Securing the Software Supply Chain: Artefact and Commit Signing

Joost van Dijk | Yubico
Nordic Software Security Summit 2024
24 September 2024

yubico

Agenda

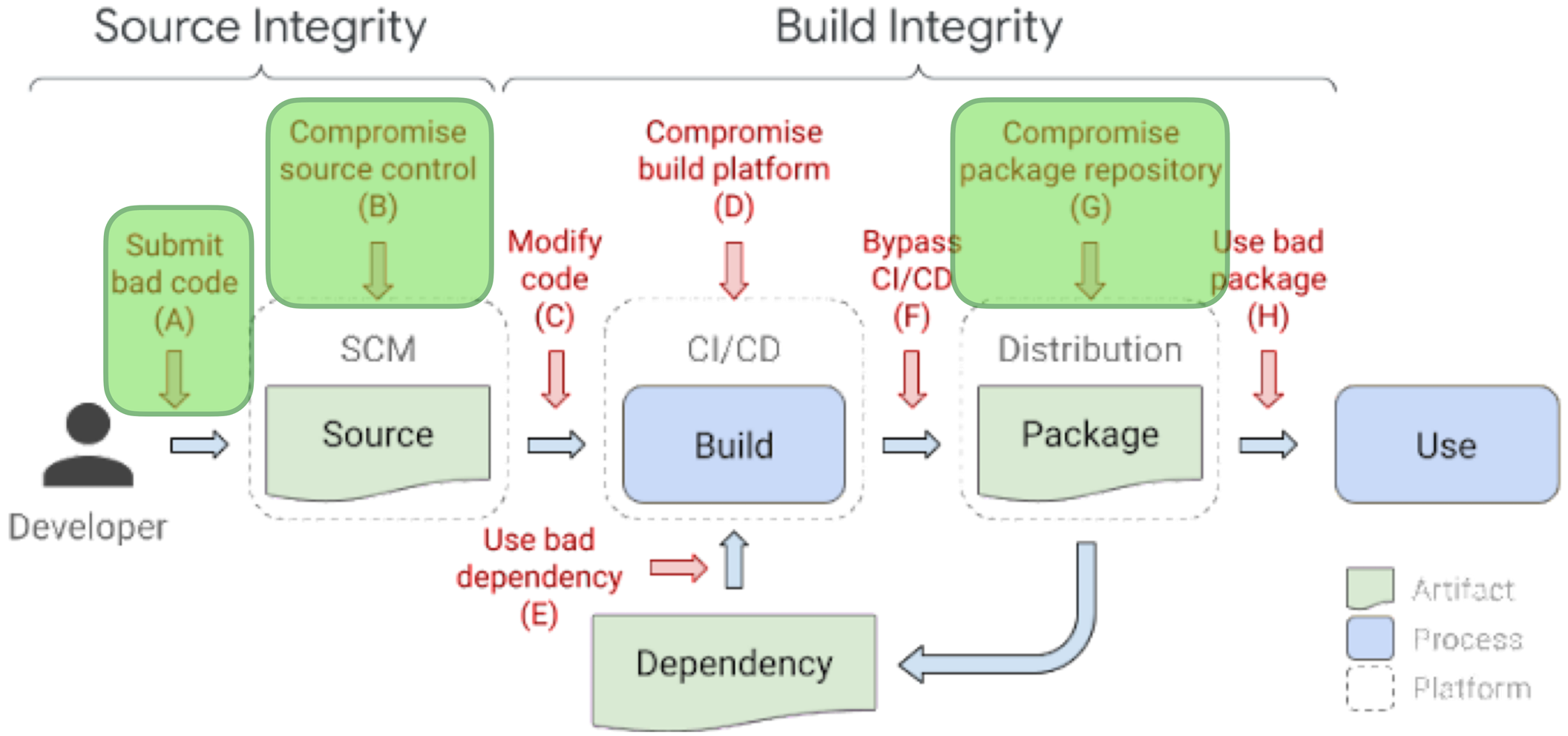
- **Introduction**
- **Signing:**
  OpenPGP, OpenSSH, OpenSSL, X.509 and PKI
- **Hardware:**
  HSMs, smartcards, PIV, PGPcard, FIDO Security keys, PKCS#11, CNG
- **Authentication**
- **Key Attestation**
- **“Keyless” signing**

yubico

Software Supply Chain Security

yubico

Software Supply Chain Security - Threats



https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html

Installing software packages

- Some examples -
  what are the differences from a security perspective?

```
# (1) install software from a web site
$ curl https://package.org/install.sh | sudo sh
. . .

# (2) install a Python package from PyPI
$ sudo pip install package_name
. . .

# (3) install a package on Debian/Ubuntu
$ sudo apt install package_name
. . .

# (4) install a package on macOS
$ sudo installer -pkg package.pkg -target /
. . .
```

Introduction

Artefact Signing

private key k

public key p

Signature = sign(k, Artefact)

result = verify(p, Artefact, Signature)

Algorithms, Key types, Formats, encodings

- There are several signing algorithms to choose from (RSA, ECDSA, EDDSA, …)
- Every signing algorithm can have several parameters (key length, elliptic curve used, hashing algorithm, padding method…)
- The keys and the generated signatures can use different formats (ASN.1, PEM, “raw”, JWK) and encodings (binary, hex, base64, …)
- JSON Web Signature example: ES256 indicates the ECDSA algorithm using the NIST P-256 curve and hash function SHA-256

Example: OpenSSL Signing

```
# generate key pair
$ openssl ecparam -genkey -name prime256v1 -noout -out private.pem
$ cat private.pem
-----BEGIN EC PRIVATE KEY-----
MHcCAQEEIPiFJmMDnooHlbLPfZ0lLdwE7ahUN814XWBijb0IkshgoAoGCCqGSM49
AwEHoUQDQgAEb3cRatQQDrCzWCg+pQLXz3xqS1npN3YOrhZaeC1r1bG0hAetUXvo
6cl87qyG8V1Uaav+8/Z7s9c21zUSD7370Q==
-----END EC PRIVATE KEY-----
# sign a message using the private key
$ cat message.txt
I owe you a drink
$ openssl dgst -sha256 -sign private.pem message.txt > signature.bin
$ xxd signature.bin
00000000: 3045 0221 00a6 4914 7220 2a66 dbf9 76c6  0E.!..I.r *f..v.
00000010: 7471 f07e 182a c571 48c3 d04b 733c 07be  tq.~.*.qH..Ks<..
00000020: 68b3 bce9 c602 2067 94ea dc77 ba70 50f8  h..... g...w.pP.
00000030: 98f3 ccba 0db7 10d6 7e4c 7471 cbc2 f65c  ........~Ltq...\
00000040: 386e ded6 9cef 9d                        8n.....
# verify the signature using the public key
$ openssl ec -in private.pem -pubout -out public.pem
read EC key
writing EC key
$ cat public.pem
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEb3cRatQQDrCzWCg+pQLXz3xqS1np
N3YOrhZaeC1r1bG0hAetUXvo6cl87qyG8V1Uaav+8/Z7s9c21zUSD7370Q==
-----END PUBLIC KEY-----
$ openssl dgst -sha256 -verify public.pem -signature signature.bin message.txt
Verified OK
$
```

Verifying signatures

- To verify a signature, you need to know
  1. the signing algorithm, formats, and encodings used
  2. the public key of the signer

- As important, you need to
  3. be able to identify the signer
  4. trust the signer

Example: OpenPGP Signing

```
$ # generate key pair
$ gpg --batch --passphrase jabberwocky --quick-generate-key johndoe@example.org
gpg: revocation certificate stored as '/Users/jd/mygnupg/openpgp-revocs.d/A71EBEDE1A231757E73201159837C22B976BD366.rev'
$ gpg --list-keys
gpg: checking the trustdb
gpg: marginals needed: 3  completes needed: 1  trust model: pgp
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
gpg: next trustdb check due at 2027-09-19
/Users/jd/mygnupg/pubring.kbx
---------------------------------
pub   ed25519 2024-09-19 [SC] [expires: 2027-09-19]
      A71EBEDE1A231757E73201159837C22B976BD366
uid           [ultimate] johndoe@example.org
sub   cv25519 2024-09-19 [E]

$ # sign a message using the private key (opens passphrase dialogue)
$ cat message.txt
I owe you a drink
$ gpg --sign --detach-sig --armor message.txt
$ cat message.txt.asc
-----BEGIN PGP SIGNATURE-----
iHUEABYKAB0WIQSnHr7eGiMXV+cyARWYN8Irl2vTZgUCZuyo6wAKCRCYN8Irl2vT
ZqZ1AP0b98tqcG4deDfS3gFplv0escGCDtkerUsK4pCT9Gf/rAEAgUbjA5DItc6z
Hmgf3IuuW3AMLs/7iOHos04HvlpnPA4=
=ab37
-----END PGP SIGNATURE-----
$ # verify the signature using the public key
$ gpg --verify message.txt.asc
gpg: assuming signed data in 'message.txt'
gpg: Signature made Fre 20 Sep 00:42:51 2024 CEST
gpg:                using EDDSA key A71EBEDE1A231757E73201159837C22B976BD366
gpg: Good signature from "johndoe@example.org" [ultimate]
$
```

Notes on PGP

- PGP is often found hard to use
- Often requires manual verification
- Sometimes PGP signatures are ignored (eg PyPI)

- In this example, the signing key is stored in a file, encrypted with a passphrase

- With PGP, public keys are associated with identifiers user@domain

- Still need to trust that identity
  - Direct Trust
  - Web of trust
  - PKI

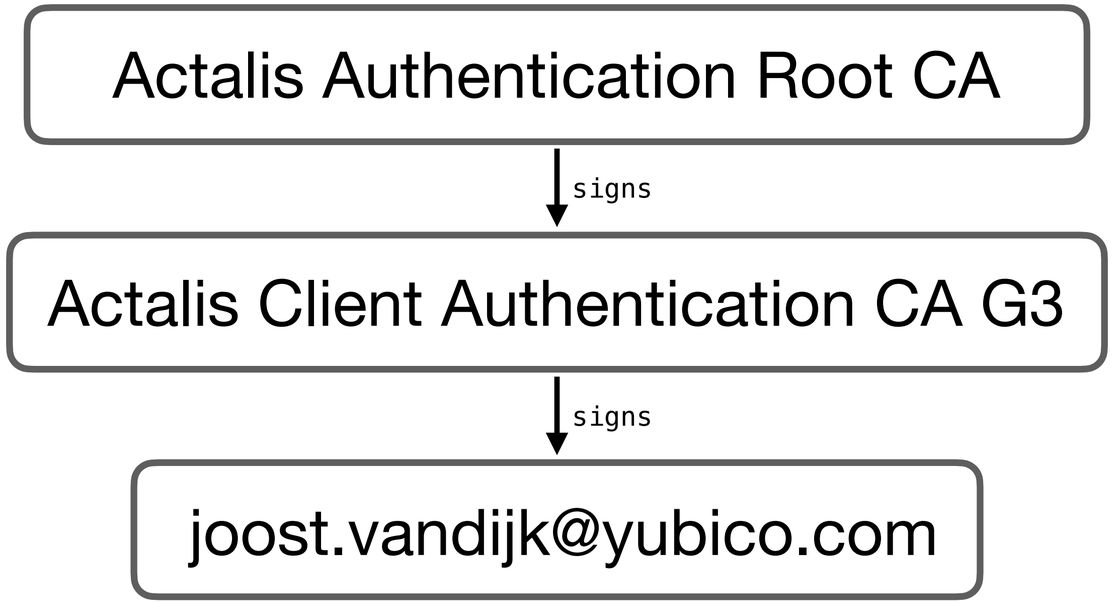
PGP: manually verify

```
$ # download tarball and signature
$ wget -q https://developers.yubico.com/YubiHSM2/Releases/yubihsm2-sdk-2024-09-ubuntu2404-amd64.tar.gz
$ wget -q https://developers.yubico.com/YubiHSM2/Releases/yubihsm2-sdk-2024-09-ubuntu2404-amd64.tar.gz.sig
$ # verify signature
$ gpg --verify yubihsm2-sdk-2024-09-ubuntu2404-amd64.tar.gz.sig
gpg: assuming signed data in 'yubihsm2-sdk-2024-09-ubuntu2404-amd64.tar.gz'
gpg: Signature made Mån  9 Sep 17:16:52 2024 CEST
gpg:                using RSA key A8CE167914EEE232B9237B5410CAC4962E03C7CC
gpg: Can't check signature: No public key
$ # retrieve key
$ gpg --recv-keys A8CE167914EEE232B9237B5410CAC4962E03C7CC
gpg: key 27A9C24D9588EA0F: 3 duplicate signatures removed
gpg: key 27A9C24D9588EA0F: public key "Aveen Ismail <aveen.ismail@yubico.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1
$ # verify signature
$ gpg --verify yubihsm2-sdk-2024-09-ubuntu2404-amd64.tar.gz.sig
gpg: assuming signed data in 'yubihsm2-sdk-2024-09-ubuntu2404-amd64.tar.gz'
gpg: Signature made Mån  9 Sep 17:16:52 2024 CEST
gpg:                using RSA key A8CE167914EEE232B9237B5410CAC4962E03C7CC
gpg: Good signature from "Aveen Ismail <aveen.ismail@yubico.com>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: 1D73 08B0 055F 5AEF 3694  4A8F 27A9 C24D 9588 EA0F
     Subkey fingerprint: A8CE 1679 14EE E232 B923  7B54 10CA C496 2E03 C7CC
$ # compare with known list of PGP keys
$ w3m -dump https://developers.yubico.com/Software_Projects/Software_Signing.html | grep -A1 1D73
  • Aveen Ismail <aveen.ismail@yubico.com> 1D73 08B0 055F 5AEF 3694 4A8F 27A9
    C24D 9588 EA0F
$
```

X.509 Certificates

```
$ # sign a message using the private key
$ openssl dgst -sha256 -sign joostvandijk.key message.txt > signature.bin
Enter pass phrase for joostvandijk.key:
$ # inspect the certificate
$ openssl x509 -in joostvandijk.crt -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            71:99:7e:36:f3:f4:85:1e:5c:95:16:54:ca:c4:5e:20
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=IT, ST=Bergamo, L=Ponte San Pietro, O=Actalis S.p.A., CN=Actalis Client Authentication CA G3
        Validity
            Not Before: Dec 15 09:37:02 2023 GMT
            Not After : Dec 15 09:37:01 2024 GMT
        Subject: CN=joost.vandijk@yubico.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:c4:40:a2:10:a3:ee:f4:f5:cb:36:60:5e:da:ef:
…
                Exponent: 65537 (0x10001)
        X509v3 extensions:
          Authority Information Access:
                CA Issuers - URI:http://cacert.actalis.it/certs/actalis-autclig3
                OCSP - URI:http://ocsp09.actalis.it/VA/AUTHCL-G3
            X509v3 Subject Alternative Name:
                email:joost.vandijk@yubico.com
…
    Signature Algorithm: sha256WithRSAEncryption
    Signature Value:
        d5:fc:57:a1:98:4e:fc:64:22:46:78:b9:e8:1a:ba:02:04:22:
…
$ # extract the public key from the certificate
$ openssl x509 -in joostvandijk.crt -noout -pubkey -out public.pem
$ # verify the signature using the public key
$ openssl dgst -sha256 -verify public.pem -signature signature.bin message.txt
Verified OK
$
```

Public Key Infrastructure (PKI)



Validate the certificate

```
$ # try to validate the certificate
$ openssl verify joostvandijk.crt
CN=joost.vandijk@yubico.com
error 20 at 0 depth lookup: unable to get local issuer certificate
error joostvandijk.crt: verification failed
$ # extract issuer information from the certificate
$ openssl x509 -in joostvandijk.crt -noout -text | grep 'Authority Information Access' -A1
            Authority Information Access:
                CA Issuers - URI:http://cacert.actalis.it/certs/actalis-autclig3
$ # retrieve the intermediate CA certificate
$ wget -q http://cacert.actalis.it/certs/actalis-autclig3
$ # validate the certificate against the local root CA store
$ openssl verify -untrusted actalis-autclig3 joostvandijk.crt
joostvandijk.crt: OK
$ # ... and show the complete CA chain
$ openssl verify -untrusted actalis-autclig3 -show_chain joostvandijk.crt
joostvandijk.crt: OK
Chain:
depth=0: CN=joost.vandijk@yubico.com (untrusted)
depth=1: C=IT, ST=Bergamo, L=Ponte San Pietro, O=Actalis S.p.A., CN=Actalis Client Authentication CA G3 (untrusted)
depth=2: C=IT, L=Milan, O=Actalis S.p.A./03358520967, CN=Actalis Authentication Root CA
$ # The list of root CA certificates is installed locally
$ grep -A5 -h 'Actalis Authentication Root CA' /opt/homebrew/opt/ca-certificates/share/ca-certificates/cacert.pem
Actalis Authentication Root CA
==============================
-----BEGIN CERTIFICATE-----
MIIFuzCCA6OgAwIBAgIIVwoRl0LE48wwDQYJKoZIhvcNAQELBQAwazELMAkGA1UEBhMCSVQxDjAM
BgNVBAcMBU1pbGFuMSMwIQYDVQQKDBpBY3RhbGlzIFMucC5BLi8wMzM1ODUyMDk2NzEnMCUGA1UE
AwweQWN0YWxpcyBBdXRoZW50aWNhdGlvbiBSb290IENBMB4XDTExMDkyMjExMjIwMloXDTMwMDky
…
$
```

Code signing certificates

- Code signing certificates to sign binaries (.exe, drivers, Java, …)
- Can be expensive for some developers
  (e.g. for Open Source Software)
- Verification automated by platform
- For instance:
  - Microsoft Authenticode:Requires FIPS 140 Level 2 or Common Criteria EAL 4+ certified hardware
  - Required subscription fees (Android, Apple)

Checking signatures

```
$ pkgutil  --check-signature /Applications/Google\ Chrome.app
Package "Google Chrome.app":
   Status: signed by a certificate trusted by macOS
   Certificate Chain:
    1. Developer ID Application: Google LLC (EQHXZ8M8AV)
       Expires: 2027-02-01 22:12:15 +0000
       SHA256 Fingerprint:
           0B DA 2A CA 4B 96 7F D1 5B B6 84 0C 54 DE 1C C2 30 92 1E FD 1A 18
           1A 6F 0F C8 14 AD A3 FF AA 4F
       ------------------------------------------------------------------------
    2. Developer ID Certification Authority
       Expires: 2027-02-01 22:12:15 +0000
       SHA256 Fingerprint:
           7A FC 9D 01 A6 2F 03 A2 DE 96 37 93 6D 4A FE 68 09 0D 2D E1 8D 03
           F2 9C 88 CF B0 B1 BA 63 58 7F
       ------------------------------------------------------------------------
    3. Apple Root CA
       Expires: 2035-02-09 21:40:36 +0000
       SHA256 Fingerprint:
           B0 B1 73 0E CB C7 FF 45 05 14 2C 49 F1 29 5E 6E DA 6B CA ED 7E 2C
           68 C5 BE 91 B5 A1 10 01 F0 24
```

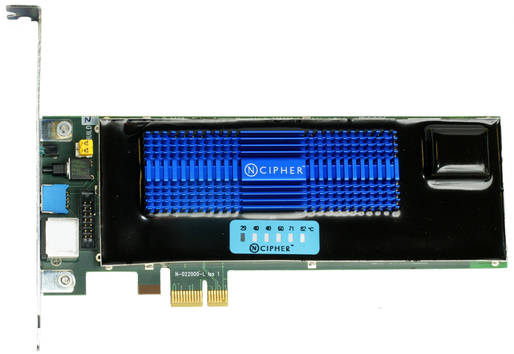
Protecting signing keys

- **The signing key needs to be kept private.**

- **Where to store the private key?**

- **in a file**
  - Easy, but hard to prevent sprawl
  - Encrypt using a passphrase or wrap key
- **in memory**
  - Easy, but volatile
  - Good for ephemeral keys
- **In hardware**
  - Better control
  - Attestation - e.g. provide proof a key was generated in secure hardware

Protecting Keys in Hardware



Hardware Security Modules

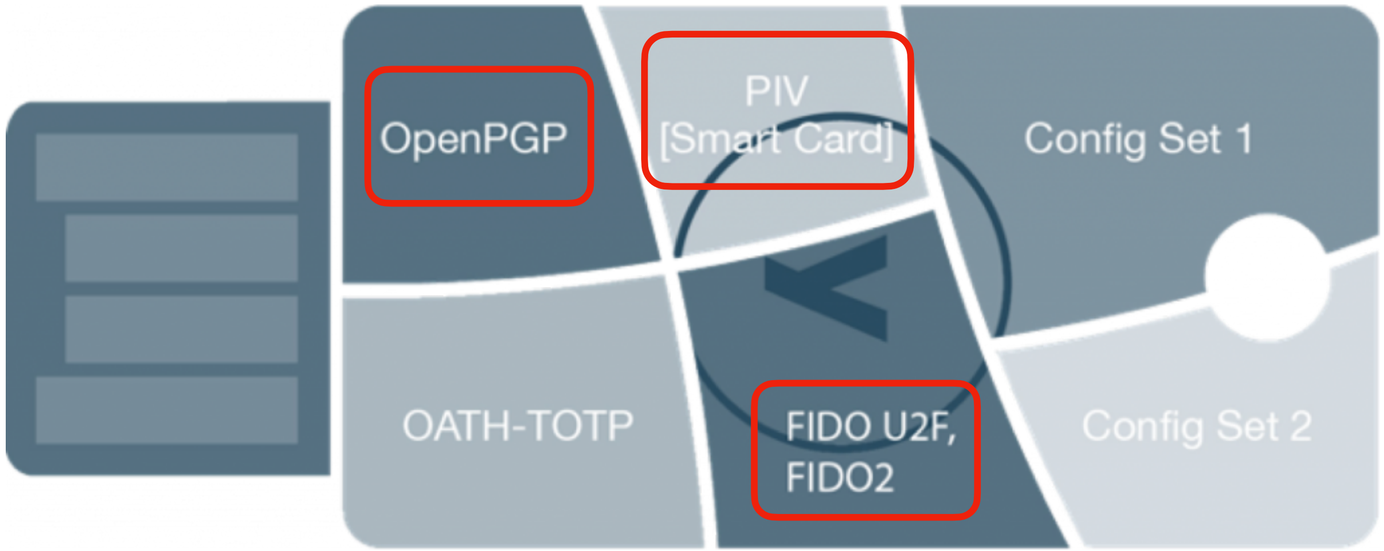
hardware tokens / smart cards / security keys

yubico

Why use hardware to protect keys?

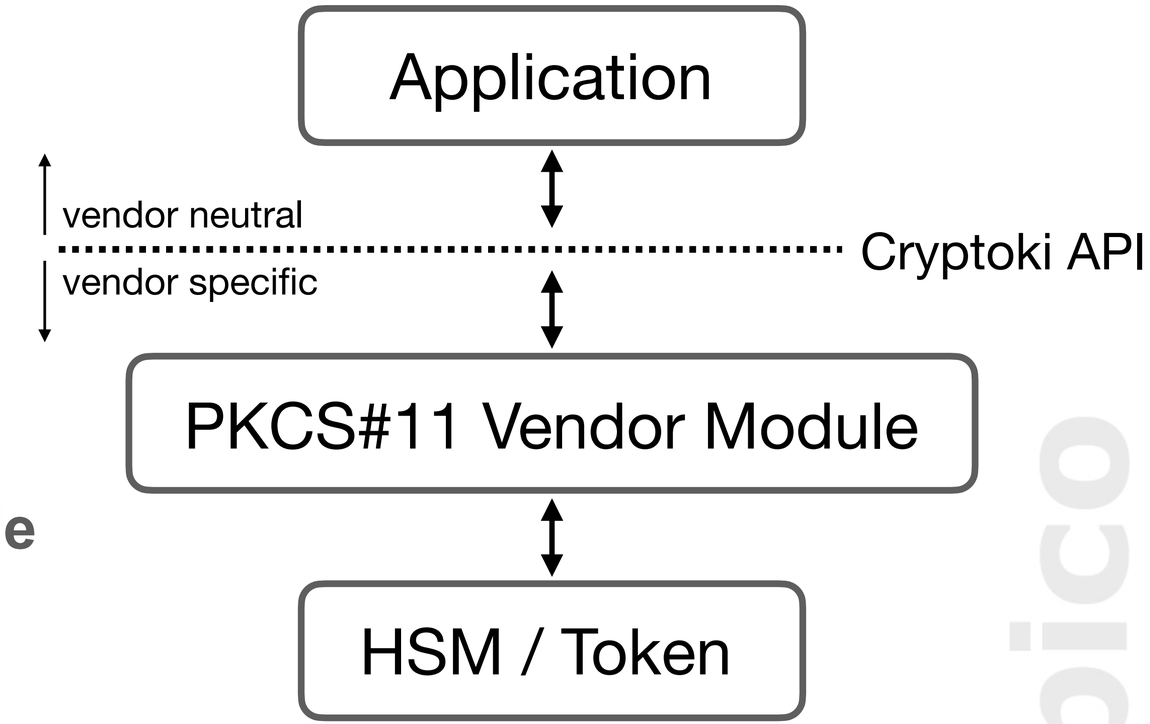
- **Help prevent private key compromise**
- **Control key sprawl**
- **Meet regulatory requirements**
- **Example: CA/B Forum requirements for code signing certificates**
  key generation and protection in a hardware crypto module that conforms to at least FIPS 140-2 Level 2 or Common Criteria EAL 4+

YubiKey Signing Applications



PKCS#11 and CNG

- PKCS #11 is a cryptographic token interface standard
- Specifies an API, called *Cryptoki*
- Lets application developers write software that is independent of any specific HSM vendor
- HSM vendor supplies a PKCS#11 module specific for a particular HSM model
- A similar API is available as part of Windows:
  Cryptography API: Next Generation (CNG)



Using PKCS#11 with YubiKey

```
$ # get info from token
$ pkcs11-tool --module /opt/homebrew/lib/libykcs11.dylib -I
Cryptoki version 3.0
Manufacturer     Yubico (www.yubico.com)
Library          PKCS#11 PIV Library (SP-800-73) (ver 2.60)
Using slot 0 with a present token (0x0)
$ # generate a new key pair, login with SO PIN (01020304050607080102030405060708010203040506070801020304050607080)
$ pkcs11-tool --module /opt/homebrew/lib/libykcs11.dylib --login --login-type so --keypairgen --key-type RSA:2048 --id 1
Using slot 0 with a present token (0x0)
Logging in to "YubiKey PIV #17671830".
Please enter SO PIN: 01020304050607080102030405060708010203040506070801020304050607080
Key pair generated:
Private Key Object; RSA
  label:      Private key for PIV Authentication
  ID:         01
  Usage:      decrypt, sign
  Access:     sensitive, always sensitive, never extractable, local
Public Key Object; RSA 2048 bits
  label:      Public key for PIV Authentication
  ID:         01
  Usage:      encrypt, verify
  Access:     local
$ # sign a message using the private key
$ pkcs11-tool --module /opt/homebrew/lib/libykcs11.dylib --sign -i message.txt -o signature.bin
Using slot 0 with a present token (0x0)
Logging in to "YubiKey PIV #17671830".
Please enter User PIN: 123456
Using signature algorithm RSA-PKCS
$ # verify the signature using the public key
$ pkcs11-tool --verify -m RSA-PKCS -i message.txt --signature-file signature.bin
Using slot 0 with a present token (0x0)
Using signature algorithm RSA-PKCS
Signature is valid
$
```

Git Signing

Signing git commits

- Anyone can spoof a Git committer or author name

```
$ git config user.name "Elvis Presley"
$ git config user.email "elvis@heaven.org"
```

- If you have permission to push commits to a repo, you can impersonate anyone.
- To avoid 'spoofing' commits, require them to be signed

- Can sign using PGP or SSH (via PIV/PKCS#11 or FIDO)

Example: OpenSSH

```
$ # generate key pair
$ ssh-keygen -f id_ecdsa -t ecdsa -N passphrase -C signer@example.org
Generating public/private ecdsa key pair.
Your identification has been saved in id_ecdsa
Your public key has been saved in id_ecdsa.pub
The key fingerprint is:
SHA256:4yJ/PZMPCYNgab7u7UndG1I+BwmuvwZLxGGXEPSCFNU signer@example.org
…
$ # sign a message using the private key
$ cat message.txt
I owe you a drink
$ ssh-keygen -Y sign -n file -f id_ecdsa message.txt
Enter passphrase:
Signing file message.txt
Write signature to message.txt.sig
$ cat message.txt.sig
-----BEGIN SSH SIGNATURE-----
U1NIU0lHAAAAAQAAAGgAAAATZWNkc2Etc2hhMi1uaXN0cDI1NgAAAAhuaXN0cDI1NgAAAE
EEvlyEH6whj3i3B7Pt5MIpc7gi/0fsok4Mv9cWBdFNXtQVkoZuVT8kFCJHKa7QIDpAEwNR
4WGO9c0fz3873grivQAAAARmaWxlAAAAAAAAAAZzaGE1MTIAAABkAAAAE2VjZHNhLXNoYT
ItbmlzdHAyNTYAAABJAAAAIGyN78vlIWbTWOMM/F31GveiZqazTvWQz6v2CWS0239HAAAA
IQCuPfSt5F1xJ7oUcw9ZUeEf4ZbwhmExJH51zKRIVS1lMg==
-----END SSH SIGNATURE-----
$ # verify the signature using the public key
$ echo "signer@example.org $(cat id_ecdsa.pub)" > allowed_signers
$ ssh-keygen -Y verify -n file -f allowed_signers -s message.txt.sig -I signer@example.org < message.txt
Good "file" signature for signer@example.org with ECDSA key SHA256:4yJ/PZMPCYNgab7u7UndG1I+BwmuvwZLxGGXEPSCFNU
$
```

OpenSSH keys on a FIDO Security Key

```
$ # generate key pair
$ ssh-keygen -f id_ecdsa -t ecdsa-sk -N '' -C signer@example.org -O verify-required
Generating public/private ecdsa-sk key pair.
You may need to touch your authenticator to authorize key generation.
Enter PIN for authenticator:
You may need to touch your authenticator again to authorize key generation.
Your identification has been saved in id_ecdsa
Your public key has been saved in id_ecdsa.pub
The key fingerprint is:
SHA256:4fVccqQhX7MXvMIXH0g/GVh+bSi5DBqEwNVzBs+ySEQ signer@example.org
…
$ # sign a message using the private key
$ cat message.txt
I owe you a drink
$ ssh-keygen -Y sign -n file -f id_ecdsa message.txt
Signing file message.txt
Enter PIN for ECDSA-SK key:
Confirm user presence for key ECDSA-SK SHA256:4fVccqQhX7MXvMIXH0g/GVh+bSi5DBqEwNVzBs+ySEQ
Write signature to message.txt.sig
$ cat message.txt.sig
-----BEGIN SSH SIGNATURE-----
U1NIU0lHAAAAAQAAAH8AAAAic2stZWNkc2Etc2hhMi1uaXN0cDI1NkBvcGVuc3NoLmNvbQ
AAAAhuaXN0cDI1NgAAAEEEHLNVvpfKKCzOzotLF0bS59a/2c3oLY35VrXMsWxnV6MwW7c3
hrYqZoqhFAxfQms7VdOPEUE4QhSV3YQTxXJRKQAAAARzc2g6AAAABGZpbGUAAAAAAAAABn
NoYTUxMgAAAHgAAAAic2stZWNkc2Etc2hhMi1uaXN0cDI1NkBvcGVuc3NoLmNvbQAAAEkA
AAAgLWRzrUI7Pca0o4rq4GDMcxJv7nyMF5fdH0ex+F5OreEAAAAhAIMJm0uYHZfyenIXHE
1s9O61YhaMBqCNqZW4RgwwhuHhBQAAAAU=
-----END SSH SIGNATURE-----
$ # verify the signature using the public key
$ echo "signer@example.org $(cat id_ecdsa.pub)" > allowed_signers
$ ssh-keygen -Y verify -n file -f allowed_signers -s message.txt.sig -I signer@example.org < message.txt
Good "file" signature for signer@example.org with ECDSA-SK key SHA256:4fVccqQhX7MXvMIXH0g/GVh+bSi5DBqEwNVzBs+ySEQ
$
```

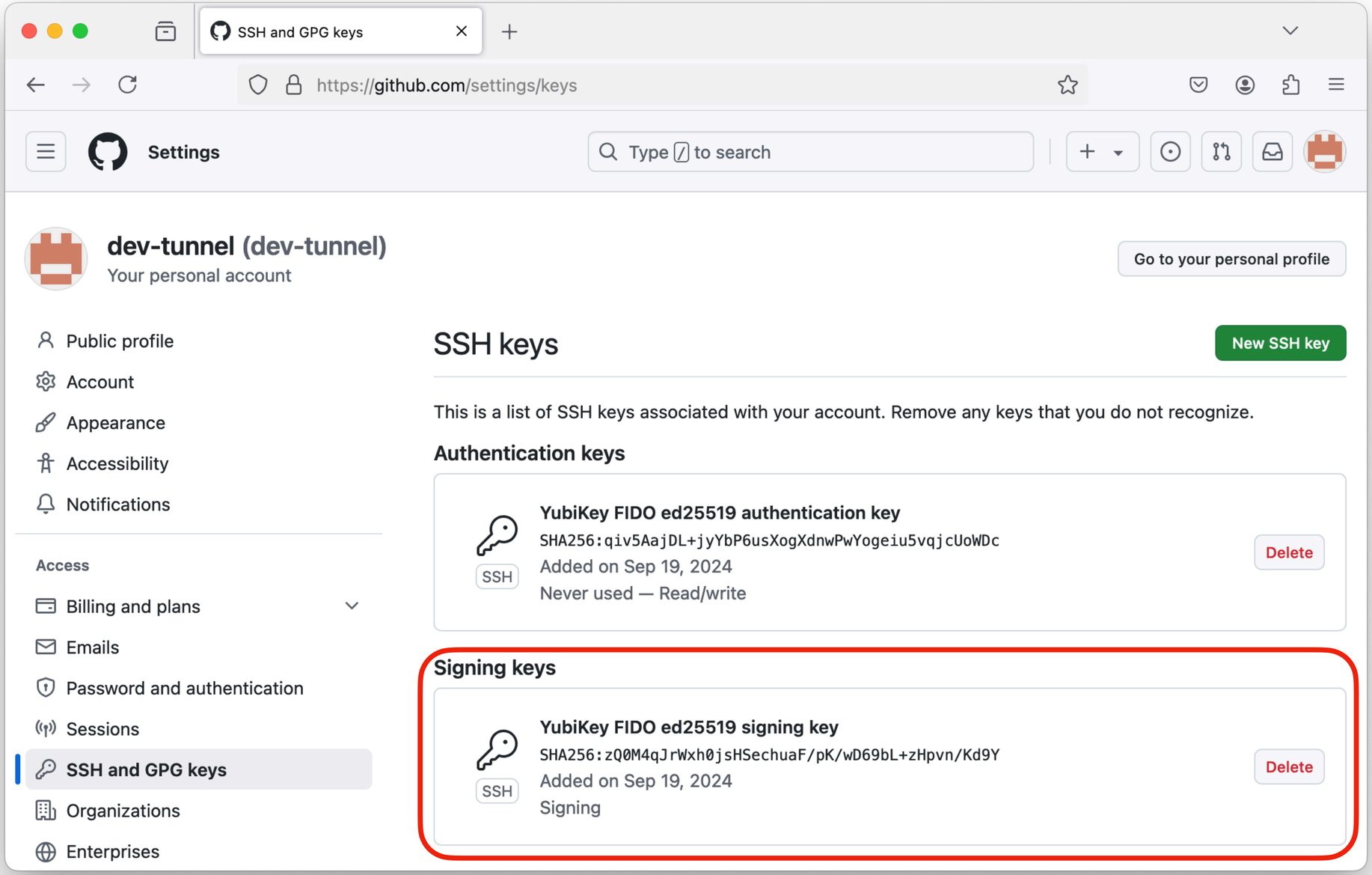
Git commit signing

```
$ # create a new git project
$ git init
Initialized empty Git repository in /Users/jd/demos/.git/
$ # configure local git repo
$ git config gpg.format ssh
$ git config user.signingKey ~/myssh/id_ecdsa
$ git config gpg.signingKey ~/myssh/id_ecdsa
$ git config gpg.ssh.allowedSignersFile ~/myssh/allowed_signers
$ # sign a commit using the private key
$ echo New Project > README
$ git add README
$ git commit -S -m'initial import'
Enter PIN for ECDSA-SK key:
[main (root-commit) 1d861bd] initial import
 1 file changed, 1 insertion(+)
 create mode 100644 README
$ git log --oneline --show-signature
1d861bd (HEAD -> main) Good "git" signature for johndoe@example.org with ECDSA-SK key SHA256:TqU8j7BgYohOy7tSvzkNbJ…
initial import
$
```

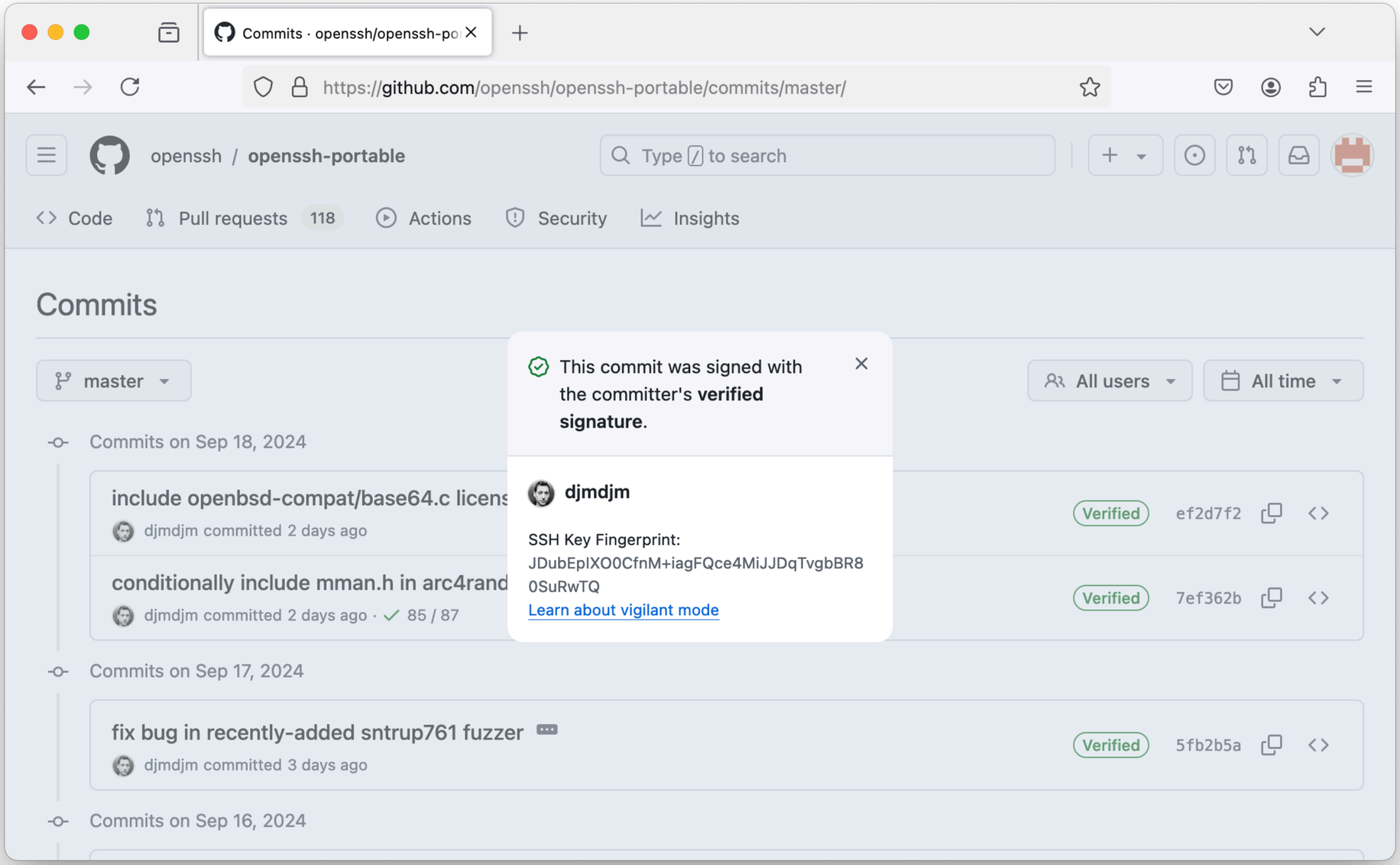
Git tag signing

```
$ # sign a tag using the private key
$ git tag -s v1.0 -m'signed v1.0 tag'
Enter PIN for ECDSA-SK key: **************
$ # view the tag signature
$ git show v1.0
tag v1.0
Tagger: John Doe <johndoe@example.org>
Date:   Thu Sep 19 22:46:37 2024 +0200

signed v1.0 tag
-----BEGIN SSH SIGNATURE-----
…
-----END SSH SIGNATURE-----

commit 1d861bd3a0ac4714d239425c3233fb2da2f3aa3d (HEAD -> main, tag: v1.0)
Author: John Doe <johndoe@example.org>
Date:   Thu Sep 19 22:46:26 2024 +0200

    initial import

diff --git a/README b/README
new file mode 100644
index 0000000..01ad0c2
--- /dev/null
+++ b/README
@@ -0,0 +1 @@
+New Project
$ # verify the tag signature
$ git tag -v v1.0
object 1d861bd3a0ac4714d239425c3233fb2da2f3aa3d
type commit
tag v1.0
tagger John Doe <johndoe@example.org> 1726778797 +0200

signed v1.0 tag
Good "git" signature for johndoe@example.org with ECDSA-SK key SHA256:TqU8j7BgYohOy7tSvzkNbJJ6foP0t9AJIE3nnEHQdCI
$
```

GitHub Signing keys



Verified commits

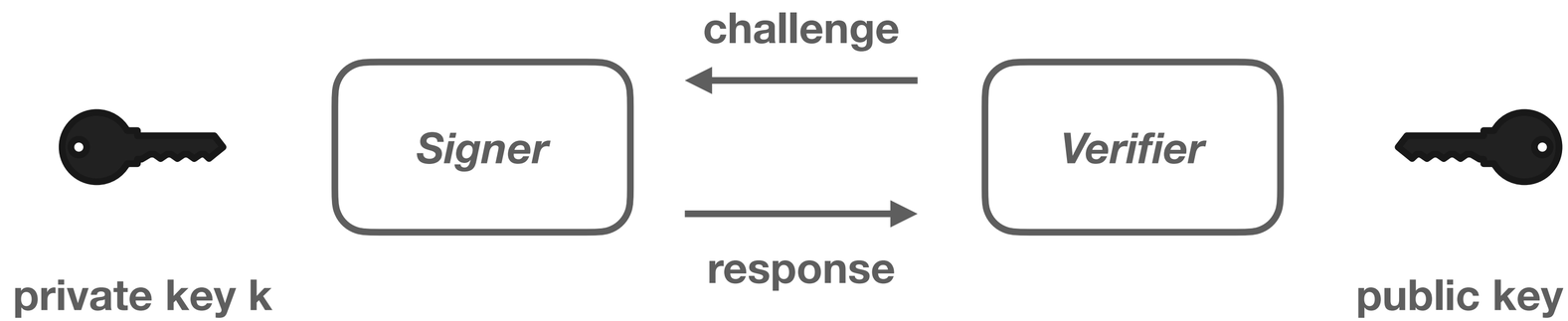


yubico

Authentication

yubico

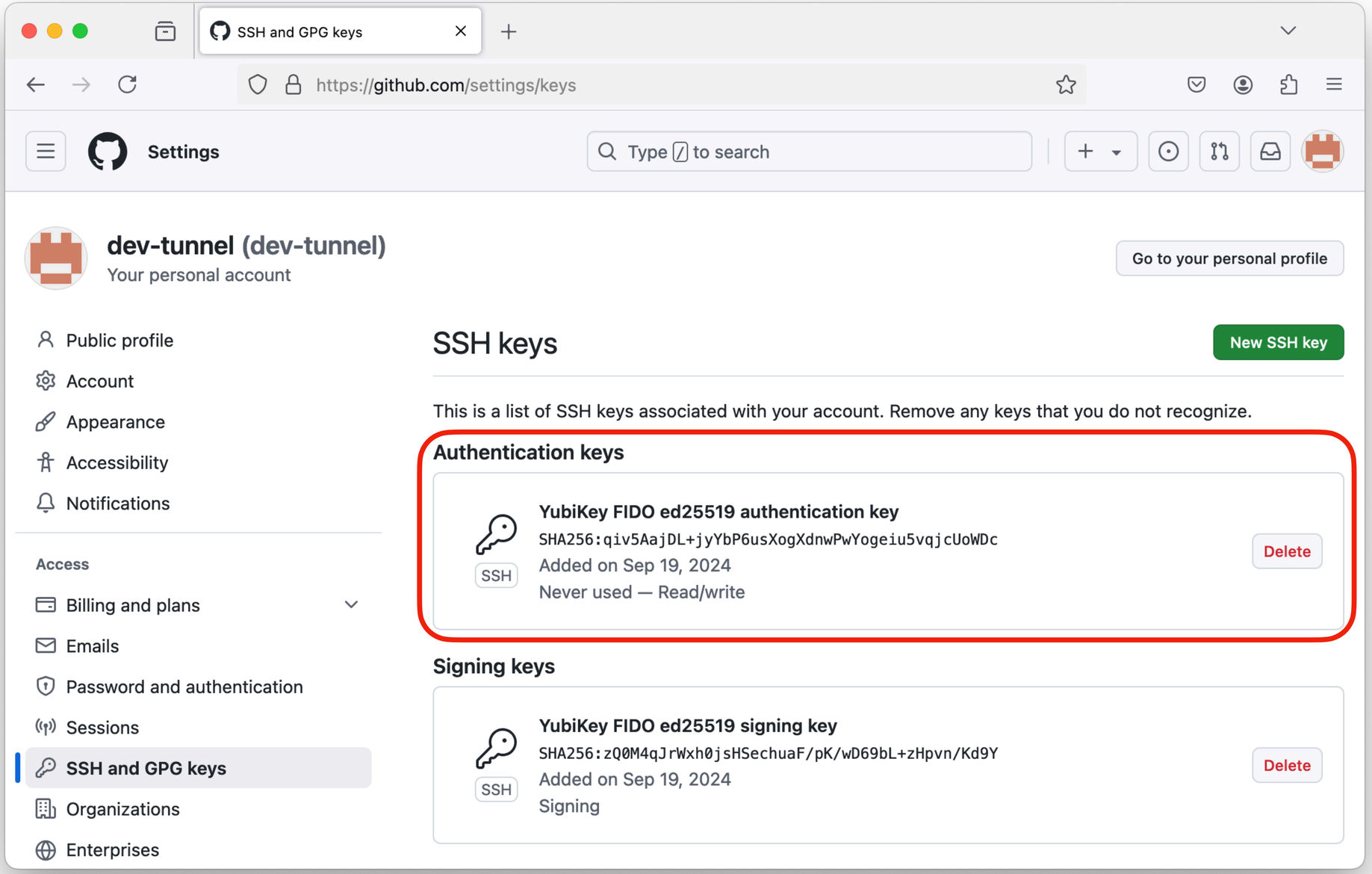
Challenge/Response authentication



response = sign(k, challenge)

result = verify(p, response, challenge)

GitHub Authentication Keys



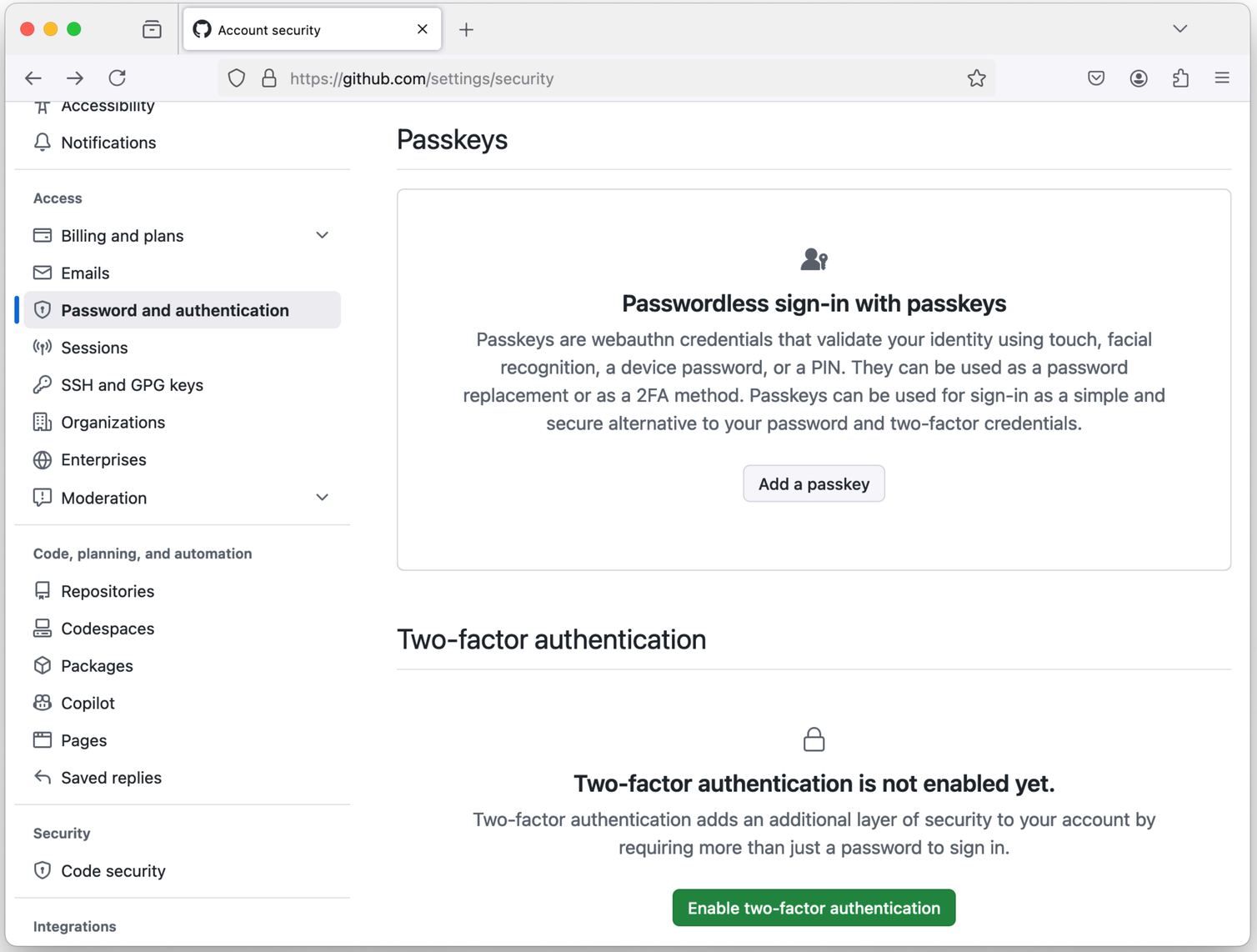
Git Remote Access

```
$ # update current repo
$ git pull
Confirm user presence for key ED25519-SK SHA256:FpybChVXHU/MnwIvOszDxV2yFSbDp9ZkYXxjQ2E+8x0
User presence confirmed
Already up to date.

$ # make some changes and commit
$ vi README.md
$ git add README.md
$ git commit -m'minor fix'
minor fix
 1 file changed, 1 insertion(+), 1 deletion(-)

$ # push changes upstream
$ git push
Confirm user presence for key ED25519-SK SHA256:FpybChVXHU/MnwIvOszDxV2yFSbDp9ZkYXxjQ2E+8x0
Enumerating objects: 9, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 10 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 1.02 KiB | 1.02 MiB/s, done.
Total 5 (delta 4), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (4/4), completed with 2 local objects.
To github.com:johndoe/myproject.git
   3db8f98..8115c84  main -> main
$
```

Secure Access GitHub / GitLab

```
$ # retrieve someone’s GitHub authentication key
$ curl https://github.com/johndoe.keys
sk-ssh-ed25519@openssh.com AAAAGnNrLXNzaC1lZDI1NTE5QG9wZW5zc2guY29tAAAAIJhUXYvdwz3Dx45bWNmxHs1R21mlUm0o63+s4iCzRoFeAAAACnNzaDpnaXRodWI=

$ # test access to GitHub
$ ssh -T git@github.com
Confirm user presence for key ED25519-SK SHA256:FpybChVXHU/MnwIvOszDxV2yFSbDp9ZkYXxjQ2E+8x0
User presence confirmed
Hi johndoe! You've successfully authenticated, but GitHub does not provide shell access.

$ # retrieve someone’s GitLab authentication key
$ curl https://gitlab.com/johndoe.keys
sk-ecdsa-sha2-nistp256@openssh.com AAAAInNrLWVjZHNhLXNoYTItbmlzdHAyNTZAb3Blbn…NzaC5jb20AAAAIbmlzdHOmdpdGxhYg== John Doe (gitlab.com)

$ # test access to GitLab
$ ssh -T git@gitlab.com
Confirm user presence for key ECDSA-SK SHA256:47DEQpj8HBSa+/TImW+5JCeuQeRkm5NMpJWZG3hSuFU
User presence confirmed
Welcome to GitLab, @johndoe!
$
```

Passwordless sign-in with passkeys



yubico

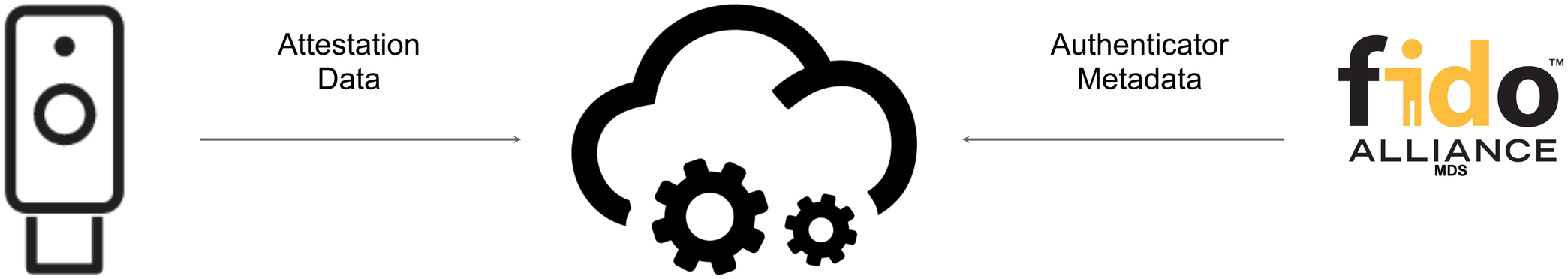
What is a passkey?

- Passkeys are a more secure alternative to passwords
- More secure, because:
  - Passkeys are resistant to phishing
  - Passkeys have no secrets that can be leaked from servers
  - Passkeys are generated automatically, never reused
- Also easier to use:
  - “Sign in with your face, your finger, or your PIN”
  - Optionally, automatically backed up and synced
- Technically, a passkey is a *FIDO2 credential*

Attestation

FIDO Attestation and Metadata

- Attestation provides verifiable evidence as to the authenticator's origin
- Based on a hardware attestation key and certificate
- Use FIDO Alliance Metadata Service to determine provenance
- Implement Allow/Deny lists to filter Authenticators
- Typically used in high-assurance (enterprise) use cases



PIV Key Attestation (proprietary)

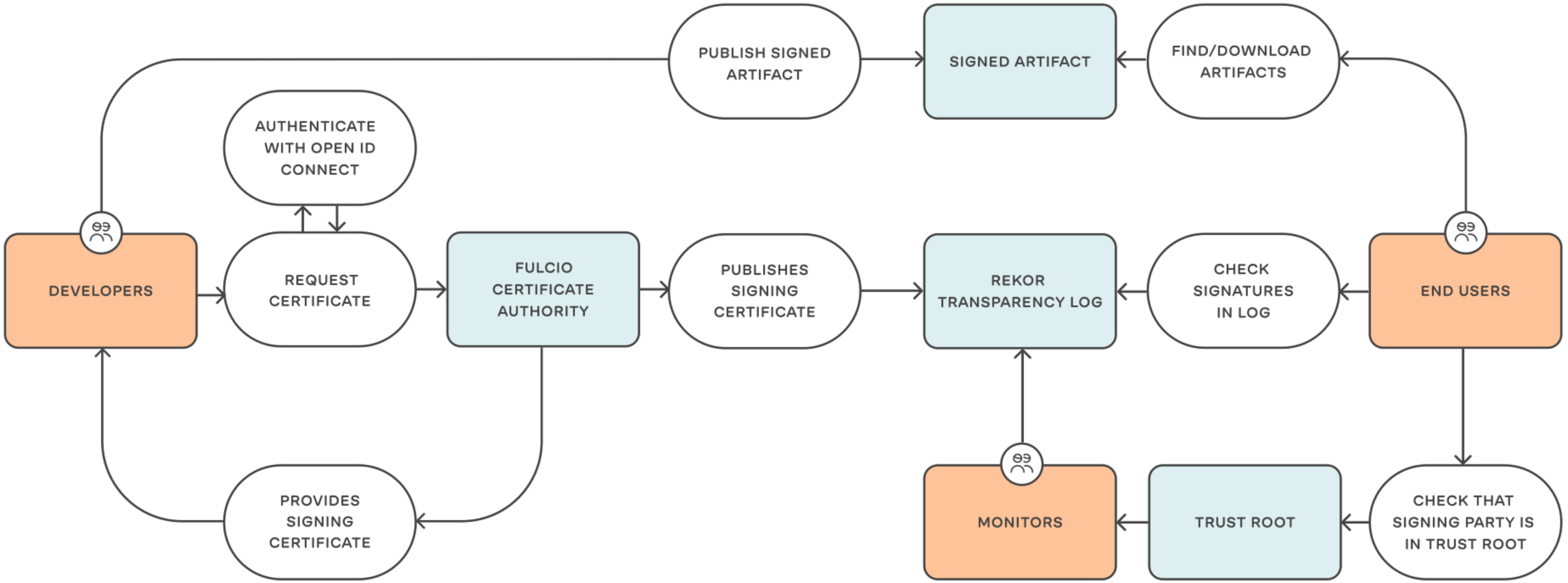
```
$ # generate a new key in the digital signature slot
$ ykman piv keys generate -a eccp256 9c public.pem
Enter a management key [blank to use default key]:
Private key generated in slot 9C (SIGNATURE), public key written to public.pem.
$ # generate a key attestation for the generated key
$ ykman piv keys attest 9c attestation.pem
Attestation certificate for slot 9C (SIGNATURE) written to attestation.pem.
$ # inspect the attestation certificate
$ step certificate inspect attestation.pem
. . .
        Issuer: CN=Yubico PIV Attestation
        Validity
            Not Before: Mar 14 00:00:00 2016 UTC
            Not After : Apr 17 00:00:00 2052 UTC
        Subject: CN=YubiKey PIV Attestation 9c
        Subject Public Key Info:
            Public Key Algorithm: ECDSA
                Public-Key: (256 bit)
                X:
                    c5:da:40:b9:d8:f6:91:29:ea:88:a4:35:e1:8e:b4:
                    16:29:77:1a:8c:c0:1c:af:4d:4c:06:a6:b1:69:04:
                    4a:4f
                Y:
                    99:06:b8:8e:21:cf:dd:eb:85:32:18:a7:d8:8d:99:
                    1d:91:d9:01:a6:d8:89:de:74:96:ba:3b:18:b9:b3:
                    04:64
                Curve: P-256
        X509v3 extensions:
            X509v3 YubiKey Firmware Version:
                5.4.3
            X509v3 YubiKey Serial Number:
                17735646
            X509v3 YubiKey Policy:
                PIN policy: always
                Touch policy: never
            X509v3 YubiKey Formfactor:
                USB-C Keychain
    Signature Algorithm: SHA256-RSA
. . .
$
```

“Keyless” signing

Sigstore

- **“Let’s encrypt” for software signing**
- **Sign software artifacts (packages, Docker images, binary blobs)**
- **Transparency: publish keys and signatures on a public append-only log**
  - Enables auditing to detect bad actors
  - Detect instead of prevent
- **Supports YubiHSM and Yubikeys (PIV)**
- **“Keyless signing”: Use your identity to sign using an ephemeral key**
- **Important to use phishing-resistant MFA at IdP!**

Sigstore Components



https://www.sigstore.dev/img/alt_landscapelayout_overview.svg

Cosign using an HSM

```
$ # Assuming an ECDSA signing key
$ ./cosign/cosign pkcs11-tool list-keys-uris --module-path /usr/local/lib/pkcs11/yubihsm_pkcs11.dylib
Enter PIN for PKCS11 token 'YubiHSM': ********
Listing URIs of keys in slot '0' of PKCS11 module '/usr/local/lib/pkcs11/yubihsm_pkcs11.dylib'
Object 0
  Label: key_cosign
  ID: dead
  URI: pkcs11:token=YubiHSM;slot-id=0;id=%de%ad;object=key_cosign?module-path=/usr/local/lib/pkcs11/yubihsm_pkcs11.dylib
$ # Retrieve the public key from the YubiHSM
$ ./cosign/cosign public-key --key “$KEY" > pub.key
$ # Generate some data to sign
$ echo hello > sample.txt
$ # Sign a blob
$ ./cosign/cosign sign-blob --yes --output-signature=sample.sig --key "$KEY" sample.txt
Enter PIN for key '' in PKCS11 token ‘YubiHSM’:********
Using payload from: sample.txt
WARNING: no x509 certificate retrieved from the PKCS11 token
...
  Note that if your submission includes personal data associated with this signed artifact, it will be part of an immutable record.
  This may include the email address associated with the account with which you authenticate your contractual Agreement.
  This information will be used for signing this artifact and will be stored in public transparency logs and cannot be removed later,
...
tlog entry created with index: 132997744
Wrote signature to file sample.sig
$ # Verify the blob
$ cosign verify-blob --key pub.key --signature sample.sig sample.txt
Verified OK
```

Signing a Docker image

```
$ # a simple Dockerfile
$ cat Dockerfile
FROM alpine
CMD ["echo", "Hello, NSSS!"]
$ # Build the docker image
$ docker build -t joostvandijk327/test-container .
. . .
What's next:
    View a summary of image vulnerabilities and recommendations → docker scout quickview
$ # run in a container
$ docker run joostvandijk327/test-container
Hello, NSSS!

$ # push to a registry
$ docker push joostvandijk327/test-container
Using default tag: latest
The push refers to repository [docker.io/joostvandijk327/test-container]
16113d51b718: Layer already exists
latest: digest: sha256:1f1632a7519f25ce8f79ba931aa9009ae26c8341cc88422cb761b5025877f082 size: 527
$ # sign Docker image
$ cosign sign joostvandijk327/test-container
Generating ephemeral keys...
Retrieving signed certificate...

   The sigstore service, hosted by sigstore a Series of LF Projects, LLC, is provided pursuant to the Hosted Project Tools Terms of Use, available at https://lfprojects.org/policies/hosted-project-to
of-use/.
   Note that if your submission includes personal data associated with this signed artifact, it will be part of an immutable record.
   This may include the email address associated with the account with which you authenticate your contractual Agreement.
   This information will be used for signing this artifact and will be stored in public transparency logs and cannot be removed later, and is subject to the Immutable Record notice at https://lfproj
policies/hosted-project-tools-immutable-records/.

By typing 'y', you attest that (1) you are not submitting the personal data of any other person; and (2) you understand and agree to the statement and the Agreement terms at the URLs listed above.
Are you sure you would like to continue? [y/N] y
…

tlog entry created with index: 133437185
Pushing signature to: index.docker.io/joostvandijk327/test-container
$ # Verify the signature
$ cosign verify --certificate-identity joost.vandijk@yubico.com --certificate-oidc-issuer https://accounts.google.com joostvandijk327/test-container

Verification for index.docker.io/joostvandijk327/test-container:latest --
The following checks were performed on each of these signatures:
  - The cosign claims were validated
  - Existence of the claims in the transparency log was verified offline
  - The code-signing certificate was verified using trusted certificate authority certificates

[{“critical":{"identity":{ … }},”Issuer”:"https://accounts.google.com","Subject":"joost.vandijk@yubico.com"}}]
```

Signing a Docker image (“Keyless”)

```
$ # a simple Dockerfile
$ cat Dockerfile
FROM alpine
CMD ["echo", "Hello, NSSS!”]
$ # Build the docker image
$ docker build -t joostvandijk327/test-container .
. . .
What's next:
    View a summary of image vulnerabilities and recommendations → docker scout quickview
$ # push to a registry
$ docker push joostvandijk327/test-container
Using default tag: latest
The push refers to repository [docker.io/joostvandijk327/test-container]
latest: digest: sha256:1f1632a7519f25ce8f79ba931aa9009ae26c8341cc88422cb761b5025877f082 size: 527
$ # sign Docker image
$ cosign sign joostvandijk327/test-container
Generating ephemeral keys...
Retrieving signed certificate...
. . .
By typing 'y', you attest that (1) you are not submitting the personal data of any other person; and
(2) you understand and agree to the statement and the Agreement terms at the URLs listed above.
Are you sure you would like to continue? [y/N] y
…

tlog entry created with index: 133437185
Pushing signature to: index.docker.io/joostvandijk327/test-container
$ # Verify the signature
$ cosign verify --certificate-identity joost.vandijk@yubico.com --certificate-oidc-issuer https://accounts.google.com \
> joostvandijk327/test-container

Verification for index.docker.io/joostvandijk327/test-container:latest --
The following checks were performed on each of these signatures:
  - The cosign claims were validated
  - Existence of the claims in the transparency log was verified offline
  - The code-signing certificate was verified using trusted certificate authority certificates

[{“critical":{"identity":{ … }},”Issuer”:"https://accounts.google.com","Subject":"joost.vandijk@yubico.com"}}]
```

Conclusion

Key Takeaways

- { Artefact, commit, tag, SBOM, … } signing
  is becoming important in improving supply-chain security

- Securing the signing process:
  - Protect signing keys using secure hardware
  - Or protect IdP accounts with phishing-resistant MFA for “Keyless” singing

Questions?

yubico

yubico

Trust the Net