer, and run the following command to see its details:

```
$ ykman info
Device type: YubiKey 5 NFC
Serial number: 12345678
Firmware version: 5.4.3
Form factor: Keychain (USB-A)
Enabled USB interfaces: OTP, FIDO, CCID
NFC transport is enabled.

Applications    USB     NFC
OTP             Enabled Enabled
FIDO U2F        Enabled Enabled
FIDO2           Enabled Enabled
OATH            Enabled Enabled
PIV             Enabled Enabled
OpenPGP         Enabled Enabled
YubiHSM Auth    Enabled Enabled
```

Each one of the listed applications has its own, separate PIN or passphrase that is set independently of the other applications (except for the FIDO U2F and FIDO2 apps, which share a single FIDO management interface). Each app's PIN or passphrase controls access to the configuration and secrets stored by the app. The config and secrets of each app can be wiped independently of the other apps.

YUBIKEY LOCK CODE

There is one PIN that rules them all, however. The YubiKey has a master config application that allows you to enable or disable each individual app on the YubiKey. When an application is disabled, its configuration and secrets cannot be accessed or changed — or wiped. This is particularly important to take note of, since otherwise wiping an app's config and secrets (aka a "factory reset") **does not** require the app's own PIN or passphrase.

The YubiKey allows you to set a 128-bit "lock code" to protect changes to this master config app. By default, your YubiKey comes with no lock code set. However, in order to prevent an adversary who gains access to your computer while your YubiKey is plugged in from being able to lock you out of all your YubiKey secrets, you should set this lock code.

Unfortunately, you must always enter this lock code as a string of 32 hex digits, which makes it cumbersome to use. Fortunately, most people will need to use it rarely, if ever.

Run the following command to set the lock code for your YubiKey to a new random number:

```
$ ykman config set-lock-code --generate
```

This will generate a new random number (as 32 hex digits), and print it out:

```
Using a randomly generated lock code: 10a43eacde8630603fc15016bc508605
Lock configuration with this lock code? [y/N]: y
```

Enter `y` at the prompt to set this as the new lock code. Write this code down on a piece of paper (along with all the other PINs and passphrases you set for the other apps on your YubiKey; see the Backing Up PINs section at the end of the article).

You'll have to enter this lock code in the future to enable or disable any apps on your YubiKey. For example, to disable the NFC interface to the YubiHSM Auth app, run the following command:

```
NFC configuration changes:
  Disable YubiHSM Auth
```

Enter `y` to confirm, then enter your 32 hex-digit lock code (`10a43eacde8630603fc15016bc508605` in this example):

```
Proceed? [y/N]: y
Enter your lock code:
```

If you do the same for the USB interface (`ykman config usb --disable HSMAUTH`), and then run the `ykman info` command again, you'll see that the YubiHSM Auth app has been disabled completely on your YubiKey:

```
$ ykman info
Device type: YubiKey 5 NFC
Serial number: 12345678
Firmware version: 5.4.3
Form factor: Keychain (USB-A)
Enabled USB interfaces: OTP, FIDO, CCID
NFC transport is enabled.
Configured capabilities are protected by a lock code.

Applications    USB         NFC
OTP             Enabled     Enabled
FIDO U2F        Enabled     Enabled
FIDO2           Enabled     Enabled
OATH            Enabled     Enabled
```

```
PIV              Enabled      Enabled
OpenPGP          Enabled      Enabled
YubiHSM Auth     Disabled     Disabled
```

You can re-enable an app by running the same commands with the `--enable` option instead of the `--disable` option:

```
$ ykman config usb --enable HSMAUTH
USB configuration changes:
  Enable YubiHSM Auth
Proceed? [y/N]: y
Enter your lock code:
```

OPENPGP

We'll set up your OpenPGP application with an Ed25519 signing key for signing messages, an X25519 decryption key for decrypting messages, and an Ed25519 authentication key for signature-based authentication (such as for SSH). We'll configure it with a strong PIN, and make touch required for each use of your private keys. If you don't already have GnuPG (aka gpg) with its smartcard daemon (aka scdaemon) installed on your computer, you'll need to install it now.

On some Linux distributions, like Fedora and friends, scdaemon comes installed with the main GnuPG package (`gnupg2`). On others, like Debian and friends, you have to install a separate package (`scdaemon`) in addition to the main GnuPG package.

TIP

If your package manager has installed a version of GnuPG from the 2.2.x branch, and you use the OpenPGP application with some other applications of your YubiKey regularly, you may want to manually build and install the latest version of GnuPG, which will work better with the other applications of your YubiKey. See the Installing GnuPG 2.4 on Ubuntu 22.04 article for a step-by-step guide.

You might also find you need to add the `disable-ccid` and `pcsc-shared` flags to your `~/.gnupg/scdaemon.conf` config file to resolve GnuPG and PC/SC conflicts:

```
# ~/.gnupg/scdaemon.conf
disable-ccid
pcsc-shared
```

Once you have GnuPG and its smartcard daemon installed and ready, update your `~/.gnupg/gpg.conf` file to include the following settings:

```
# ~/.gnupg/gpg.conf
display-charset utf-8
keyid-format 0xlong
keyserver hkp://keys.openpgp.org

personal-cipher-preferences AES AES256
personal-digest-preferences SHA256 SHA512
personal-compress-preferences BZIP2 ZLIB Uncompressed
default-preference-list AES AES256 SHA256 SHA512 BZIP2 ZLIB Uncompressed
cert-digest-algo SHA256
s2k-digest-algo SHA512
s2k-count 65011712
```

This sets your GnuPG preferences to use UTF-8 and the "long" key ID format (eg `0x1234567890ABCDEF`) when displaying keys, to help mitigate any tricks an adversary might attempt with similar-looking key identities. It also directs GnuPG to use AES and SHA-2 where possible (instead of some of the older or unusual crypto algorithms it also supports) — the `default-preference-list` option configures the preferences that will be embedded in the new keys you generate, while the `personal-*-preferences` options allow you to filter and order the preferences from other people's keys whenever you encrypt or sign a new message. And keys.openpgp.org is one of the few public keyservers still in operation and accepting new keys.

Run the card-status command to check the status of the OpenPGP application on your YubiKey:

```
$ gpg --card-status
Reader ...........: Yubico YubiKey OTP FIDO CCID 00 00
Application ID ...: D2760001240100000006123456780000
Application type .: OpenPGP
Version ..........: 3.4
Manufacturer .....: Yubico
Serial number ....: 12345678
Name of cardholder: [not set]
Language prefs ...: [not set]
Salutation .......:
URL of public key : [not set]
Login data .......: [not set]
Signature PIN ....: not forced
Key attributes ...: rsa2048 rsa2048 rsa2048
Max. PIN lengths .: 127 127 127
PIN retry counter : 3 0 3
Signature counter : 0
KDF setting ......: off
Signature key ....: [none]
Encryption key....: [none]
```

```
Authentication key: [none]
General key info..: [none]
```

To set up a new set of OpenPGP keys, run the following command:

```
$ gpg --edit-card
...

gpg/card>
```

At the prompt, run the `admin` command to toggle into the admin mode:

```
gpg/card> admin
Admin commands are allowed

gpg/card>
```

Run the `help` command to list the available commands:

```
gpg/card> help
quit           quit this menu
admin          show admin commands
help           show this help
list           list all available data
name           change card holder's name
url            change URL to retrieve key
fetch          fetch the key specified in the card URL
login          change the login name
lang           change the language preferences
salutation     change card holder's salutation
cafpr          change a CA fingerprint
forcesig       toggle the signature force PIN flag
generate       generate new keys
passwd         menu to change or unblock the PIN
verify         verify the PIN and list all data
unblock        unblock the PIN using a Reset Code
factory-reset  destroy all keys and data
kdf-setup      setup KDF for PIN authentication (on/single/off)
key-attr       change the key attribute
uif            change the User Interaction Flag

gpg/card>
```

WARNING

Three commands listed are unreversable: `generate`, `factory-reset`, and `kdf-setup`. The `generate` command will overwrite any existing OpenPGP keys on the YubiKey, and the `factory-reset` command will wipe them. The `kdf-setup` command will reset your existing PINs to the defaults.

In particular, just don't ever run the `kdf-setup` command (which sets up the YubiKey to require that clients of the OpenPGP app run a KDF on your PINs before providing them to the YubiKey) — this will cause incompatibilities with some OpenPGP clients, and it doesn't provide a practical security benefit unless you re-use the same PIN with different security cards or applications.

We will run the `generate` command presently to generate a new set of OpenPGP keys; however, you usually will need to run this command only one time ever.

If you want to start over from scratch with OpenPGP on your YubiKey (now, or some point in the future), you can run the `factory-reset` command. It will wipe your existing OpenPGP keys and settings from your YubiKey — so you won't be able to use the OpenPGP keys you had previously generated on the YubiKey ever again for signing, decryption, or authentication. It won't affect any of the other applications on your YubiKey (like PIV, FIDO, or OATH), however.

If you do run the `factory-reset` command after having already generated a set of OpenPGP keys on your YubiKey that you intend never to use again, make sure you also run the `gpg --delete-secret-and-public-key` command (once you exit the edit-card CLI) with the ID of the old master key, to delete the previously stored information about them from your computer's GnuPG keyring (otherwise you and GnuPG may become confused about which are the new keys on the YubiKey, and which are the old keys that are no longer available).

OPENPGP PINS

First, use the `passwd` command to change the existing PINs from their defaults:

```
gpg/card> passwd
gpg: OpenPGP card no. D2760001240100000006123456780000 detected

1 - change PIN
2 - unblock PIN
3 - change Admin PIN
4 - set the Reset Code
Q - quit

Your selection?
```

The OpenPGP app has 3 PINs: the user PIN, the reset code (aka PUK, PIN Unblock Key), and the admin PIN. The user PIN is the one you use for day-to-day access to your OpenPGP private keys; you don't need and won't set a reset code; and the admin PIN is used to change the settings of the OpenPGP app itself (and to unblock or change the user PIN if you forget it or enter it incorrectly too many times in a row). When prompted to enter your "PIN" with no "admin" qualifier, this usually means your should enter the user PIN, not the admin PIN.

Enter `1` at the prompt for the `passwd` command, and then enter `123456` at the first PIN prompt (the default user PIN is `123456`). Next, enter the new user PIN (you'll be prompted for it twice). Do not use a number — instead use a simple passphrase (in the "correct horse battery staple" vein) that's at least 6 characters long and easy to type (you'll need to type it frequently, probably at least several times a day). Make sure it's different than any other PIN or passphrase you've ever used before.

TIP

Generate a good random PIN by running the following command:

```
$ shuf -n2 /usr/share/dict/words
society's
mangroves
```

This PIN doesn't need to be anywhere as good as a normal passphrase you'd use elsewhere, since the YubiKey will automatically block the use of the OpenPGP application after several consecutive wrong PIN attempts (more on that later) — it just needs to be outside the first 100 or so guesses an adversary would try. Any good random word or two will do — the only requirement is that it be unrelated to facts about yourself (you should assume an adversary will know every possible personal fact about you, like your birthday, your kid's names, your favorite sports teams, the places you've lived, etc).

The CLI will output `PIN changed` if successful:

```
Your selection? 1
PIN changed.

1 - change PIN
2 - unblock PIN
3 - change Admin PIN
4 - set the Reset Code
Q - quit

Your selection?
```

Next, enter `3` to set the admin PIN, and then enter `12345678` at the first PIN prompt (the default admin PIN is `12345678`). Next, enter the new admin PIN (you'll be prompted for it twice). Like with the user PIN above, do not use a number — instead use a simple passphrase at least 8 characters long. It doesn't need to be any stronger than the user PIN, just different (enough so that an adversary wouldn't be able to guess the admin PIN if she finds out your user PIN).

The CLI will again output `PIN changed` if successful:

```
Your selection? 3
PIN changed.

1 - change PIN
2 - unblock PIN
3 - change Admin PIN
4 - set the Reset Code
Q - quit

Your selection?
```

Write down your new PIN passphrases on paper, along with the serial number of the YubiKey, and a note that these are for the OpenPGP application (each YubiKey app, like PIV, FIDO, OATH, etc, uses its own separate set of PINs or passwords). In particular, you will rarely use your admin PIN after you finish setting up the OpenPGP application the first time, so you will surely forget the admin PIN unless you write it down.

Enter `q` to return back to the main card-edit CLI:

```
Your selection? q

gpg/card>
```

OPENPGP KEYS

Now, use the `key-attr` command so that when you generate your keys, it will generate Curve 25519 keys instead of RSA keys:

```
gpg/card> key-attr
Changing card key attribute for: Signature key
Please select what kind of key you want:
   (1) RSA
   (2) ECC
Your selection?
```

Select the `ECC` option:

```
Your selection? 2
Please select which elliptic curve you want:
   (1) Curve 25519
   (4) NIST P-384
Your selection?
```

Select `Curve 25519`:

```
Your selection? 1
The card will now be re-configured to generate a key of type: ed25519
Note: There is no guarantee that the card supports the requested size.
      If the key generation does not succeed, please check the
      documentation of your card to see what sizes are allowed.
```

Enter the admin PIN when prompted to save your changes for that key. Select the same options (`ECC` and `Curve 25519`) as the prompts continue for the encryption key and authentication key:

```
Changing card key attribute for: Encryption key
Please select what kind of key you want:
   (1) RSA
   (2) ECC
Your selection? 2
Please select which elliptic curve you want:
   (1) Curve 25519
   (4) NIST P-384
Your selection? 1
The card will now be re-configured to generate a key of type: cv25519
Changing card key attribute for: Authentication key
Please select what kind of key you want:
   (1) RSA
   (2) ECC
Your selection? 2
Please select which elliptic curve you want:
   (1) Curve 25519
   (4) NIST P-384
Your selection? 1
The card will now be re-configured to generate a key of type: ed25519
```

```
gpg/card>
```

If you list the card info, you should now see the `Key attributes` setting listed as `ed25519 cv25519 ed25519`:

```
gpg/card> list

Reader ...........: Yubico YubiKey OTP FIDO CCID 00 00
Application ID ...: D2760001240100000006123456780000
Application type .: OpenPGP
Version ..........: 3.4
Manufacturer .....: Yubico
Serial number ....: 12345678
Name of cardholder: [not set]
Language prefs ...: [not set]
Salutation .......:
URL of public key : [not set]
Login data .......: [not set]
Signature PIN ....: not forced
Key attributes ...: ed25519 cv25519 ed25519
Max. PIN lengths .: 127 127 127
PIN retry counter : 3 0 3
Signature counter : 0
KDF setting ......: off
Signature key ....: [none]
Encryption key....: [none]
Authentication key: [none]
General key info..: [none]

gpg/card>
```

Now you're ready to generate a new set of OpenPGP keys on the YubiKey, using the `generate` command:

```
gpg/card> generate
Make off-card backup of encryption key? (Y/n)
```

Enter `n` to ensure that the private keys never leave the YubiKey, and enter the admin PIN when prompted:

```
Make off-card backup of encryption key? (Y/n) n

Please note that the factory settings of the PINs are
   PIN = '123456'     Admin PIN = '12345678'
```

```
You should change them using the command --change-pin

Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
```

Then enter `0` to prevent the keys from expiring:

```
Key is valid for? (0) 0
Key does not expire at all
Is this correct? (y/N) y

GnuPG needs to construct a user ID to identify your key.

Real name:
```

When prompted for your real name, email address, and comment, use the “real name” field for the display name or alias you want associated with the OpenPGP key, the “email address” field for the email account associated with the key, and the “comment” field for a word or phrase that will distinguish this key from other keys you have used or will use in the future with the same name and email. For example, I might enter the following for real name, email address, and comment:

```
Real name: Justin Ludwig
Email address: justin.ludwig@example.com
Comment: work key 1
```

GnuPG will show this user identity (aka uid) as `Justin Ludwig (work key 1) <justin.ludwig@example.com>` when I later list all the keys I own on my keyring:

```
$ gpg --list-keys justin
pub   ed25519/0xABCDEF1234567890 2023-01-01 [SC]
      1234567890ABCDEF1234567890ABCDEF12345678
uid                   [ultimate] Justin Ludwig (work key 1) <justin.ludwig@ex
sub   ed25519/0xFEDCBA0987654321 2023-01-01 [A]
sub   cv25519/0x1234567890ABCDEF 2023-01-01 [E]

pub   ed25519/0xBCDEF1234567890A 2023-01-01 [SC]
      234567890ABCDEF1234567890ABCDEF123456789
uid                   [ultimate] Justin Ludwig (work key 2) <justin.ludwig@ex
sub   ed25519/0xEDCBA0987654321F 2023-01-01 [A]
sub   cv25519/0x234567890ABCDEF1 2023-01-01 [E]

pub   ed25519/0xCDEF1234567890AB 2023-02-01 [SC]
      34567890ABCDEF1234567890ABCDEF1234567890
uid                   [ultimate] Justin Ludwig (home key 1) <jl@example.org>
uid                   [ultimate] Justin (hk1) <justin@example.net>
sub   ed25519/0xDCBA0987654321FE 2023-02-01 [A]
sub   cv25519/0x34567890ABCDEF12 2023-02-01 [E]

pub   ed25519/0xDEF1234567890ABC 2023-02-01 [SC]
      4567890ABCDEF1234567890ABCDEF1234567890A
uid                   [ultimate] Justin Ludwig (home key 2) <jl@example.org>
uid                   [ultimate] Justin (hk2) <justin@example.net>
sub   ed25519/0xCBA0987654321FED 2023-02-01 [A]
sub   cv25519/0x4567890ABCDEF123 2023-02-01 [E]
```

NOTE

You can later add more uids to an OpenPGP key via the `gpg --edit-key` command. You can also delete existing uids from a key the same way — but deleting uids can be difficult to get completely right and correctly propagated to all copies of the key. If you publicly distribute a key, you should consider any uids distributed with the key at that time as effectively permanently attached to the key.

Continuing on with the generation process, confirm that you are happy with the uid values you selected by entering `o` when prompted:

```
Real name: Justin
Email address: justin@example.com
Comment: key1
You selected this USER-ID:
    "Justin (key1) <justin@example.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? o
```

Finally, you'll be prompted to enter the admin PIN again, and then the user PIN. Then the ID of the newly generated master key will be printed:

```
gpg: key 0xABCDEF1234567890 marked as ultimately trusted
gpg: revocation certificate stored as '/home/justin/.gnupg/openpgp-revocs.d/1
public and secret key created and signed.


gpg/card>
```

Now if you list the card details, you'll see the new keys listed:

```
gpg/card> list

Reader ...........: Yubico YubiKey OTP FIDO CCID 00 00
Application ID ...: D2760001240100000006123456780000
Application type .: OpenPGP
Version ..........: 3.4
Manufacturer .....: Yubico
Serial number ....: 12345678
Name of cardholder: [not set]
Language prefs ...: [not set]
Salutation .......:
URL of public key : [not set]
Login data .......: [not set]
Signature PIN ....: not forced
Key attributes ...: ed25519 cv25519 ed25519
Max. PIN lengths .: 127 127 127
PIN retry counter : 3 0 3
Signature counter : 0
KDF setting ......: off
Signature key ....: 1234 5678 90AB CDEF 1234  5678 90AB CDEF 1234 5678
      created ....: 2023-01-01 12:34:56
Encryption key....: ABCD EF12 3412 3456 7890  ABCD EF12 3456 7890 ABCD
      created ....: 2023-01-01 12:34:56
Authentication key: 0987 6543 21FE DCBA 0987  6543 21FE DCBA 0987 6543
      created ....: 2023-01-01 12:34:56
General key info..:
pub  ed25519/0xABCDEF1234567890 2023-01-01 Justin (key1) <justin@example.com>
sec>  ed25519/0xABCDEF1234567890  created: 2023-01-01  expires: never
                                  card-no: 0006 12345678
ssb>  ed25519/0xFEDCBA0987654321  created: 2023-01-01  expires: never
                                  card-no: 0006 12345678
ssb>  cv25519/0x1234567890ABCDEF  created: 2023-01-01  expires: never
                                  card-no: 0006 12345678


gpg/card>
```

And now if you run the `quit` command, GnuPG will print out your new keys in its usual format:

```
gpg/card> quit
pub   ed25519/0xABCDEF1234567890 2023-01-01 [SC]
      1234567890ABCDEF1234567890ABCDEF12345678
uid                   [ultimate] Justin (key1) <justin@example.com>
```

```
sub   ed25519/0xFEDCBA0987654321 2023-01-01 [A]
sub   cv25519/0x1234567890ABCDEF 2023-01-01 [E]

$
```

OPENPGP SETTINGS

Finally, we'll use the ykman CLI to adjust the PIN and touch policies for the new OpenPGP key. Run the following command to check their current settings:

```
$ ykman openpgp info
OpenPGP version:            3.4
Application version:        5.4.3
PIN tries remaining:        3
Reset code tries remaining: 0
Admin PIN tries remaining:  3
Require PIN for signature:  Once
Touch policies:
  Signature key:      Off
  Encryption key:     Off
  Authentication key: Off
  Attestation key:    Off
```

TIP

If the above command fails with an message like `ERROR: Failed to connect to YubiKey`, try killing the GnuPG agent, and then restarting the pcsc-lite dameon:

```
$ gpgconf --kill gpg-agent
$ sudo systemctl restart pcscd
```

Since you set a good random PIN for both the user and admin PINs earlier, you can relax the PIN-tries constraints a little, say to 9 each:

```
$ ykman openpgp access set-retries 9 9 9
```

Enter the admin PIN when prompted, and `y` to confirm the change:

```
Enter Admin PIN:
Set PIN retry counters to: 9 9 9? [y/N]: y
```

If sometime in the future you fail 9 times in a row to enter the user PIN correctly, the YubiKey will block you from accessing any of its OpenPGP private keys. If that happens, you can use the `passwd` sub-command of the `gpg --edit-card` command we saw earlier to “unblock” the user PIN to allow more attempts (or to change the user PIN to something else) — as long as you can correctly enter the admin PIN.

If you fail 9 times in a row to enter the admin PIN correctly, the YubiKey will block you from updating any of its OpenPGP settings, or unblocking the user PIN — permanently. As long as the user PIN itself isn’t blocked, however, even with the admin PIN blocked you can still use the OpenPGP private keys on the YubiKey. And you can always use the `factory-reset` sub-command of the `gpg --edit-card` command mentioned earlier to wipe the existing OpenPGP keys and start over from scratch, even if both user and admin PINs are blocked.

NOTE

Canceling a PIN attempt or failing to enter a PIN before the PIN-entry dialog times-out (or failing to touch the YubiKey when blinking) **does not** count as a PIN attempt — so you can always safely abort from a PIN challenge without consequences if you don’t actually want to execute the operation protected by the PIN.

One last setting you should change for the OpenPGP app: run the following command to require a touch to access the OpenPGP signature key:

```
$ ykman openpgp keys set-touch sig cached
```

Enter the admin PIN when prompted, and `y` to confirm the change:

```
Enter Admin PIN:
Set touch policy of SIG key to cached? [y/N]: y
```

Do the same for the OpenPGP decryption and authentication keys:

```
$ ykman openpgp keys set-touch enc cached
Enter Admin PIN:
Set touch policy of ENC key to cached? [y/N]: y
$ ykman openpgp keys set-touch aut cached
Enter Admin PIN:
Set touch policy of AUT key to cached? [y/N]: y
```

Whenever you need to sign or decrypt something with your OpenPGP keys, the YubiKey will blink rapidly, and you must touch the key within 15 seconds or the attempt will be rejected by the YubiKey. Using `cached` instead of `on` for the above settings means that the YubiKey will allow the use of your OpenPGP keys for 15 seconds after a touch, without requiring more touches (otherwise signing a bunch of commits after a rebase or decrypting a bunch of individual files from an archive becomes a test of patience and fine-motor skills).

OPENPGP TEST

You can run a quick test of PGP signing and decryption by running the following command (replace `key1` in the following command with the comment you used when generating your new key, or with its full key ID, like `0xABCDEF1234567890`):

```
$ echo test | gpg -se -u key1 -r key1 | gpg -d
```

You'll first be prompted for your user PIN (to sign the test), then your YubiKey will blink until you touch it. Once you touch it, you'll be prompted again for your user PIN (this time to decrypt the test). Then you should see output like the following:

```
gpg: encrypted with cv25519 key, ID 0xABCDEF1234567890, created 2023-01-01
      "Justin (key1) <justin@example.com>"
test
gpg: Signature made Mon 27 Mar 2023 01:13:36 PM PDT
gpg:                using EDDSA key 1234567890ABCDEF1234567890ABCDEF12345678
gpg:                issuer "justin@example.com"
gpg: Good signature from "Justin (key1) <justin@example.com>" [ultimate]
```

OPENPGP KEY DISTRIBUTION

Finally, you're ready to publish your new OpenPGP keys. You can export the public keys to a file that can be shared to other keyrings using the following command (replace `key1` with the comment you used when generating your new key, or with its full key ID, like `0xABCDEF1234567890`):

```
$ gpg --armor --export key1 > key1.asc
```

This file can be copied to another computer, and imported on that computer by running the inverse command:

```
$ gpg --import key1.asc
```

You can also publish your OpenPGP keys publicly by sending them to a PGP keyserver. The following command will send them to the keyserver configured in your `~/.gnupg/gpg.conf` file:

```
$ gpg --send-keys 0xABCDEF1234567890
```

And if you want to use your new authentication key for SSH authentication, you can export it with the following command:

```
$ gpg --export-ssh-key key1
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Copy the command output to a line in the `~/.ssh/authorized_keys` file on any host with which you want to use your new OpenPGP key to authenticate. If your GnuPG agent is configured with SSH support, you should be able to SSH into one of those hosts (like `bastion.example.com`) with the following command:

```
$ SSH_AUTH_SOCK=$(gpgconf --list-dirs agent-ssh-socket) ssh me@bastion.exampl
```

TIP

For SSH, FIDO generally works much better than OpenPGP — modern versions of OpenSSH have support for FIDO built-in, instead of requiring finicky interoperation with GnuPG. See the FIDO SSH section for details.

PIV

We'll set up the PIV application (PIV in this case stands for Personal Identity Verification, the NIST SP 800-73 smartcard standard produced by the US government for use in their identity cards) on your YubiKey with a second set of private keys that can be used for general asymmetric cryptography: signatures, encryption/decryption, and (signature-based) authentication. The PIV application doesn't support Curve 25519, however, so we'll use the P-384 curve for these keys instead.

Clients of the PIV application use the PKCS #11 API to access it. To enable PKCS #11 access to your YubiKey, you'll need to install the YKCS11 library. You can either build this library yourself with the yubico-piv-tool source code, or install it via package manager.

On a Debian and friends, you can install it with the `ykcs11` package; the library will be installed on your system at `/usr/lib/$(uname -p)-linux-gnu/libykcs11.so.2`:

```
$ sudo apt install ykcs11
...
$ ls -l /usr/lib/$(uname -p)-linux-gnu/libykcs11*
lrwxrwxrwx 1 root root     14 Dec  8  2021 /usr/lib/x86_64-linux-gnu/libykcs1
lrwxrwxrwx 1 root root     18 Dec  8  2021 /usr/lib/x86_64-linux-gnu/libykcs1
-rw-r--r-- 1 root root 182672 Dec  8  2021 /usr/lib/x86_64-linux-gnu/libykcs1
```

On a Fedora and friends, you can install it with the `yubico-piv-tool` package; the library will be installed on your system at `/usr/lib64/libykcs11.so.2`:

```
$ sudo dnf install yubico-piv-tool
...
$ ls -l /usr/lib*/libykcs11*
lrwxrwxrwx. 1 root root     18 Feb  7 11:06 /usr/lib64/libykcs11.so.2 -> liby
-rwxr-xr-x. 1 root root 157584 Feb  7 11:06 /usr/lib64/libykcs11.so.2.3.1
```

You don't need to have the YKCS11 library installed, however, to use the ykman CLI to configure the PIV app, or to generate keys for it.

Run the following command to view the current status of the PIV application on your YubiKey:

```
$ ykman piv info
PIV version:              5.4.3
PIN tries remaining:      3/3
Management key algorithm: 3
WARNING: Using default PIN!
WARNING: Using default Management key!
CHUID: No data available
CCC:   No data available
```

PIV PINS

The first thing to set up is to change the allowed number of PIN retries. Changing the number of retries automatically resets the PIN and PUK (PIN Unblock Key) to their factory defaults, so do this first before setting the PIN and PUK. Run the following command to set the PIN and PUK retries to 9:

```
$ ykman piv access set-retries 9 9
```

Simply press enter at the management-key prompt, then enter `123456` at the PIN prompt:

```
Enter a management key [blank to use default key]:
Enter PIN:
```

Then enter `y` to confirm the change:

```
WARNING: This will reset the PIN and PUK to the factory defaults!
Set the number of PIN and PUK retry attempts to: 9 9? [y/N]: y
Default PINs are set:
        PIN:    123456
        PUK:    12345678
```

Next, run the following command to change the PIN:

```
$ ykman piv access change-pin
```

The PIN must be 6-8 alphanumeric characters. You'll use this PIN frequently if you use the PIV app regularly; it's required to use the private keys stored in the PIV app, and to change any of its settings. It's hard to choose a great PIN with so few letters available — but because an adversary will have only 9 attempts to guess it, it doesn't have to be great — it just needs to be a password you've never used before, and isn't related to any facts about yourself (like your birthday, your kid's names, your favorite sports team, etc).

TIP

Generate a decent 6-8 character PIN that will be relatively easy to remember and type by running the following command:

```
$ awk '/^\w{6,8}$/' /usr/share/dict/words | shuf -n1
noughts
```

Enter the old PIN, `123456`, first:

```
Enter the current PIN:
```

Then enter your new PIN, twice:

```
Enter the new PIN:
Repeat for confirmation:
New PIN set.
```

Next, run the following command to change the PUK (PIN unblock code):

```
$ ykman piv access change-puk
```

The PUK must also be 6-8 alphanumeric characters. An adversary will also have only 9 attempts to guess it, so it doesn't have to be great, either. However, since you should basically never have to enter it (it's used only to unblock the PIN if you accidentally entered the wrong PIN 9 times in a row), and it can only be up to 8 characters long, you might as well use an 8-character random number for it.

TIP

Generate a random 8-character PUK by running the following command:

```
openssl rand -base64 10 | tr -d +/= | cut -c 1-8
kXHt2J1u
```

Enter the old PUK, `12345678`, first:

```
Enter the current PUK:
```

Then enter your new PUK, twice:

```
Enter the new PUK:
Repeat for confirmation:
New PUK set.
```

Write down your new PIN and PUK on the same piece of paper you wrote down your OpenPGP user PIN and admin PIN, noting that they're for the YubiKey's PIV application (each YubiKey app, like OpenPGP, FIDO, etc, uses its own separate set of PINs or passwords).

Next, run the following command to change the PIV management key:

```
$ ykman piv access change-management-key --algorithm aes128 --protect --touch
```

Internally, the management key is required by the PIV application anytime the app’s settings are changed (like to generate a new key, or import certificates or other data into the app’s storage). Unlike the PIN or PUK, however, the number of consecutive wrong attempts to use it are not capped — so the key should be long and strong.

In lieu of having to enter in a long random string of characters every time you need to change a setting on the PIV app, YubiKeys have a feature that allow you to store a long, strong management key in the PIV app itself, and protect the key with the PIV application’s regular PIN (plus a YubiKey touch). The ykman CLI and the YKCS11 library will automatically detect when this feature is being used, and transparently request for and use the PIN to access the management key when needed (so you never need to know or enter the management key itself).

The `--protect` option in the above command directs the YubiKey PIV app to use this feature when setting up the new management key (and to automatically generate a new strong random key). The `--touch` option directs it to also save the key in a way that requires a touch to for each access.

Press enter when prompted, then enter the PIN (the new one you just set via the `piv access change-pin` command above):

```
Enter the current management key [blank to use default key]:
Enter PIN:
```

Now when you run the following command, the output should show that you’re no longer using the default PIN or management key for the PIV application:

```
$ ykman piv info
PIV version:              5.4.3
PIN tries remaining:      9/9
Management key algorithm: 8
Management key is stored on the YubiKey, protected by PIN.
CHUID: No data available
CCC:   No data available
```

PIV KEYS

Now you’re ready to generate a new set of PIV keys for the YubiKey. There are 25 separate key slots (numbered with hex digits, like `f9` or `9a` etc) available on your

YubiKey; 5 with a prescribed definition, and 20 miscellaneous slots:

Table 1. PIV Key Slots

Slot	Label	Purpose
f9	PIV Attestation	Equivalent of PGP “cert” key
9a	PIV Authentication	Equivalent of PGP “auth” key
9c	Digital Signature	Equivalent of PGP “sign” key
9d	Key Management	Equivalent of PGP “encr” key
9e	Card Authentication	Physical authentication of card holder
82-95	Retired Key 1-20	Retired decryption keys

The `f9` slot (PIV Attestation) comes preloaded with a key and certificate generated and signed by Yubico. The PIV application automatically uses this key to sign the other PIV keys you generate on your YubiKey. It’s the only slot that isn’t wiped if you reset your PIV application, although it can be overwritten with a different key if you like. Unless you have a specific reason to change it, you should leave it as is (and not use it for anything other than key certification).

The `9e` slot (Card Authentication) is intended to be used to unlock doors or other devices that don’t have a PIN pad (but if you don’t use your YubiKey for physical authentication, you can use it for whatever you want). We’ll skip it.

For example purposes, this guide will show you how to set up the `9a` slot (PIV Authentication) with an SSH key, the `9c` slot (Digital Signature) as signing key with a certificate signed by another CA, the `9d` slot (Key Management) for key agreement (ie deriving a shared encryption/decryption key with another party), and the `82` slot (Retired Key 1) as a simple root CA.

NOTE

Despite having looser or firmer definitions for all 25 slots, the PIV application is actually equally capable of using each slot as a decryption or signing (or signature-based authentication) key. For any client application you have that supports PKCS #11 and the YubiKey’s available key algorithms (RSA-1024, RSA-2048, P-256, or P-384), you can choose any of the 25 slots to use with it — even if it doesn’t match the slot’s intended purpose.

Since you’re going to be saving a few public keys and other temporary files to disk for the next steps, for your convenience, create a new directory and change to it:

```
$ mkdir ykpivfiles && cd ykpivfiles
```

Now generate your first PIV key, entering your PIV PIN and touching the YubiKey when prompted:

```
$ ykman piv keys generate --algorithm eccp384 --pin-policy once --touch-polic
Enter PIN:
Touch your YubiKey...
```

This will generate a new P-384 key in the PIV app's `9a` slot (PIV Authentication). When using this private key in the future, you'll have to provide the PIV PIN once per PKCS #11 client session, as well as touch your YubiKey once per use (if you hadn't already touched it in the previous 15 seconds).

The new public key will be saved to the file `key1-9a.pub`:

```
$ cat key1-9a.pub
-----BEGIN PUBLIC KEY-----
MHYwEAYHKoZIzj0CAQYFK4EEACIDYgAE0hUUs8J8bWeVQvpELjcUJUDsOudV4HP0
3hfqf2wb6Pm8th3MIVdSBNx+mc8IiQwjTK0IJYc1BbO68ClR87jAGyM0fGeUZkYZ
nKfuS9DOCevRTvaf9xXD+S+rjmQVoH7O
-----END PUBLIC KEY-----
```

We'll use this file in the next step, but you don't have to save it permanently — you can always ask the PIV application to export a new copy of the public key for you later. For example, the following command will export it to stdout:

```
$ ykman piv keys export 9a -
-----BEGIN PUBLIC KEY-----
MHYwEAYHKoZIzj0CAQYFK4EEACIDYgAE0hUUs8J8bWeVQvpELjcUJUDsOudV4HP0
3hfqf2wb6Pm8th3MIVdSBNx+mc8IiQwjTK0IJYc1BbO68ClR87jAGyM0fGeUZkYZ
nKfuS9DOCevRTvaf9xXD+S+rjmQVoH7O
-----END PUBLIC KEY-----
```

Next, we'll generate a self-signed certificate for the new key. You don't necessarily need a certificate for every key, but some PKCS #11 clients may require it, so it's nice to have. You can always replace it later with a different certificate (self-signed by the same key, or signed by some other CA).

To generate a new self-signed certificate, run the following command, entering your PIV PIN and touching the YubiKey when prompted:

```
$ ykman piv certificates generate --subject 'CN=Justin (key1-9a) <justin@exam
Enter PIN:
Touch your YubiKey...
```

You'll need to specify a subject DN (Distinguished Name) for the certificate via the `--subject` option. If you don't have a particular DN format required for your use of the key, you can just specify a CN (Common Name) component with whatever text will help you (or anyone else who views the certificate) disambiguate it from other certificates you may use. Using the same format as an OpenPGP UID (eg `real name (comment) <email address>`) can be a good starting point. You may also want to note the YubiKey and PIV slot in which the key can be found (like the `(key1-9a)` text from the example above).

You should also specify a validity time period for the certificate via the `--valid-days` option (the default is 365 days). The key will still be good after this period expires — but the certificate won't be. While short validity periods in general are a good security practice, they provide limited practical benefits for a self-signed certificate (since the key holder can simply generate a new self-signed certificate on demand). A validity period of 4000 days (almost 11 years) should last you the lifetime of the key.

Whenever you need to access this certificate, just run the following command to save it to a file:

```
$ ykman piv certificates export 9a key1-9a.crt
```

You can use OpenSSL to inspect the certificate details:

```
$ ykman piv certificates export 9a - | openssl x509 -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            14:f3:43:15:4e:8d:2b:95:41:41:c8:8e:f0:15:71:06:eb:84:57:a9
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: CN = "Justin (key1-9a) <justin@example.com>"
        Validity
            Not Before: Mar 28 21:39:54 2023 GMT
            Not After : Mar 10 21:39:54 2034 GMT
        Subject: CN = "Justin (key1-9a) <justin@example.com>"
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (384 bit)
                pub:
                    04:d2:15:14:b3:c2:7c:6d:67:95:42:fa:44:2e:37:
```

```
                    14:25:40:ec:3a:e7:55:e0:73:f4:de:17:ea:7f:6c:
                    1b:e8:f9:bc:b6:1d:cc:21:57:52:04:dc:7e:99:cf:
                    08:89:0c:23:4c:ad:08:25:87:35:05:b3:ba:f0:29:
                    51:f3:b8:c0:1b:23:34:7c:67:94:66:46:19:9c:a7:
                    ee:4b:d0:ce:09:eb:d1:4e:f6:9f:f7:15:c3:f9:2f:
                    ab:8e:64:15:a0:7e:ce
                ASN1 OID: secp384r1
                NIST CURVE: P-384
    Signature Algorithm: ecdsa-with-SHA256
    Signature Value:
        30:65:02:30:43:bc:33:58:65:4f:d4:15:47:86:3b:b5:7b:e6:
        55:84:6d:f6:dc:05:ee:c4:37:a6:6f:c4:e7:e4:30:dc:0d:90:
        c7:5c:6d:b2:7a:f2:dd:2d:af:a9:66:a3:d7:79:a3:12:02:31:
        00:b2:65:f8:81:10:94:c0:9b:a6:20:e5:7c:69:ee:4b:a6:ae:
        3b:27:b7:23:92:46:78:18:d4:9f:d4:81:ef:b0:5d:17:23:a0:
        1f:6e:2d:8f:ec:28:fe:8d:a7:28:ff:f3:a7
-----BEGIN CERTIFICATE-----
MIIBnTCCASOgAwIBAgIUFPNDFU6NK5VBQciO8BVxBuuEV6kwCgYIKoZIzj0EAwIw
MDEuMCwGA1UEAwwlSnVzdGluIChrZXkxLTlhKSA8anVzdGluQGV4YW1wbGUuY29t
PjAeFw0yMzAzMjgyMTM5NTRaFw0zNDAzMTAyMTM5NTRaMDAxLjAsBgNVBAMMJUp1
c3RpbiAoa2V5MS05YSkgPGp1c3RpbkBleGFtcGxlLmNvbT4wdjAQBgcqhkjOPQIB
BgUrgQQAIgNiAATSFRSzwnxtZ5VC+kQuNxQlQOw651Xgc/TeF+p/bBvo+by2Hcwh
V1IE3H6ZzwiJDCNMrQglhzUFs7rwKVHzuMAbIzR8Z5RmRhmcp+5L0M4J69FO9p/3
FcP5L6uOZBWgfs4wCgYIKoZIzj0EAwIDaAAwZQIwQ7wzWGVP1BVHhju1e+ZVhG32
3AXuxDemb8Tn5DDcDZDHXG2yevLdLa+pZqPXeaMSAjEAsmX4gRCUwJumIOV8ae5L
pq47J7cjkkZ4GNSf1IHvsF0XI6Afbi2P7Cj+jaco//On
-----END CERTIFICATE-----
```

Generate a key and certificate for any other PIV keys you want the same way, such as for slot 9c (Digital Signature):

```
$ ykman piv keys generate --algorithm eccp384 --pin-policy once --touch-polic
Enter PIN:
Touch your YubiKey...
$ ykman piv certificates generate --subject 'CN=Justin (key1-9c) <justin@exam
Enter PIN:
Touch your YubiKey...
```

And slot 9d (Key Management — encryption/decryption):

```
$ ykman piv keys generate --algorithm eccp384 --pin-policy once --touch-polic
Enter PIN:
Touch your YubiKey...
$ ykman piv certificates generate --subject 'CN=Justin (key1-9d) <justin@exam
```

```
Enter PIN:
Touch your YubiKey...
```

And slot 82 (Retired Key 1 — a miscellaneous key we'll use for an example CA):

```
$ ykman piv keys generate --algorithm eccp384 --pin-policy once --touch-polic
Enter PIN:
Touch your YubiKey...
$ ykman piv certificates generate --subject 'CN=Justin (key1-82) <justin@exam
Enter PIN:
Touch your YubiKey...
```

Now if you run the following command, you'll see the details listed for each of the 4 keys we just generated:

```
$ ykman piv info
PIV version:              5.4.3
PIN tries remaining:      9/9
Management key algorithm: 8
Management key is stored on the YubiKey, protected by PIN.
CHUID: 3019d4e739da739ced39ce739d836858210842108421c84210c3eb34103c7e8730f912
CCC:   No data available
Slot 82:
  Algorithm:   ECCP384
  Subject DN:  CN=Justin (key1-82) \<justin@example.com\>
  Issuer DN:   CN=Justin (key1-82) \<justin@example.com\>
  Serial:      267690053478081854363456881922522153929109828202
  Fingerprint: a2815042003b14b265001d46be2671b374f08360a602b81ebfab2c5ff404cf
  Not before:  2023-03-28T21:45:00
  Not after:   2034-03-10T21:45:00

Slot 9a:
  Algorithm:   ECCP384
  Subject DN:  CN=Justin (key1-9a) \<justin@example.com\>
  Issuer DN:   CN=Justin (key1-9a) \<justin@example.com\>
  Serial:      119604740273219646681371186984377340943623542697
  Fingerprint: 40f1a1830c8251640e9e3962c42c749fd193faf0f2c3e574f1405f9dbb81f7
  Not before:  2023-03-28T21:39:54
  Not after:   2034-03-10T21:39:54

Slot 9c:
  Algorithm:   ECCP384
  Subject DN:  CN=Justin (key1-9c) \<justin@example.com\>
  Issuer DN:   CN=Justin (key1-9c) \<justin@example.com\>
  Serial:      708868326279358429629319480715759510696124050124
```

```
    Fingerprint: 1568d64ab4512d560e3ed092c6c1f7f77f34c30eed05654136a9dae3baae7f
    Not before:  2023-03-28T21:44:28
    Not after:   2034-03-10T21:44:28

  Slot 9d:
    Algorithm:   ECCP384
    Subject DN:  CN=Justin (key1-9d) \<justin@example.com\>
    Issuer DN:   CN=Justin (key1-9d) \<justin@example.com\>
    Serial:      543818019060290952793661077378467821810601313771
    Fingerprint: a53f2898862615be970d80d17d6f4ff797afd9e57fcff4845bbddbc1771d93
    Not before:  2023-03-28T21:44:43
    Not after:   2034-03-10T21:44:43
```

PIV SSH

To use key 9a (PIV Authentication) that we generated above for SSH authentication, first run the following command to print out the available PIV keys on your YubiKey in SSH authorized-keys format:

```
$ ssh-keygen -D /usr/lib/x86_64-linux-gnu/libykcs11.so.2
ecdsa-sha2-nistp384 AAAAE2VjZHNhLXNoYTItbmlzdHAzODQAAAAIbmlzdHAzODQAAABhBNIVF
ecdsa-sha2-nistp384 AAAAE2VjZHNhLXNoYTItbmlzdHAzODQAAAAIbmlzdHAzODQAAABhBCzyC
ecdsa-sha2-nistp384 AAAAE2VjZHNhLXNoYTItbmlzdHAzODQAAAAIbmlzdHAzODQAAABhBBsDK
ecdsa-sha2-nistp384 AAAAE2VjZHNhLXNoYTItbmlzdHAzODQAAAAIbmlzdHAzODQAAABhBFnYz
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC/p7ppATZ75y1RwLiswoTEKwX2qixHNjc1Xfo1/
```

Replace `/usr/lib/x86_64-linux-gnu/libykcs11.so.2` with the path to your YKCS11 library (see the beginning of the PIV section) if different.

Find the line in the output with a comment containing “PIV Authentication” (the label for key 9a), and copy it to the `~/.ssh/authorized_keys` file on any host with which you want to use the key to authenticate (like `bastion.example.com`).

Then you can SSH into that server by running the following command:

```
$ ssh -I /usr/lib/x86_64-linux-gnu/libykcs11.so.2 me@bastion.example.com
```

Enter your PIN when prompted, and then touch the YubiKey when it starts blinking:

```
Enter PIN for 'YubiKey PIV #12345678':
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-1030-aws aarch64)
```

```
Last login: Tue Mar 28 23:15:35 2023 from 198.51.100.123
me@bastion:~$
```

CAUTION

This will offer all the PIV public keys on your YubiKey to the server — not just the keys you added to the server's authorized-keys file. (This is not a security concern, but it could be a privacy concern with a malicious server.)

PIV SIGNED CERTIFICATE

To get a certificate from a real CA (Certificate Authority) for key 9c (Digital Signature) that we generated above, first run the following command to generate a CSR (Certificate Signing Request):

```
$ ykman piv certificates request --subject 'CN=Justin,OU=IT,O=Example Co.,ST=
```

Set the `--subject` option to the appropriate subject DN for the certificate, and use the public key file `key1-9c.pub` you exported earlier (or re-export it if necessary). Enter your PIN, touch your YubiKey, and the command will save the CSR to the file `key1-9c.csr`:

```
Enter PIN:
Touch your YubiKey...
```

View the CSR with the following OpenSSL command:

```
$ openssl req -text -in key1-9c.csr
Certificate Request:
    Data:
        Version: 1 (0x0)
        Subject: C = US, ST = Washington, O = Example Co., OU = IT, CN = Just
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (384 bit)
                pub:
                    04:2c:f2:0a:76:03:7b:d2:34:38:35:d0:9a:e2:ef:
                    03:f9:e6:17:70:3d:a8:1d:51:fd:e5:a3:11:0f:cb:
                    22:3b:2a:86:b7:41:08:65:c3:35:7d:ba:0d:48:45:
                    95:07:15:57:67:6a:0e:50:b9:36:39:75:c0:d6:87:
                    03:90:07:cb:4e:44:08:4b:92:c6:db:ee:51:b0:f2:
                    cf:89:01:e5:0e:1c:35:14:8f:19:c7:94:a1:00:15:
                    7e:e2:fb:eb:c4:cd:af
                ASN1 OID: secp384r1
                NIST CURVE: P-384
        Attributes:
            (none)
            Requested Extensions:
    Signature Algorithm: ecdsa-with-SHA256
    Signature Value:
        30:66:02:31:00:e2:01:de:34:96:53:cf:a3:cf:b7:dc:33:4e:
        58:7b:2e:e0:01:c4:df:9b:06:97:ce:ca:bf:c2:79:26:6e:4b:
        77:43:d0:99:01:f6:48:35:52:cb:f6:fc:72:1c:33:15:45:02:
        31:00:86:c0:b3:77:46:9b:ff:1d:c6:28:27:3c:0b:e6:92:cf:
        26:16:32:71:37:e0:68:1d:35:ba:dc:97:59:7e:7a:06:38:94:
        c8:04:19:e4:10:be:3e:c1:d6:82:38:13:89:10
-----BEGIN CERTIFICATE REQUEST-----
MIIBTzCB1QIBADBWMQswCQYDVQQGEwJVUzETMBEGA1UECAwKV2FzaGluZ3RvbjEU
MBIGA1UECgwLRXhhbXBsZSBDby4xCzAJBgNVBAsMAklUMQ8wDQYDVQQDDAZKdXN0
aW4wdjAQBgcqhkjOPQIBBgUrgQQAIgNiAAQs8gp2A3vSNDg10Jri7wP55hdwPagd
Uf3loxEPyyI7Koa3QQhlwzV9ug1IRZUHFVdnag5QuTY5dcDWhwOQB8tORAhLksbb
7lGw8s+JAeUOHDUUjxnHlKEAFX7i++vEza+gADAKBggqhkjOPQQDAgNpADBmAjEA
4gHeNJZTz6PPt9wzTlh7LuABxN+bBpfOyr/CeSZuS3dD0JkB9kg1Usv2/HIcMxVF
AjEAhsCzd0ab/x3GKCc8C+aSzyYWMnE34GgdNbrcl1l+egY4lMgEGeQQvj7B1oI4
E4kQ
-----END CERTIFICATE REQUEST-----
```

TIP

If you need to add additional attributes or other extensions (such as `subjectAlternateName`) to the CSR, you'll have to do that using OpenSSL. Following is an example of of using the `openssl req` command to generate a CSR for key 9c:

```
$ PKCS11_MODULE_PATH=/usr/lib/x86_64-linux-gnu/libykcs11.so.2 \
    openssl req -new \
    -engine pkcs11 -keyform engine -key "pkcs11:object=Private key for Di
    -out key-9c.csr
```

Provide the CSR to the CA operator; and when the CA operator returns the signed certificate, import it by running the following command (where the certificate file is named `key1-9c.crt` in this example):

```
$ ykman piv certificates import --verify 9c key1-9c.crt
```

Use the `--verify` option when importing a certificate to verify that the certificate matches the key stored in slot 9c (so you don't accidentally import a certificate for some other key). Enter your PIN, touch your YubiKey, and the command will import the certificate (if it matches the key):

```
Enter PIN:
Touch your YubiKey...
```

The command will fail with an error message if the certificate doesn't match the key:

```
ERROR: This certificate is not tied to the private key in the SIGNATURE slot.
```

Your YubiKey will now provide the cert from the CA, instead of the self-signed certificate you generated originally, when queried:

```
$ ykman piv info
PIV version:              5.4.3
PIN tries remaining:      9/9
Management key algorithm: 8
Management key is stored on the YubiKey, protected by PIN.
CHUID: 3019d4e739da739ced39ce739d836858210842108421c84210c3eb3410ba09f7ff3812
CCC:   No data available
Slot 82:
  Algorithm:   ECCP384
  Subject DN:  CN=Justin (key1-82) \<justin@example.com\>
  Issuer DN:   CN=Justin (key1-82) \<justin@example.com\>
  Serial:      267690053478081854363456881922522153929109828202
  Fingerprint: a2815042003b14b265001d46be2671b374f08360a602b81ebfab2c5ff404cf
  Not before:  2023-03-28T21:45:00
  Not after:   2034-03-10T21:45:00

Slot 9a:
  Algorithm:   ECCP384
  Subject DN:  CN=Justin (key1-9a) \<justin@example.com\>
  Issuer DN:   CN=Justin (key1-9a) \<justin@example.com\>
  Serial:      119604740273219646681371186984377340943623542697
  Fingerprint: 40f1a1830c8251640e9e3962c42c749fd193faf0f2c3e574f1405f9dbb81f7
  Not before:  2023-03-28T21:39:54
  Not after:   2034-03-10T21:39:54

Slot 9c:
  Algorithm:   ECCP384
  Subject DN:  CN=Justin,OU=IT,O=Example Co.,ST=Washington,C=US
  Issuer DN:   CN=Example CA XYZ,OU=IT,O=Example Co.,ST=Washington,C=US
  Serial:      291
  Fingerprint: ba6803b4a95047bc15e573c205dc6401e6c5c4fe3cb72f9d6c386f4b0831d5
  Not before:  2023-03-29T01:46:18
```

```
    Not after:   2024-03-28T01:46:18

  Slot 9d:
    Algorithm:   ECCP384
    Subject DN:  CN=Justin (key1-9d) \<justin@example.com\>
    Issuer DN:   CN=Justin (key1-9d) \<justin@example.com\>
    Serial:      543818019060290952793661077378467821810601313771
    Fingerprint: a53f2898862615be970d80d17d6f4ff797afd9e57fcff4845bbddbc1771d93
    Not before:  2023-03-28T21:44:43
    Not after:   2034-03-10T21:44:43
```

PIV KEY AGREEMENT

If you want to use a PIV key like 9d (Key Management) for decryption, like to exchange messages with some other party, say your friend Alice, you'd first exchange public keys. You'd give her the `key1-9d.pub` public key file that you exported for your 9d key, and she'd give you her public key (it would have to be a public key of the same type as yours, for the P-384 curve).

She might create her key pair with the following OpenSSL commands:

```
$ openssl genpkey -algorithm ec -pkeyopt ec_paramgen_curve:secp384r1 -out ali
$ openssl pkey -pubout -in alice.key -out alice.pub
```

Her private key was saved to the `alice.key` file, and her public key to the `alice.pub` file. With the `key1-9d.pub` file containing your public key, she can derive a shared secret with her private key by running the following command:

```
$ openssl pkeyutl -derive -inkey alice.key -peerkey key1-9d.pub | xxd
00000000: 0c4c 06e8 1afb a9da 160f 8eff 8800 c43d  .L.............=
00000010: e642 04c9 381d 2d67 caf2 e058 9b50 b77f  .B..8.-g...X.P..
00000020: c8b5 dcd0 6176 a877 8ffa 5fbb 26a1 6495  ....av.w.._.&.d.
```

She can feed this shared secret into the `openssl enc` command to use it to encrypt a message that only you and she can read:

```
$ echo 'Hi Dude!' |
    openssl enc -chacha20 -pbkdf2 -a \
    -pass file:<(openssl pkeyutl -derive -inkey alice.key -peerkey key1-9d.pu
U2FsdGVkX1/f6Cna6360s7SrA6zo0OY0mQ==
```

If she sends you the output of this command (`U2FsdGVkX1/f6Cna6360s7SrA6zo0OY0mQ==`), you can decrypt it by deriving the same shared secret with the private key stored in the 9d slot of your YubiKey, plus her `alice.pub` public key file:

```
$ echo U2FsdGVkX1/f6Cna6360s7SrA6zo0OY0mQ== |
    openssl enc -d -chacha20 -pbkdf2 -a \
    -pass file:<(PKCS11_MODULE_PATH=/usr/lib/x86_64-linux-gnu/libykcs11.so.2
        openssl pkeyutl -derive \
        -engine pkcs11 -keyform engine -inkey "pkcs11:object=Private key for
        -peerkey alice.pub)
Engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiKey PIV #12345678:
```

NOTE

To run the above command, you need the PKCS #11 engine plugin for OpenSSL (plus have the YKCS11 library installed). See the next section (PIV CA) for more details about that.

Enter your PIV PIN at the prompt, touch your YubiKey, and you should see the message from Alice output as a result of the command:

```
Hi Dude!
```

WARNING

Don’t use the DH (Diffie-Hellman) “derive” primitive directly like this example, except for testing (because it’s too easy to screw up something that would allow an adversary to figure out your shared secret, or possibly even your private key). Instead, use higher-level abstractions, like the seal/open evp functionality of libcrypto for encrypting and decrypting messages/files, or libssl for encrypting and decrypting streams.

PIV CA

To use key 82 (Retired Key 1) for a root CA, you can set up a simple CA with OpenSSL. First, you have to install the PKCS #11 Engine Plugin for OpenSSL. You can build it from source, or install it from your package manager.

On a Debian and friends, you can install it with the `libengine-pkcs11-openssl` package; on a Fedora and friends, you can install it with the `openssl-pkcs11` package. The following command will output the engine’s name if installed:

```
$ openssl engine pkcs11 2>/dev/null
(pkcs11) pkcs11 engine
```

Make sure you've also installed the YKCS11 library (as described at the beginning of the PIV section).

You can test that the engine and the YKCS11 library have been installed correctly and can access your PIV keys by running the following commands to sign a test file:

```
$ echo test > test.txt
$ PKCS11_MODULE_PATH=/usr/lib/x86_64-linux-gnu/libykcs11.so.2 \
    openssl pkeyutl -sign \
    -engine pkcs11 -keyform engine -inkey "pkcs11:object=Private key for Reti
    -in test.txt -out test.sig
```

Replace `/usr/lib/x86_64-linux-gnu/libykcs11.so.2` with the actual path to your YKCS11 library if different. The `Private key for Retired Key 1` text in the command is the label for private key of slot 82. Many PKCS #11 clients, such as the OpenSSL plugin, refer to keys by label (aka key alias), rather than by slot number. You can look up the label for each PIV slot in the "Key Alias per Slot and Object Type" table of the YKCS11 Functions and values documentation.

The sign command should prompt you for your PIV PIN; enter it, then touch your YubiKey when it starts blinking:

```
Engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiKey PIV #12345678:
```

This command will output a `test.sig` file. You can verify the signature using the `key1-82.pub` file that was output when you generated the key earlier:

```
$ openssl pkeyutl -verify -pubin -inkey key1-82.pub -in test.txt -sigfile tes
Signature Verified Successfully
```

To generate a simple root CA cert with key 82, run the following command:

```
$ PKCS11_MODULE_PATH=/usr/lib/x86_64-linux-gnu/libykcs11.so.2 \
    openssl req -new -x509 \
    -engine pkcs11 -keyform engine -key "pkcs11:object=Private key for Retire
    -out key1-82.crt
```

Enter your PIN when prompted:

```
Engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiKey PIV #12345678:
```

Then enter the default components for the subject DN of the new cert when prompted:

```
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Washington
Locality Name (eg, city) []:Seattle
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Example Co.
Organizational Unit Name (eg, section) []:IT
Common Name (e.g. server FQDN or YOUR name) []:Example CA 123
Email Address []:
```

At the end of this process, your YubiKey will start to blink; touch it, and the new CA cert will be output to the file `key1-82.crt`.

Now import this new cert into the certificate slot for key 82 on your YubiKey with the following command:

```
$ ykman piv certificates import 82 key1-82.crt
```

Enter your PIN when prompted, then touch your YubiKey:

```
Enter PIN:
Touch your YubiKey...
```

You can export this certificate at any time from your YubiKey by running the following command:

```
$ ykman piv certificates export 82 - | openssl x509 -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            2d:de:77:8f:c6:c0:bf:cd:8a:ad:4c:87:0d:d2:bc:53:fa:bb:af:0e
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: C = US, ST = Washington, O = Example Co., OU = IT, CN = Examp
        Validity
            Not Before: Mar 29 20:12:36 2023 GMT
            Not After : Apr 28 20:12:36 2023 GMT
        Subject: C = US, ST = Washington, O = Example Co., OU = IT, CN = Exam
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (384 bit)
                pub:
                    04:59:d8:cd:7b:53:2e:e0:58:ff:3e:fa:10:9e:f8:
                    78:d2:c6:f7:e8:f4:aa:4f:12:ee:90:3f:60:09:e0:
                    4e:59:05:ed:5e:30:b7:59:f3:f1:db:30:9f:02:2e:
                    95:28:c4:80:28:89:b0:3c:7d:83:26:03:cc:21:83:
                    ea:62:2a:33:3a:a8:ad:7c:19:9d:26:1e:a1:dc:29:
                    96:0c:93:9b:2d:fb:b5:c5:56:a9:8c:5e:8a:ec:94:
                    42:25:46:74:b2:fd:70
                ASN1 OID: secp384r1
                NIST CURVE: P-384
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                97:44:2C:98:02:61:6D:CE:64:96:63:C0:30:B4:FF:36:7B:6F:AD:14
            X509v3 Authority Key Identifier:
                97:44:2C:98:02:61:6D:CE:64:96:63:C0:30:B4:FF:36:7B:6F:AD:14
            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: ecdsa-with-SHA256
    Signature Value:
        30:65:02:31:00:c3:d3:ec:ef:5a:47:3c:9c:d0:b8:45:56:da:
        a8:83:e8:0e:73:3a:49:d7:15:18:6e:0e:16:05:e7:3c:d6:28:
        cd:8f:d7:60:21:cd:04:b7:c3:4b:dd:4f:4c:6a:b4:2a:56:02:
        30:66:6a:18:36:de:9b:84:0e:ed:64:ff:33:b1:f3:a0:67:40:
        14:0d:51:f9:05:60:02:33:c3:e9:d2:68:88:26:68:7a:de:41:
        36:4d:cb:7f:60:22:85:39:30:48:48:c0:ee
-----BEGIN CERTIFICATE-----
MIICTjCCAdSgAwIBAgIULd53j8bAv82KrUyHDdK8U/q7rw4wCgYIKoZIzj0EAwIw
XjELMAkGA1UEBhMCVVMxEzARBgNVBAgMCldhc2hpbmd0b24xFDASBgNVBAoMC0V4
YW1wbGUgQ28uMQswCQYDVQQLDAJJVDEXMBUGA1UEAwwORXhhbXBsZSBDQSAxMjMw
HhcNMjMwMzI5MjAxMjM2WhcNMjMwNDI4MjAxMjM2WjBeMQswCQYDVQQGEwJVUzET
MBEGA1UECAwKV2FzaGluZ3RvbjEUMBIGA1UECgwLRXhhbXBsZSBDby4xCzAJBgNV
BAsMAklUMRcwFQYDVQQDDA5FeGFtcGxlIENBIDEyMzB2MBAGByqGSM49AgEGBSuB
BAAiA2IABFnYzXtTLuBY/z76EJ74eNLG9+j0qk8S7pA/YAngTlkF7V4wt1nz8dsw
nwIulSjEgCiJsDx9gyYDzCGD6mIqMzqorXwZnSYeodwplgyTmy37tcVWqYxeiuyU
```

```
QiVGdLL9cKNTMFEwHQYDVR0OBBYEFJdELJgCYW3OZJZjwDC0/zZ7b60UMB8GA1Ud
IwQYMBaAFJdELJgCYW3OZJZjwDC0/zZ7b60UMA8GA1UdEwEB/wQFMAMBAf8wCgYI
KoZIzj0EAwIDaAAwZQIxAMPT7O9aRzyc0LhFVtqog+gOczpJ1xUYbg4WBec81ijN
j9dgIc0Et8NL3U9MarQqVgIwZmoYNt6bhA7tZP8zsfOgZ0AUDVH5BWACM8Pp0miI
Jmh63kE2Tct/YCKFOTBISMDu
-----END CERTIFICATE-----
```

To sign someone else’s key with this CA, first have the holder of the other key generate a CSR with their key and provide it to you. For example, say your co-worker Bob generates a new private key and CSR with the following commands:

```
$ openssl genpkey -algorithm ed25519 -out bob.key
$ openssl req -new -key bob.key -out bob.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Washington
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Example Co.
Organizational Unit Name (eg, section) []:IT
Common Name (e.g. server FQDN or YOUR name) []:Bob
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

He’d provide the output file, `bob.csr` to you; from which you now can produce a signed certificate with your CA.

With a real CA, you’d have a custom OpenSSL config file and directory structure set up; but for this simple example, we’ll use OpenSSL’s default “demoCA” config and directory structure. Create the following directories and files:

```
$ mkdir demoCA && touch demoCA/index.txt && echo 01 > demoCA/serial
```

Then you can sign Bob’s CSR with the following command:

```
$ PKCS11_MODULE_PATH=/usr/lib/x86_64-linux-gnu/libykcs11.so.2 \
    openssl ca \
    -engine pkcs11 -keyform engine -keyfile "pkcs11:object=Private key for Re
    -cert key1-82.crt -in bob.csr -outdir .
```

Enter the PIV PIN when prompted:

```
Engine "pkcs11" set.
Using configuration from /usr/lib/ssl/openssl.cnf
Enter PKCS#11 token PIN for YubiKey PIV #12345678:
```

OpenSSL will print the details of Bob's CSR for you to review:

```
Check that the request matches the signature
Signature ok
Certificate Details:
        Serial Number: 1 (0x1)
        Validity
            Not Before: Mar 29 20:56:42 2023 GMT
            Not After : Mar 28 20:56:42 2024 GMT
        Subject:
            countryName               = US
            stateOrProvinceName       = Washington
            organizationName          = Example Co.
            organizationalUnitName    = IT
            commonName                = Bob
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 Subject Key Identifier:
                0F:FD:F3:44:43:05:16:C1:79:3C:73:DD:97:FB:7B:C8:07:6C:81:6F
            X509v3 Authority Key Identifier:
                97:44:2C:98:02:61:6D:CE:64:96:63:C0:30:B4:FF:36:7B:6F:AD:14
Certificate is to be certified until Mar 28 20:56:42 2024 GMT (365 days)
Sign the certificate? [y/n]:
```

If that looks OK, enter `y` at the prompt; then your YubiKey will start blinking — touch it to sign the new certificate. OpenSSL will then prompt you to update the "database" files in the `demoCA` folder; enter `y` again:

```
1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=US, ST=Washington, O=Example Co., OU=IT, CN=Example CA 123
        Validity
            Not Before: Mar 29 20:56:42 2023 GMT
            Not After : Mar 28 20:56:42 2024 GMT
        Subject: C=US, ST=Washington, O=Example Co., OU=IT, CN=Bob
        Subject Public Key Info:
            Public Key Algorithm: ED25519
                ED25519 Public-Key:
                pub:
                    7a:c6:42:6a:62:9c:d7:f6:36:5e:7e:d9:7d:46:64:
                    ad:29:9c:44:3b:35:c8:da:cf:9e:eb:e5:12:7a:ba:
                    bd:6b
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 Subject Key Identifier:
                0F:FD:F3:44:43:05:16:C1:79:3C:73:DD:97:FB:7B:C8:07:6C:81:6F
            X509v3 Authority Key Identifier:
                97:44:2C:98:02:61:6D:CE:64:96:63:C0:30:B4:FF:36:7B:6F:AD:14
    Signature Algorithm: ecdsa-with-SHA256
    Signature Value:
        30:66:02:31:00:82:31:46:2f:05:a4:59:40:37:8d:bd:43:bc:
        42:59:17:df:d4:a2:9e:2f:48:d3:2d:e7:69:25:f9:c1:40:91:
        e4:57:26:6c:b2:a9:4e:98:58:0e:3d:78:91:cf:1c:0e:62:02:
        31:00:e6:fc:aa:93:90:0b:ca:ed:ce:f2:17:63:09:68:e0:da:
        92:3d:68:3a:c6:ce:25:e1:e5:ac:e0:30:d0:75:5a:a1:d9:6c:
        f0:93:53:55:94:5e:01:0d:59:ec:3a:06:4b:7c
-----BEGIN CERTIFICATE-----
MIIB3zCCAWSgAwIBAgIBATAKBggqhkjOPQQDAjBeMQswCQYDVQQGEwJVUzETMBEG
A1UECAwKV2FzaGluZ3RvbjEUMBIGA1UECgwLRXhhbXBsZSBDby4xCzAJBgNVBAsM
AklUMRcwFQYDVQQDDA5FeGFtcGxlIENBIDEyMzAeFw0yMzAzMjkyMDU2NDJaFw0y
NDAzMjgyMDU2NDJaMFMxCzAJBgNVBAYTAlVTMRMwEQYDVQQIDApXYXNoaW5ndG9u
MRQwEgYDVQQKDAtFeGFtcGxlIENvLjELMAkGA1UECwwCSVQxDDAKBgNVBAMMA0Jv
YjAqMAUGAytlcAMhAHrGQmpinNf2Nl5+2X1GZK0pnEQ7Ncjaz57r5RJ6ur1ro00w
SzAJBgNVHRMEAjAAMB0GA1UdDgQWBBQP/fNEQwUWwXk8c92X+3vIB2yBbzAfBgNV
HSMEGDAWgBSXRCyYAmFtzmSWY8AwtP82e2+tFDAKBggqhkjOPQQDAgNpADBmAjEA
gjFGLwWkWUA3jb1DvEJZF9/Uop4vSNMt52kl+cFAkeRXJmyyqU6YWA49eJHPHA5i
AjEA5vyqk5ALyu3O8hdjCWjg2pI9aDrGziXh5azgMNB1WqHZbPCTU1WUXgENWew6
Bkt8
-----END CERTIFICATE-----
Data Base Updated
```

The new certificate will be saved as with a file name matching its serial number — in this case, `01.pem`. Give this file back to Bob. You (or anyone else with your public CA cert, `key1-82.crt`) can verify that it's been signed by your private key by running the following command:

```
$ openssl verify -CAfile key1-82.crt 01.pem
01.pem: OK
```

TIP

See Jamie Nguyen's excellent OpenSSL Certificate Authority guide for a much more thorough tutorial for setting up a CA with OpenSSL. Just replace the argument for the CA private key in the commands which use it in that guide with a reference to the PKCS #11 engine key as shown in the above examples. (Note that the argument name for the private key is not consistent across OpenSSL commands — sometimes it's `-key` , sometimes it's `-inkey` , sometimes it's `-keyfile` , and so on.)

FIDO

The YubiKey FIDO U2F and FIDO2 applications share a common management and user interface, so as an end-user of FIDO, you don't need to worry about the difference between the two. You can think of U2F as "FIDO 1.0" and FIDO2 as "FIDO 2.0", where YubiKeys are able to transparently support clients that can take advantage of the new features in FIDO2, but are still backwards compatible with older clients that only support the older U2F feature-set. ("FIDO" originally was an acronym for Fast IDentity Online; now it's the umbrella name for a suite of standards related to enabling signature-based authentication across a variety of different applications and devices.)

WebAuthn, the web browser API that uses FIDO credentials, works with both U2F and FIDO2. The "passwordless" feature of WebAuthn, known as Passkeys, requires FIDO2, however. On modern browsers running on modern operating systems (released within the last 2 years or so), you shouldn't have to install anything special to use your YubiKey with WebAuthn — it should just work "out of the box" (but see the WebAuthn support matrices by Yubico and Okta for a few platform-specific notes).

FIDO KEYS

The YubiKey FIDO application allows you to generate a bunch of signing key-pairs, where the private key of the signing key-pair is accessible only to the YubiKey (each key pair, plus the metadata about it, comprise a FIDO credential). Each individual key-pair is intended to be used for one single purpose in coordination with one other single party (that party is called the "relying party", aka RP). You use the private key of the key pair; the relying party uses the public key.

These key pairs come in two varieties: one variety is stored on the YubiKey itself, called "resident credentials" (aka "discoverable credentials"); and the other variety (non-resident credentials) is stored outside of the YubiKey, by the relying party. Even though the relying party stores the private key of non-resident credentials, it does not have access to use the private key — the private key is always exported in encrypted form, encrypted by a secret master key held by the YubiKey itself. Therefore, only the YubiKey can use the credential's private key (and only when the credential is remembered and supplied by the relying party).

U2F ("FIDO 1.0") supported only non-resident credentials; whereas FIDO2 supports both resident and non-resident credentials. Your YubiKey can store up to 25 resident credentials — but it can generate an infinite number of non-resident credentials. Resident credentials are primarily intended for use as WebAuthn Passkeys (where the FIDO credential is used as the primary, and often only, factor), as this allows a website to avoid publicly leaking the Passkeys it has stored.

With WebAuthn, non-resident credentials are generally expected to be used only for 2FA, where a website first requires a user to authenticate by some other means (like a password); and only if the first factor is successful will the website supply the user's stored FIDO credentials to allow the user to attempt 2FA with them. However, with other applications like SSH or PAM, where enumerating users by username is not possible or not a concern, non-resident credentials are fine for use as the primary authentication factor.

FIDO PIN

You should set the PIN for the FIDO application, which will allow you to list and delete resident credentials (important since the YubiKey can hold only 25 of them). Before setting your PIN, the ykman CLI will display the following when you query the state of the FIDO app:

```
$ ykman fido info
PIN is not set.
$ ykman fido credentials list
ERROR: Credential Management requires having a PIN. Set a PIN first.
```

NOTE

Once you set a FIDO PIN, some websites using WebAuthn may require you to enter that PIN in order to log in with a WebAuthn credential (while some may not). If required, you will be prompted for your FIDO PIN in the same browser dialog box that directs you to touch your YubiKey in order to authenticate with the site (however, don't ever enter your PIN directly into the website's own authentication form — your PIN is a secret used by the YubiKey itself, and never provided to the website directly).

In general, the PIN protects the FIDO app from listing, creating, and deleting resident credentials — but not from using the credentials themselves. However, for specific credentials or authentication attempts, a website (the credential's relying party) may require you to supply the PIN to your YubiKey in order to use the credential (via a permanent flag embedded in the credential itself, or via a temporary flag requested for an individual authentication attempt), for the purpose of user verification.

Most WebAuthn best-practice guides will recommend that websites require your PIN to be supplied when using WebAuthn as the only authentication factor, but not when using WebAuthn as a second authentication factor. However, it is ultimately up to the website to decide whether or not to require your PIN.

Run the following command to set your FIDO PIN:

```
$ ykman fido access change-pin
```

The PIN must be between 4 and 64 characters. Don't use a number — instead use a simple passphrase (in the "correct horse battery staple" vein) that's at least 4 characters long and easy to type. Make sure it's different than any other PIN or passphrase you've ever used before (including the OpenPGP or PIV PINs you set for this YubiKey). See the tip in the OpenPGP section for how to generate a good, random PIN.

WARNING

Similar to OpenPGP or PIV PINs, if you enter the wrong FIDO PIN too many times in a row, the YubiKey will block you from making any more attempts (unlike OpenPGP or PIV, this limit is not configurable — it's hardcoded to block after 8 wrong PIN attempts in a row). If that happens, you will not be able to list or delete your FIDO resident credentials anymore — and you will not be able to use any FIDO credentials that require your PIN to be supplied.

Enter your new PIN when prompted, and then repeat it:

```
Enter your new PIN:
Repeat for confirmation:
```

Write down your FIDO PIN on the same piece of paper you wrote down your OpenPGP and PIV PINs (be sure to note that it's for the YubiKey's FIDO application, as each application uses an independent set of PINs or passwords).

Once you've set your FIDO PIN, the ykman CLI will display this when you query the state of the FIDO app:

```
$ ykman fido info
PIN is set, with 8 attempt(s) remaining.
$ ykman fido credentials list
Enter your PIN:
Credential ID  RP ID  Username  Display name
```

FIDO SSH

FIDO can be used to generate SSH authentication keys. You can in fact generate an infinite number of non-resident FIDO SSH keys, one for each host to which you connect via SSH (or you can generate just a single SSH key, and use it for all hosts).

You need to be running at least OpenSSH 8.2 on both client and server in order to use FIDO SSH keys. Run the following command to check your SSH client version:

```
$ ssh -V
OpenSSH_8.9p1 Ubuntu-3ubuntu0.1, OpenSSL 3.0.2 15 Mar 2022
```

Run the following `ssh-keygen` command to generate a non-resident key that your SSH client will present by default to all SSH servers:

```
$ ssh-keygen -t ed25519-sk
Generating public/private ed25519-sk key pair.
You may need to touch your authenticator to authorize key generation.
```

Enter your FIDO PIN when prompted, then touch your YubiKey:

```
Enter PIN for authenticator:
You may need to touch your authenticator (again) to authorize key generation.
```

Like your other SSH keys, enter a strong, unique passphrase for it (use your password manager to generate and store a random password for the SSH key):

```
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/justin/.ssh/id_ed25519_sk
Your public key has been saved in /home/justin/.ssh/id_ed25519_sk.pub
```

```
The key fingerprint is:
SHA256:61lJwqyM7gVkGjshgI1Ye2AM+wT4VjzvWE8xQ4SRlh4 justin@mylaptop
```

Copy the contents of the newly generated `id_ed25519_sk.pub` file:

```
$ cat ~/.ssh/id_ed25519_sk.pub
sk-ssh-ed25519@openssh.com AAAAGnNrLXNzaC1lZDI1NTE5QG9wZW5zc2guY29tAAAAIF+8W+
```

And paste it in as a new line into the `~/.ssh/authorized_keys` file on every remote host with which you want to use this SSH key (or use the `ssh-copy-id` command as a shortcut for every remote host to which you already have SSH access).

When you SSH into a host (for example, `bastion.example.com`) using this new SSH key, you'll be prompted to enter the passphrase for the SSH key:

```
$ ssh bastion.example.com
Enter passphrase for key '/home/justin/.ssh/id_ed25519_sk':
```

And then you'll be prompted to touch your YubiKey:

```
Confirm user presence for key ED25519-SK SHA256:61lJwqyM7gVkGjshgI1Ye2AM+wT4V
[YubiKey touched]
User presence confirmed
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-1030-aws aarch64)

Last login: Tue Mar 29 01:12:03 2023 from 198.51.100.123
justin@bastion:~$
```

If you're using an SSH agent to cache your SSH credentials, you won't have to enter the SSH key's passphrase on subsequent log-ins (for as long as the key is cached) — you'll just have to touch your YubiKey when it blinks:

```
$ ssh bastion.example.com
[YubiKey touched]
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-1030-aws aarch64)

Last login: Tue Mar 29 01:13:25 2023 from 198.51.100.123
justin@bastion:~$
```

To generate additional SSH keys, specify a custom private-key file (and optionally a custom comment to store in the public-key file). For example, I might run the following command to generate an SSH key specifically for the production SSH bastion host:

```
$ ssh-keygen -t ed25519-sk -f ~/.ssh/bastion_key1 -C 'justin on key1 for prod
Generating public/private ed25519-sk key pair.
You may need to touch your authenticator to authorize key generation.
Enter PIN for authenticator:
[YubiKey touched]
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/justin/.ssh/bastion_key1
Your public key has been saved in /home/justin/.ssh/bastion_key1.pub
The key fingerprint is:
SHA256:IwZCagv9RzaAlRX8n8yL39TTF2/7dhYeleLM1wGJ2Dw justin on key1 for prod ba
```

The `-f` option specifies the path at which to save the new private-key file (a `.pub` extension will be appended to this path for the public-key file), and the `-C` option specifies a new comment to store in the public-key file.

Copy the content of the `~/.ssh/bastion_key1.pub` file, and paste it into the `~/.ssh/authorized_keys` file on the host. Then you can SSH into the host by specifying the path to the private-key file via the `-i` option:

```
$ ssh -i ~/.ssh/bastion_key1 bastion.example.com
Enter passphrase for key '/home/justin/.ssh/bastion_key1':
Confirm user presence for key ED25519-SK SHA256:IwZCagv9RzaAlRX8n8yL39TTF2/7d
[YubiKey touched]
User presence confirmed
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-1030-aws aarch64)

Last login: Tue Mar 29 01:18:57 2023 from 198.51.100.123
justin@bastion:~$
```

TIP

To avoid having to specify the `-i` flag for a custom private-key file every time you SSH into a specific host, use the `IdentityFile` option in your `~/.ssh/config` file, under the entry for the appropriate host or hosts:

```
# ~/.ssh/config
Host bastion.example.com
    IdentitiesOnly yes
    IdentityFile ~/.ssh/bastion_key1
    IdentityFile ~/.ssh/bastion_key2
```

FIDO PAM

FIDO can be used to log into your local computer, or to run `sudo` commands on your local computer (or generally anything that uses PAM for authentication on your local computer) — either as a second factor, or as the only factor. You need to have the pam-u2f library installed on your computer in order to use FIDO PAM keys.

On Debian and friends, install the `libpam-u2f` and `pamu2fcfg` packages:

```
$ sudo apt install libpam-u2f pamu2fcfg
```

Or on Fedora and friends, install the `pam-u2f` and `pamu2fcfg` packages:

```
$ sudo dnf install pam-u2f pamu2fcfg
```

Then run the following `pamu2fcfg` command to generate a non-resident key that can be used for PAM:

```
$ pamu2fcfg --type eddsa --origin pam://mylaptop; echo
```

The `--type` flag specifies the key type and algorithm (ie EdDSA vs conventional ECDSA vs RSA), and the `--origin` flag specifies the name of the host on which the key can be used. You don't need to use the literal hostname of the computer in the origin string, but it must begin with the prefix `pam://`.

NOTE

EdDSA (aka Ed25519) keys require version 1.2 or newer of the pam-u2f library. Check the version installed on your computer by running the following command:

```
$ pamu2fcfg --version
pam_u2f 1.3.0
```

If you have a version older than 1.2, use ECDSA with curve P-256 instead of EdDSA:

```
$ pamu2fcfg --type es256 --origin pam://mylaptop; echo
```

Enter the FIDO PIN when prompted, then touch your YubiKey:

```
Enter PIN for /dev/hidraw6:
```

```
justin:qTUtCkST3kmnn59wQkdMLFdwq5jkqD0iXz8zNRlTt7RrcKerJAy+uXJyTaibpnG0rMxxvN
```

Create a new `/etc/u2f_mappings` file as root on your local computer, and copy and paste the output of the above command (beginning with your username) into the file.

NOTE

If you generate multiple FIDO keys (so you can, for example, use several different YubiKeys to authenticate on your local computer), combine all the keys into one line, separated by colons (with your username as the first entry on the line, and not repeated anywhere else).

For example, to enable 3 different FIDO keys to be used for PAM authentication, I'd combine them all into one line like the following:

```
# /etc/u2f_mappings
justin:qTUtCkST3kmnn59wQkdMLFdwq5jkqD0iXz8zNRlTt7RrcKerJAy+uXJyTaibpnG0rMx
```

For illustration purposes, if I replaced the all but the first three characters of each big base64-encoded string in the above with ellipses, the line would look like the following:

```
justin:qTU...,YtE...,eddsa,+presence:Kxm...,JA6...,es256,+presence:*,KZp.
```

And if I then replaced each colon with a newline, it would look like the following:

```
justin
qTU...,YtE...,eddsa,+presence
Kxm...,JA6...,es256,+presence
*,KZp...,es256,+presence
```

(The third key in the above example happens to be from a resident credential, which is why it uses `*` as the first section of its configuration, instead of a big base64-encoded key handle like the non-resident keys do.)

Next, edit the `/etc/pam.d/sudo` file as root. Add the following line to the top of it (but keep the rest of the file intact!):

```
# /etc/pam.d/sudo
auth sufficient pam_u2f.so authfile=/etc/u2f_mappings origin=pam://mylaptop d
```

Change the `origin` flag in the above example to match exactly the origin string you used when you generated your FIDO PAM keys (`pam://mylaptop` in this example).

IMPORTANT

The origin you used when you generated a key with the `pamu2fcfg` command **must** match the origin you use in the PAM config files.

If you don't specify an origin, PAM will use the current hostname by default. The problem with that is, if the hostname ever changes, your key will stop working.

Save the file — but don't quit your editor yet! Test it by running `sudo` in a new terminal, like the following:

```
$ sudo -k echo make me a sandwich
```

If all goes well, you should see a bunch of debug output sent to your terminal, and your YubiKey should start blinking:

```
debug(pam_u2f): pam-u2f.c:111 (parse_cfg): called.
debug(pam_u2f): pam-u2f.c:112 (parse_cfg): flags 32768 argc 3
debug(pam_u2f): pam-u2f.c:114 (parse_cfg): argv[0]=authfile=/etc/u2f_mappings
debug(pam_u2f): pam-u2f.c:114 (parse_cfg): argv[1]=origin=pam://mylaptop
debug(pam_u2f): pam-u2f.c:114 (parse_cfg): argv[2]=debug
...
adjust_assert_count: cbor_type
cbor_decode_assert_authdata: buf=0x5625a2af9bf0, len=37
debug(pam_u2f): util.c:1106 (get_authenticators): Found key in authenticator
fido_tx: dev=0x5625a2ae9f70, cmd=0x10
fido_tx: buf=0x5625a2ae9a00, len=145
0000: 02 a4 01 6c 70 61 6d 3a 2f 2f 6d 79 68 6f 73 74
0016: 02 58 20 77 f0 0e 7f ec e2 d8 25 49 14 32 cd 4d
0032: 34 ee 5b 7e 34 2f 55 de 2a c9 e8 34 6c dc 4d c9
0048: 2d 34 4e 03 81 a2 62 69 64 58 40 2b 19 a4 e3 35
0064: 93 ae 7e 26 26 20 58 2f 77 03 dd f1 7c 3a 52 b2
0080: b4 97 cf 27 8c 77 82 5d 1e cb 3d 32 9f 6c 32 a6
0096: 3a cd 74 86 d7 9e 73 0f 47 ad 2b 4d c8 ea 98 fc
0112: 00 75 f7 78 ea c0 1b 1d ce a3 47 64 74 79 70 65
0128: 6a 70 75 62 6c 69 63 2d 6b 65 79 05 a1 62 75 70
0144: f5
```

Touch your YubiKey, watch more debug output scroll by, and you should at the end of the process see your test command execute successfully:

```
fido_rx: dev=0x5625a2ae9f70, cmd=0x10, ms=-1
rx_preamble: buf=0x7ffe162cc480, len=64
0000: 63 5e 2e 3a 90 00 cb 00 a3 01 a2 62 69 64 58 40
0016: 2b 19 a4 e3 35 93 ae 7e 26 26 20 58 2f 77 03 dd
0032: f1 7c 3a 52 b2 b4 97 cf 27 8c 77 82 5d 1e cb 3d
0048: 32 9f 6c 32 a6 3a cd 74 86 d7 9e 73 0f 47 ad 2b
rx: payload_len=203
rx: buf=0x7ffe162cc480, len=64
...
fido_check_flags: flags=01
fido_check_flags: up=2, uv=0
debug(pam_u2f): pam-u2f.c:473 (pam_sm_authenticate): done. [Success]
make me a sandwich
```

If the test command was successful, remove the `debug` flag from the end of the line in your `/etc/pam.d/sudo` file:

```
# /etc/pam.d/sudo
auth sufficient pam_u2f.so authfile=/etc/u2f_mappings origin=pam://mylaptop
```

The above PAM control value `sufficient` allows your YubiKey to act as an optional primary factor for sudo authentication. If you fail to touch your YubiKey (or if it's unplugged), you can still use your user account password for sudo authentication — and if you do touch your YubiKey, you won't have to enter your password.

To configure sudo to use your YubiKey as a mandatory second factor, change the `sufficient` control value to `requisite`:

```
# /etc/pam.d/sudo
auth requisite pam_u2f.so authfile=/etc/u2f_mappings origin=pam://mylaptop
```

Now if you run a test sudo command, like the following:

```
$ sudo -k echo make me a sandwich
```

First your YubiKey will blink, and the command will pause until you touch it; then you'll be prompted for your user account password:

```
[sudo] password for justin:
```

And then, once you enter your password, the test command will run:

```
make me a sandwich
```

If you don’t touch your YubiKey, or you don’t have it plugged in, you won’t be able to run any sudo commands.

You can apply the same configuration line to the various other files in your `/etc/pam.d` sudo directory to enable the use of your YubiKey as a second factor (or primary factor) for various other services (such as to log into your desktop environment).

TIP

If you screw up your PAM configuration (or it stops working after a system update), and you can no longer sudo or log into your local computer, you can always boot from a recovery USB stick, mount your root drive to some mount point on your recovery system, and edit the `/etc/pam.d` files from that mount point to update or remove whatever FIDO settings you had added (and then reboot and log in as normal).

NOTE

Fedora and derivatives use Authselect to manage PAM configuration. If you use a distro with Authselect, you can use it directly to set up YubiKey PAM integration; see the Use FIDO U2F security keys with Fedora Linux article for details.

FIDO KEY DELETION

You can delete any of the resident credentials stored on your YubiKey by first running the following command to look up the first few hex digits of the credential’s ID:

```
$ ykman fido credentials list
Enter your PIN:
Credential ID  RP ID         Username    Display name
87422920...    webauthn.me   test        Test User
53badd61...    webauthn.io   Justin Two  Justin Two
8b17e0e4...    webauthn.io   Justin One  Justin One
```

Specify the first few digits of the credential’s ID as the last argument to the following command to delete it:

```
$ ykman fido credentials delete 8b17e0e4
```

Enter your FIDO PIN, and then `y` to confirm:

```
Enter your PIN:
Delete webauthn.io Justin One Justin One (8b17e0e4a1de2aceb7565dc4b9552b7078e
```

The credential will be permanently revoked, and impossible to use again:

```
$ ykman fido credentials list
Enter your PIN:
Credential ID  RP ID         Username    Display name
87422920...    webauthn.me   test        Test User
53badd61...    webauthn.io   Justin Two  Justin Two
```

You can't delete any of the non-resident credentials you have created (since they are stored by the relying party). As long as the relying party continues to store the credential, it can always ask you (or whoever holds your YubiKey and knows its FIDO PIN) to use the credential sometime in the future.

The only way to fully revoke your non-resident credentials is to reset the FIDO app on your YubiKey. Not only will this delete all your resident credentials, but it will also wipe the secret master key needed to decrypt and use your existing FIDO credentials — both resident and non-resident:

```
$ ykman fido reset
WARNING! This will delete all FIDO credentials, including FIDO U2F credential
Remove and re-insert your YubiKey to perform the reset...
Touch your YubiKey...
```

WARNING

Do not run the `reset` command unless you're 100% sure you've added a back-up primary authentication option at every website where you've used your YubiKey's FIDO app for passwordless authentication (ie Passkeys), and a back-up two-factor authentication option at every website where you've used your YubiKey's FIDO app for 2FA! You will no longer be able to log into any websites where you've failed to add a back-up option.

In most cases where you've used FIDO for Passkeys, the website will have used a resident credential — you'll be able to view those websites via the `ykman fido credentials list` command. However, there is no equivalent command to view non-resident credentials (your YubiKey simply does not keep track of them) — so if the website used a non-resident credential (rarely the case with primary authentication, but more frequently the case with 2FA), you will not know unless you noted it at the time you set up the credential.

Therefore, it's a good idea to manually keep track of which websites you've used your YubiKey for FIDO authentication (either as notes in your password manager, or via a standalone spreadsheet), so that you will know at which websites you need to update your credentials when you replace (or reset) your YubiKey.

OATH

The OATH application doesn't require much set up. This app can be used to store up to 32 TOTP (Time-based One Time Password) or HOTP (HMAC-based One Time Password) secrets on your YubiKey, from which you can generate the 2FA (2-Factor Authentication) codes used by many websites.

You may wish to install the Yubico Authenticator app on your phone in order to help you use this app — in particular to scan the QR codes that can be used as a shortcut to set up a new TOTP secret in your YubiKey. However, you don't need to use the Yubico Authenticator if you don't want — you can save secrets and generate codes purely through the ykman CLI.

OATH PASSWORD

The one set-up step you should take before using this app is to set a password for the app. Once set, your YubiKey will require this password to add or delete secrets, as well as to generate 2FA codes.

Run the following command to set a new OATH password:

```
$ ykman oath access change
```

There are no size or character constraints on this password. However, unlike the PINs of other YubiKey apps, there is no maximum limit on the number of consecutive wrong attempts an adversary can make — if given unfettered access to your YubiKey, an adversary would be limited only by the YubiKey hardware's ability to process password attempts, allowing around 100,000 password attempts per day.

Therefore you should use a reasonably strong password of the "correct horse battery staple" variety.

TIP

Generate a password with four random words by running the following command:

```
$ shuf -n4 /usr/share/dict/words
goldsmith
sentimental
weepings
sidewall
```

Enter your new password at the prompt, and then repeat it for confirmation:

```
Enter the new password:
Repeat for confirmation:
Password updated.
```

Write down the password on the same piece of paper where you wrote down your OpenPGP PIN and other YubiKey passphrases.

TIP

You can mitigate the inconvenience of using a strong OATH password by saving the password on your computer. The Yubico Authenticator on most platforms will allow you to save the password in the platform's keychain or keystore.

With the ykman CLI, you can run the following command to save your password in the platform's keychain or keystore:

```
$ ykman oath access remember
Enter the password:
Password remembered.
```

And the following command to remove it:

```
$ ykman oath access forget
Password forgotten.
```

ADD TOTP SECRET

When you set up 2FA for a website using TOTP, the website usually will present you with a QR code to scan. This QR code embeds the TOTP secret (as well as some related metadata, such as the website's name and your username on it). If the website also shows the secret itself (which will be a 32-character code like `ABCDEFGHIJKLMNOPQRSTUVWXYZ234567`), you can copy that secret and add it to your YubiKey OATH app via the ykman CLI.

If a website displays the just QR code for a TOTP secret, and not the secret itself, you can decode the QR code to find the secret. If you download the image containing the QR code, you can usually use Zbar to decode it (available in most Linux distros as the `zbar`, `zbarimg`, or `zbar-tools` package). You may also be able to decode the QR code in the browser with a browser extension such as QR Code Reader.

If you decode the QR code, the result will look like this:

```
otpauth://totp/justin%40example.com?issuer=Example%20Co.&secret=ABCDEFGHI
```

The value of the URL parameter named `secret` in the decoded QR code is the TOTP secret.

Run the following command to save the secret in your YubiKey OATH app:

```
$ ykman oath accounts add --oath-type TOTP --touch --issuer example.com justi
```

The `--touch` flag adds the requirement to touch your YubiKey whenever you want to generate a 2FA code from the secret. The secret will be stored using a combination of issuer (`example.com` in the above example) and username (`justin` above).

NOTE

In the Yubico Authenticator, for each entry, the issuer will be displayed above the username, like this:

> **example.com**
> justin

> **Google**
> jl123@example.net

> **AOL**
> justin

Enter the new secret when prompted, and then enter your OATH password:

```
Enter a secret key (base32): ABCDEFGHIJKLMNOPQRSTUVWXYZ234567
Enter the password:
```

GENERATE TOTP 2FA CODE

You can view all the accounts for which you've stored secrets by running the following command:

```
$ ykman oath accounts list
```

When prompted, enter your OATH password:

```
Enter the password:
```

All stored OATH accounts will be listed, each formatted with the issuer name, a colon, and then the account username:

```
example.com:justin
Google:jl123@example.net
AOL:justin
```

You can generate a 2FA code for any account that matches a unique string from this list (like `jl123`) by running the following command:

```
$ ykman oath accounts code jl123
```

Enter your OATH password and touch your YubiKey when prompted, and the matching account will be listed, along with the current 2FA code for the account:

```
Enter the password:
Touch your YubiKey...
Google:jl123@example.net  904875
```

Enter the code shown in the command's output (`904875` in the above example) into the website's 2FA challenge form.

OTP

The OTP app manages 2 slots that each can contain one of four different types of credentials:

1. A YubiCloud secret
2. A TOTP (Time-based One Time Password) secret
3. A HOTP (HMAC-based One Time Password) secret
4. A static secret (up to 38 characters long)

The OTP app will emit a 2FA (2-Factor Authentication) code for types 1-3, or emit the static secret for type 4, when you “short press” (slot 1) or “long press” (slot 2) the YubiKey.

Slot 1 comes preloaded with a YubiCloud secret. If you use, or intend to use, YubiCloud for 2FA, you should protect it with an access code (to prevent overwriting the secret or its config settings).

If you don’t intend to use YubiCloud, you can just delete the preloaded secret with the ykman CLI (you can always re-enroll your YubiKey with a different YubiCloud secret later):

```
$ ykman otp delete 1
Do you really want to delete the configuration of slot 1? [y/N]: y
Deleting the configuration in slot 1...
```

If you used up all 32 slots of the OATH app, you could use the 2 OTP slots for TOTP (or HOTP) secrets. The most common use for the OTP slots, however, is to input a static secret into a password dialog on your computer when it starts up (such as the password to unlock the encrypted boot drive, or to log into your user account).

If you use one or both OTP slots for a static secret, you should protect them with an access code (to prevent overwriting the secret or its config settings). Each slot uses a different access code (and by default, has no code). Before setting an access code on a slot, however, you have to configure it with a secret.

OTP STATIC SECRET

To set up OTP slot 1 to use a static secret, run the following command:

```
$ ykman otp static --generate --length 38 1
```

This will generate a random 38-character password (using Yubico’s custom modhex encoding by default), and store it in slot 1.

To view the password, enter the following command, and then press the YubiKey until it starts emitting text:

```
$ read
nJXB.Gtipd>YnIEbukC.jpyDj>PGKncdikgtTP
```

OTP ACCESS CODE

To protect a slot with an access code (after the slot has been configured with a secret), run the following command:

```
$ ykman otp settings --new-access-code - 1
```

The above sets the access code for slot 1 (each slot has a separate access code). The access code must be 12 hex digits. You'll only need it if you want to change the secret (or other config settings) of slot 1, or to delete the secret.

TIP

To generate a good random access code, run the following command:

```
$ openssl rand -hex 6
576bc54882f3
```

Enter the new access code once, then again, and then enter `y` to confirm:

```
Enter new access code:
Repeat for confirmation:
Update the settings for slot 1? All existing settings will be overwritten. [y
Updating settings for slot 1...
```

The "existing settings will be overwritten" warning doesn't apply to the secret itself, but to the other options that can be specified with the settings command:

- `--no-enter`: Don't emit an enter keypress after the secret (defaults to false unless specified)
- `--pacing`: Milliseconds between each character emitted (defaults to 0 unless specified)
- `--use-numeric-keypad`: Emit keypad scancodes (defaults to false unless specified)

Write down the access code (along with the slot number) on the same piece of paper on which you wrote your OpenPGP PINs. You'll need it later if you want to overwrite the secret with something else (or delete it).

For example, to delete the secret in slot 1 now, run the following command:

```
$ ykman otp --access-code - delete 1
```

If you don't specify the `--access-code` option, the command will fail with an error. Supplying `-` to the `--access-code` option will cause the command to prompt for the access code. Enter it when prompted:

```
Enter the access code:
Do you really want to delete the configuration of slot 1? [y/N]: y
Deleting the configuration in slot 1...
```

The slot's secret (as well as the slot's configuration settings, including access code), will be wiped from the YubiKey.

BACKING UP

You don't back up a YubiKey — the most important feature of a hardware security card like a YubiKey is that its private keys are held secret within the card itself, and can't be extracted. But you will loose a YubiKey at some point — you'll misplace it, or drive over it, or have it stolen out of your backpack, or drop it in a lake.

The best back-up is the buddy system: make sure at least one other person has an equivalent set of credentials for every application for which you use your YubiKey.

For everything you sign with your YubiKey (like software releases or PKI certificates), make sure someone else has another signature key that will be accepted equivalently by everything that needs to verify your signature; for everything encrypted with your YubiKey (like disk drives or archived backups), make sure those things are encrypted to allow someone else's decryption key to also decrypt them; and for everything to which your YubiKey helps you authenticate (with signature-based or "passwordless" primary authentication, or with a one-time password or FIDO credential as a second factor), make sure someone else has a separate set of credentials that will allow her to authenticate and manage the application in your place.

The second best back-up is a second YubiKey (or another hardware security card, or set of cards, that have all the features you use from your YubiKey). Make sure your second YubiKey's signatures will be accepted everywhere your first YubiKey's are. Make sure you (and others) encrypt everything for your second YubiKey's decryption keys, just like your first YubiKey. Make sure your second YubiKey's authentication keys are accepted as authorized keys everywhere your first YubiKey is; make sure you set up passwordless authentication with your second YubiKey everywhere you've done so for your first YubiKey; and make sure to add your second YubiKey as a second factor everywhere you've set up your first YubiKey to be.

Finally, consider adding a third (buddy or YubiKey) — there's an old saying "two is one, and one is none". Anything that's really important should have a third key that can access it, where at least one of the three keys is rarely, if ever, at the same place at the same time as the other two.

BACKING UP PINS

Make sure you write down all the PINs and passphrases that you use for your YubiKey's apps on a piece of paper. (This same information is a good candidate to store in your password manager — but you may not be able to access your password manager without first having to enter one the very PINs that you have forgotten.)

Apply the same "rule of threes" that you use for other backups to paper copies of your PINs: Keep three copies, with at least one copy at a different geographical location than the other two. You will also want to have at least one copy near at hand to where you usually work, so you can refer to it easily whenever you need to enter a PIN that you've forgotten.

Here's an example of what you should have written down:

```
YubiKey Serial Number: 1234578
Nickname: key1
Config lock code: 10a43eacde8630603fc15016bc508605
OpenPGP master key ID: 0xABCDEF1234567890
OpenPGP user PIN: scalded penlight
OpenPGP admin PIN: preachiest chickens
PIV PIN: noughts
PIV PUK: kXHt2J1u
FIDO PIN: modicums overproduced
OATH password: goldsmith sentimental weepings sidewall
OTP slot 1: work laptop FDE
OTP slot 1 access code: 576bc54882f3
OTP slot 1 secret: nJXB.Gtipd>YnIEbukC.jpyDj>PGKncdikgtTP
```

4/5/2023 by Justin Ludwig

Smart Cards, Keys, SSH