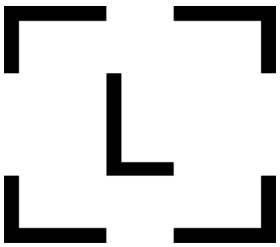
OpenPGP Card Application

Add-on

Ledger SAS



https://github.com/LedgerHQ/app-openpgp

February 14, 2024

Contents

1 License

2 Introduction

2.1 OpenPGP Card Application add-ons summary

2.1.1 Key management:

OpenPGP Application manage 4 keys to Perform Security Operation (PSO) plus 2 for secure channel.

The 4 keys are defined as follow:

- One asymmetric signature private key (RSA or EC), named 'sig'
- One asymmetric decryption private key (RSA or EC), named 'dec'
- One asymmetric authentication private key (RSA or EC), named 'aut'
- One symmetric decryption private key (AES), named 'sym0'

The 3 first asymmetric keys can be either randomly generated on-card or explicitly imported from outside.

The 4th is imported from outside.

It's never possible to retrieve private key from the card.

This add-on specification propose a solution to derive those keys from the master seed managed by the Ledger Token. This allows owner to restore a broken token without the needs to keep track of keys outside the card.

Moreover this add-on specification propose to manage multiple set of the 4 previously described keys.

2.1.2 Keys Slots

To modify the keys slot, just select the corresponding menu from the screen.

2.1.3 Random number generation

OpenPGP Application provides, as optional feature, to generate random bytes.

This add-on specification propose new type of random generation:

- random prime number generation
- seeded random number
- seeded prime number generation

2.1.4 Key Backup

A full key backup mechanism is provided.

3 Ledger OpenPGP Application

3.1 How

3.1.1 Deterministic key derivation

The deterministic key derivation process relies on the **BIP32** scheme. The master install path of the App is set to `/0x80'GPG'`, aka `/80475047`.

Deterministic key derivation maybe activated in:

```
Settings -> Seed Mode -> Set on
```

This activation remains effective until *set off* is selected.

The key management remains the same if seed mode is *on* or *off*. So there is no performance impact when using seeded keys.

Seeded keys are generated as follow:

Step 1:

For a given keys slot n, starting from 1, a seed is first derived with the following path

```
Sn = BIP32_derive (/0x80475047/n)
```

Step 2:

Then specific seeds are derived with the *SHA3-XOF* function for each of the 4 key:

```
Sk[i] = SHA3-XOF(SHA256(Sn \| <key_name> \| int16(i)), length)
```

Where:

- Sn is the dedicated slot seed from step 1.
- key_name is one of 'sig ','dec ', 'aut ', 'sym0', each 4 characters.
- i is the index, starting from 1, of the desired seed (see below)

Step 3:

RSA key generation

Generate two seed Sp, Sq in step2 with:

- i € {1,2}
- length equals to half key size

Generate two prime numbers p, q:

- p = next_prime(Sp)
- q = next_prime(Sq)

Generate RSA key pair as usual:

- choose e
- n = p*q
- d = inv(e) mod (p-1)(q-1)

ECC key generation

Generate one seed Sd in step2 with:

- i = 1
- length equals to curve size

Generate ECC key pair:

- d = Sd
- W = d.G

AES key generation

Generate one seed Sd in step2 with:

- i = 1
- length equals to 16

Generate AES key:

- k = Sk

3.1.2 Deterministic random number

The deterministic random number generation relies on the **BIP32** scheme. The master install path of the App is set to `/0x80'GPG'`, aka `/80475047`.

Random prime number generation:

For a given length L:

- generate random number r of L bytes.
- generate rp = next_prime(r)
- return rp

Seeded random number:

For a given length L and seed S:

- generate Sr = BIP32_derive(/0x80475047/0x0F0F0F0F)
- generate r = SHA3-XOF(SHA256(Sr | 'rnd' | S), L)
- return r

Seeded prime number generation:

For a given length L and seed S:

- generate r as for "Seeded random number"
- generate rp = next_prime(r)
- return rp

3.1.3 Key Backup & Restore

In order to backup/restore private key the commands put_data and get_data accept the tags:

- B6 (signature key)
- B8 (encryption key)
- A4 (authentication).

put_data command accept the exact output of get_data. The get_data command return both the public and private key.

For security and confidentiality, private key is returned encrypted in AES. The key used is derived according to previously described AES key derivation with name 'key'.

The data payload is formatted as follow:

size	Description
4	OS Target ID
4	API Level
4	compliance Level
4	public key size
var	public key
4	private key size
var	encrypted private key

3.2 APDU Modification

3.2.1 Key Slot management

Key slots are managed by data object *01F1* and *01F2* witch are manageable by PUT/GET DATA command as for others DO and organized as follow.

On application reset, the *01F2* content is set to *Default Slot* value of *01F1*.

01F1:

bytes	Description	R/W
1	Number of slot	R
2	Default slot	R/W
3	Allowed slot selection method	R/W

Byte 3 is endoced as follow:

b8	b7	b6	b5	b4	b3	b2	b1	Meaning
-	-	-	-	-	-	-	x	selection by APDU
-	-	-	-	-	-	x	-	selection by screen

01F2:

bytes	Description	R/W
1	Current slot	R/W

01F0:

bytes	Description	R/W
1-3	01F1 content	R
4	01F2 content	R

Access Conditions:

DO	Read	Write
01F0	Always	Never
01F1	Always	Verify PW3
01F2	Always	Verify PW2

3.2.2 Deterministic key derivation

P2 parameter of GENERATE ASYMMETRIC KEY PAIR is set to (hex value):

- 00 for true random key generation
- 01 for seeded random key

3.2.3 Deterministic random number

P1 parameter of GET CHALLENGE is a bit-field encoded as follow:

b8	b7	b6	b5	b4	b3	b2	b1	Meaning
-	-	-	-	-	-	-	x	seeded random
-	-	-	-	-	-	x	-	prime random

When *seeded mode* is set, data field contains the seed and P2 contains the length of random bytes to generate.

3.3 Other minor add-on

GnuPG use both fingerprints and serial number to identify key on card. So, the put_data command is able to modify the AID file with '4F' tag. In that case the data field shall be 4 bytes length and shall contain the new serial number. '4F' is protected by PW3 (admin) PIN.