Ledger Device for Monero

v0.8

Cédric Mesnil (cedric@ledger.fr)

Ledger SAS

January 30, 2018

Contents

1 License

Author: Cédric Mesnil <cslashm@gmail.com>

License:

2 Introduction

We want to enforce key protection, transaction confidentiality and transaction integrity against potential malware on the Host. To achieve that we propose to use a Ledger NanoS as a 2nd factor trusted device. Such a device has small amount of memory and is not capable of holding the entire transaction or building the required proofs in RAM. So we need to split the process between the host and the NanoS. This draft note explain how.

Moreover this draft note also anticipates a future client feature and proposes a solution to integrate the PR2056 for sub-address. This proposal is based on kenshi84 fork, branch sub-address-v2.

To summarize, the signature process is:

. Generate a TX key pair (r, R)

. Process Stealth Payment ID

. For each input T_{in} to spend:

- Compute the input public derivation data $\mathfrak{D}_{\mathrm{in}}$
- Compute the spend key (x_{in}, P_{in}) from R_{in} and b
- Compute the key image I_{in} of x_{in}

. For each output T_{out} :

- Compute the output secret derivation data $\mathfrak{D}_{\mathrm{out}}$
- Compute the output public key P_{out}

. For each output T_{out} :

- compute the range proof
- blind the amount

. Compute the final confidential ring signature

. Return TX

3 Notation

Elliptic curve points, such as pubic keys, are written in italic upper case, and scalars, such as private keys, are written in italic lower case:

- spk : protection key
- (r, R) : transaction key pair
- (a, A) (b, B) : sender main view/spend key pair
- (c, C) (d, D) : sender sub view/spend key pair
- A_{out} B_{out} : receiver main view/spend public keys
- C_{out} D_{out} : receiver sub view/spend public key
- `keccak` : 2nd group generator, such $H = h.G$ and `keccak` is unknown
- v : amount to send/spent
- k : secret amount mask factor
- C_v : commitment to a with v such $C_v = k.G + v.H$
- α_{in} : secret co-signing key for ith input
- x_{in} : secret signing key for ith input
- P_{in} : public key of ith input
- P_{out} : public key of ith output
- $\mathfrak{D}_{\mathrm{out}}$ $\mathfrak{D}_{\mathrm{in}}$: first level derivation data

Hash and encryption function:

- `AES` : [k](m) AES encryption of m with key k
- $\texttt{AES}^{-1}$: [k](c) AES decryption of c with key k

Others:

- $PayID$: Stealth payment ID
- `ENC_PAYMENT_ID_TAIL` : 0x82

4 State Machine

TBD

5 Commands overview

5.1 Introduction

Hereafter are the code integration and application specification.

The commands are divided in three sets:

- Provisioning
- Low level crypto command
- High level transaction command

The low level set is a direct mapping of some crypto Monero function. For such command the Monero function will be referenced.

The high level set encompasses functions that handle the confidential/sensitive part of full transaction

5.2 Common command format

All command follow the generic ISO7816 command format, with the following meaning:

byte	length	description
CLA	01	Always zero '00'
INS	01	Command
P1	01	Sub command
P2	01	Command/Sub command counter
LC	01	byte length of data
data +	01 ———+ var	options ——————— ———+

When a command/sub-command can be sent repeatedly, the counter must be increased by one at each command. The flag `last sub command indicator` must be set to indicate another command will be sent.

Common option encoding

`x--------`	Last sub command indicator
`1--------`	More identical subcommand
`0--------`	forthcoming
	Last sub command

6 Provisioning

6.1 Overview

There is no provisioning in a standard setup. Both key pairs (a, A) and (b, B) should be derived under BIP44 path.

The general BIP44 path is :

```
/ purpose' / coin_type' / account' / change / address_index
```

and is defined as follow for any Monero main address:

```
/44'/128'/account'/0/0
```

so in hexa:

```
/0x8000002C/0x80000080/0x8......./0x00000000/0x00000000
```

The *address_index* is set to *0* for the main address and will be used as sub-address index according to kenshi84 fork.

In case an already existing key needs to be transferred, an optional dedicated command may be provided. As there is no secure messaging for now, this transfer shall be done from a trusted Host. Moreover, as provisioning is not handled by Monero client, a separate tool must be provided.

6.2 Commands

6.2.1 Put keys

Description

Put sender key pairs.

The application shall:

check $A == a.G$
check $B == b.G$
store a, A, b, B

Command

CLA	INS	P1	P2	LC	data description
00	32	00	00	80	

Command data

Length	Value
01	00
20	a
20	A
20	b
20	B
5f	Base58 encoded public key

Response data

Length	Value

6.2.2 Get Public Key

Command

CLA	INS	P1	P2	LC	data description
00	30	01	00	80	

Command data

Length	Value
01	00

Response data

Length	Value
5f	Base58 encoded public key

6.2.3 Get Secret Keys

Command

CLA	INS	P1	P2	LC	data description
00	30	02	00	80	

Command data

Length	Value
01	00

Response data

Length	Value
20	Encrypted view key
20	Encrypted send key

7 Low level crypto commands

7.1 Overview

TODO

7.2 Commands

7.2.1 Derive Subaddress Public Key

Monero

crypto_ops::derive_subaddress_public_key

Description

compute $\widetilde{\mathfrak{D}_{\text{in}}} = \texttt{AES}^{-1}[spk](\widetilde{\mathfrak{D}_{\text{in}}})$
compute $s = \texttt{keccak}(\mathfrak{D}_{\text{in}} \mid \text{varint}(index))$
compute $s = s \;\%\; \#\texttt{n}$
compute $P\text{'} = P$ - $s.G$

return P'

Command

CLA	INS	P1	P2	LC	data description
00	46	00	00	00	

Command data

Length	Value
01	00
32	public key P
32	encrypted derivation key $\widetilde{\mathfrak{D}_{\text{in}}}$
04	index *index*

Response data

Length	Value
32	sub public key P'

7.2.2 Get Subaddress Spend Public Key

Monero

get_subaddress_spend_public_key

Description

get_subaddress_secret_key:

compute $s = \texttt{keccak}(\text{“SubAddr”} \mid A \mid \mathit{index}\ )$
compute $x = s \ \% \ \#\texttt{n}$

then:

compute $d = B + x.G$

return d

Command

CLA	INS	P1	P2	LC	data description
00	4a	00	00	00	

Command data

Length	Value
01	00
08	index (Major.minor) *index*

Response data

Length	Value
32	sub spend public key d

7.2.3 Get Subaddress

Monero

Description

get_subaddress_secret_key:

compute $s = \texttt{keccak}(\text{“SubAddr”} \mid A \mid \mathit{index}\ )$

compute $x = s \ \% \ \#\mathtt{n}$

then:

compute $d = B + x.G$
compute $c = A.d$

return c, d

Command

CLA	INS	P1	P2	LC	data description
00	48	00	00	00	

Command data

Length	Value
01	00
08	index (Major.minor) *index*

Response data

Length	Value
32	sub view public key c
32	sub spend public key d

7.2.4 Get Subaddress Secret Key

Monero

get_subaddress_secret_key

Description

compute $x = \mathtt{AES}^{-1}[spk](\widetilde{x})$
compute $s = \mathtt{keccak}(\text{“SubAddr”} \mid x \mid index\ )$
compute $d = s \ \% \ \#\mathtt{n}$
compute $\widetilde{d_i} = \mathtt{AES}^{-1}[spk](d)$

return $\widetilde{d_i}$

Command

CLA	INS	P1	P2	LC	data description
00	4c	00	00	39	

Command data

Length	Value
01	00
32	secret key $\widetilde{x}$
08	index (Major.minor) *index*

Response data

Length	Value
32	sub secret key $\widetilde{d_i}$

7.2.5 Verify Keys

Monero

Description

Command

CLA	INS	P1	P2	LC	data description
00	26	00	00	00	

Command data

Length	Value
01	00
00	

Response data

Length	Value
00	
00	

7.2.6 Scalarmult Key

Monero

rct::scalarmultKey

Description

compute $x = \texttt{AES}^{-1}[spk](\widetilde{x})$
compute $xP = x.P$

return xP

Command

CLA	INS	P1	P2	LC	data description
00	42	00	00	00	

Command data

Length	Value
01	00
32	public key P
32	secret key $\widetilde{x}$

Response data

Length	Value
00	new public key xP

7.2.7 Scalarmult Base

Monero

rct::scalarmultBase

Description

compute $x = \texttt{AES}^{-1}[spk](\widetilde{x})$
compute $xG = x.G$

return xG

Command

CLA	INS	P1	P2	LC	data description
00	44	00	00	00	

Command data

Length	Value
01	00
32	secret key $\widetilde{x}$

Response data

Length	Value
00	
00	new public key xG

7.2.8 Secret Add

Monero

Description

compute $x = \texttt{AES}^{-1}[spk](\widetilde{x})$
compute $x = \texttt{AES}^{-1}[spk](\widetilde{x})$
compute $x = x + x$
compute $\widetilde{x} = \texttt{AES}[spk](x)$

return $\widetilde{x}$

Command

CLA	INS	P1	P2	LC	data description
00	3c	00	00	00	

Command data

Length	Value
01	00
32	secret key $\widetilde{x}$
32	secret key $\widetilde{x}$

Response data

Length	Value
32	secret key $\widetilde{x}$

7.2.9 Generate Keys

Monero

Description

generate x
compute $xP = x.P$
compute $\widetilde{x} = \texttt{AES}[spk](x)$

return P, $\widetilde{x}$

Command

CLA	INS	P1	P2	LC	data description
00	40	00	00	00	

Command data

Length	Value
01	00

Response data

Length	Value
00	public key P
00	encrypted secret key $\widetilde{x}$

7.2.10 Generate Key Derivation

Monero

Description

compute $x = \texttt{AES}^{-1}[spk](\widetilde{x})$
compute $d = x.P$
compute $\mathfrak{D}_{\text{in}} = 8.d$
compute $\widetilde{\mathfrak{D}_{\text{in}}} = \texttt{AES}[spk](\mathfrak{D}_{\text{in}})$

return $\widetilde{\mathfrak{D}_{\text{in}}}$

Command

CLA	INS	P1	P2	LC	data description
00	32	00	00	00	

Command data

Length	Value
01	00
32	public key P
32	secret key $\widetilde{x}$

Response data

Length	Value
32	encrypted key derivation $\widetilde{\mathfrak{D}_{\text{in}}}$

7.2.11 Derivation To Scalar

Monero

derivation_to_scalar

Description

> compute $\mathfrak{D}_{\text{in}} = \texttt{AES}^{-1}[spk](\widetilde{\mathfrak{D}_{\text{in}}})$
> compute $s = \texttt{keccak}(\mathfrak{D}_{\text{in}} \mid \text{varint}(index))$
> compute $s = s \ \% \ \#\texttt{n}$
> compute $\widetilde{s} = \texttt{AES}[spk](s)$

return $\widetilde{s}$

Command

CLA	INS	P1	P2	LC	data description
00	34	00	00	00	

Command data

Length	Value
01	00
32	encrypted key derivation $\widetilde{\mathfrak{D}_{\text{in}}}$
04	index

Response data

Length	Value
32	encrypted scalar $\widetilde{s}$

7.2.12 Derive Secret Key

Monero

derive_scecret_key

Description

> compute $\widetilde{\mathfrak{D}_{\text{in}}} = \texttt{AES}^{-1}[spk](\widetilde{\mathfrak{D}_{\text{in}}})$
> compute $x = \texttt{AES}^{-1}[spk](\widetilde{x})$

derivation_to_scalar:

> compute $s = \texttt{keccak}(\mathfrak{D}_{\text{in}} \mid \text{varint}(index))$
> compute $s = s \ \% \ \#\texttt{n}$

then:

> compute $x' = (x{+}s) \ \% \ \#\texttt{n}$
> compute $\widetilde{x}' = \texttt{AES}[spk](x)$

return $\widetilde{x}$

Command

CLA	INS	P1	P2	LC	data description
00	38	00	00	00	

Command data

Length	Value
01	00
32	encrypted key derivation $\widetilde{\mathfrak{D}_{\text{in}}}$
04	index
32	encrypted secret key $\widetilde{x}$

Response data

Length	Value
32	encrypted drevived secret key $\widetilde{x}'$

7.2.13 Derive Public Key

Monero

derive_public_key

Description

compute $\widetilde{\mathfrak{D}_{\text{in}}} = \texttt{AES}^{-1}[spk](\widetilde{\mathfrak{D}_{\text{in}}})$

derivation_to_scalar:

compute $s = \texttt{keccak}(\mathfrak{D}_{\text{in}} \mid \text{varint}(index))$
compute $s = s \ \% \ \#\texttt{n}$

then:

compute $P\text{'} = P{+}s.G$

return P

Command

CLA	INS	P1	P2	LC	data description
00	36	00	00	00	

Command data

Length	Value
01	00
32	encrypted key derivation $\widetilde{\mathfrak{D}_{\text{in}}}$
04	index
32	encrypted secret key P

Response data

Length	Value
32	public key P'

7.2.14 Secret Key To Public Key

Monero

secret_key_to_public_key

Description

compute $x = \texttt{AES}^{-1}[spk](\widetilde{x})$
compute $P = x.G$

return P

Command

CLA	INS	P1	P2	LC	data description
00	30	00	00	00	

Command data

Length	Value
01	00
32	encrypted secret key $\widetilde{x}$

Response data

Length	Value
32	public key P

7.2.15 Generate Key Image

Monero

generate_key_image

Description

compute $x = \texttt{AES}^{-1}[spk](\widetilde{x})$
compute $s = \texttt{keccak}(P')$
compute $P' = \text{ge\_from\_fe}(s)$
compute $Img(P) = x.P'$

return $Img(P)$

Command

CLA	INS	P1	P2	LC	data description
00	3a	00	00	00	

Command data

Length	Value
01	00
32	public key P
32	secret key $\widetilde{x}$

Response data

Length	Value
32	key image $Img(P)$

8 High Level Transaction command

8.1 Transaction process overview

The transaction is mainly generated in construct_tx_and_get_tx_key (or construct_tx) and construct_tx_with_tx_key functions.

First, a new transaction keypai , (r, R) is generated.

Then, the stealth payment id is processed if any.

Then, for each input transaction to spend, the input key image is retrieved.

Then, for each output transaction, the destination key and the change address are computed.

Once T_{in} and T_{out} keys are set up, the genRCT/genRctSimple function is called.

First a commitment C_v to each v amount and its associated range proof are computed to ensure the v amount confidentiality. The commitment and its range proof do not imply any secret and generate C_v, k such $C_v = k.G + v.H$.

Then k and v are blinded by using the $\mathcal{AK}_{\text{amount}}$ which is only known in an encrypted form by the host.

After all commitments have been setup, the confidential ring signature happens. This signature is performed by calling proveRctMG which then calls MLSAG_Gen.

At this point the amounts and destination keys must be validated on the NanoS. This information is embedded in the message to sign by calling get_pre_mlsag_hash, prior to calling ProveRctMG. So the get_pre_mlsag_hash function will have to be modified to serialize the rv transaction to NanoS which will validate the tuple <amount,dest> and compute the prehash. The prehash will be kept inside NanoS to ensure its integrity. Any further access to the prehash will be delegated.

Once the prehash is computed, the proveRctMG is called. This function only builds some matrix and vectors to prepare the signature which is performed by the final call MLSAG_Gen.

During this last step some ephemeral key pairs are generated : α_{in}, $\alpha_{in}.G$. All α_{in} must be kept secret to protect the x in keys. Moreover we must avoid signing arbitrary values during the final loop.

In order to achieve this validation, we need to approve the original destination address A_{out}, which is not recoverable from P out . Here the only solution is to pass the original destination with the k, v. (Note this implies to add all A_{out} in the rv structure). So with A_{out}, we are able to recompute associated D_{out} (see step 3), unblind k and v and then verify the commitment $C_v = k.G + v.H$. If C_v is verified and user validate A_{out} and v, $\mathcal{L}$ is updated and we process the next output.

8.2 Transaction Commands

8.2.1 Open TX

Monero

Description

Open a new transaction. Once open the device impose a certain order in subsequent commands:

- OpenTX
- Stealth
- Blind *
- Initialize MLSAG-prehash
- Update MLSAG-prehash *
- Finalize MLSAG-prehash
- MLSAG prepare
- MLSAG hash *
- MLSAG sign
- CloseTX

During this sequence low level API remains available, but no other transaction can be started until the current one is finished or aborted.

Command

CLA	INS	P1	P2	LC	data description
00	70	01	cnt	var	

Command data

+——+————————————————————+ | Length | Value |
+========+==
| 01 | options | +——+————————————————————+
+——+————————————————————+

Response data

+——+————————————————————+ | Length | Value |
+========+==
+——+————————————————————+

8.2.2 Set Signature Mode

Monero

Description

Set the signature to ‘fake’ or ‘real’. In fake mode a random key is used to signed the transaction and no user confirmation is requested.

Command

CLA	INS	P1	P2	LC	data description
00	72	01	cnt	var	

Command data

Length	Value
01	options
01	‘fake’ or ‘real’

Response data

+——+————————————————————+ | Length | Value |
+========+==
+——+————————————————————+

8.2.3 Blind Amount and Mask

Monero

Description

compute $\mathcal{AK}_{\text{amount}} = \mathtt{AES}^{-1}[spk](\widetilde{\mathcal{AK}_{\text{amount}}})$
compute $\widetilde{k} = k + \mathtt{keccak}(\mathcal{AK}_{\text{amount}})$

compute $\widetilde{v} = k + \texttt{keccak}(\texttt{keccak}(\mathcal{AK}_{\text{amount}}))$
update $\mathcal{L}$: $\texttt{H}_{\texttt{update}}(v \mid k \mid \mathcal{AK}_{\text{amount}})$
if option ‘last’ is set:
finalize $\mathcal{L}$

The application returns $\widetilde{v}$, $\widetilde{k}$

Command

CLA	INS	P1	P2	LC	data description
00	7E	01	cnt	var	

Command data

Length	Value
01	options
20	value v
20	mask k
20 |	encrypted private derivation data $\widetilde{\mathcal{AK}}_{\text{amount}}$

Response data

Length	Value
20	blinded value $\widetilde{v}$
20	blinded mask $\widetilde{k}$

8.2.4 Pre Hash

8.2.4.1 Initialize MLSAG-prehash

Description

During the first step, the application updates the $\mathcal{H}$ with the transaction header:

Initialize $\mathcal{C}$
Initialize $\mathcal{L}$’
Initialize $\mathcal{H}$: $\texttt{H}_{\texttt{update}}(header)$

Command

CLA	INS	P1	P2	LC	data description
00	82	01	cnt	var	

Command data

if `cnt==1` :

Length	Value
01	options
01	type
varint	txnFee

if `cnt>1` :

Length	Value
20	pseudoOut

8.2.4.2 Update MLSAG-prehash

Description

On the second step the application receives amount and destination and check values. It also re-compute the $\mathcal{L}$ value to ensure consistency with steps 3 and 4. So for each command received, do:

compute $\mathfrak{D}_{\text{in}} = 8.r.A_{out}$
compute $k = \widetilde{k}$ - `keccak`$(\mathfrak{D}_{\text{in}})$
compute $v = \widetilde{k}$ - `keccak`(`keccak`$(\mathfrak{D}_{\text{in}})$)
check $C_v = k.G + v.H$

ask user validation of A_{out}, B_{out}
ask user validation of v

update $\mathcal{C}$: $\mathtt{H}_{\mathtt{update}}(C_v)$
update $\mathcal{L}$' : $\mathtt{H}_{\mathtt{update}}(v \mid k \mid \mathfrak{D}_{\text{in}})$

update $\mathcal{H}$: $\mathtt{H}_{\mathtt{update}}(ecdhInfo)$

Command

CLA	INS	P1	P2	LC	data description
00	82	02	cnt	var	

Command data

Length	Value
01	options
20	Real destination view key A_{out}

Length	Value
20	Real destination spend key B_{out}
20	C_v of v,k

40	one serialized ecdhInfo :
	{
	bytes[32] mask ($\widetilde{k}$)
	bytes[32] amount ($\widetilde{v}$)
	}

8.2.4.3 Finalize MLSAG-prehash

Description

Finally the application receives the last part of data:

> finalize $\mathcal{L}$' : $\texttt{H}_{\texttt{finalize}}()$
> check $\mathcal{L} == \mathcal{L}$'
>
> finalize $\mathcal{C}$: $\texttt{H}_{\texttt{finalize}}()$
> compute $\mathcal{C}$' $= \texttt{H}_{\texttt{finalize}}(commitment_0.Ct|commitment_1.Ct|.....)$ |
> check $\mathcal{C} == \mathcal{C}$'
>
> finalize $\mathcal{H}$: $\texttt{H}_{\texttt{finalize}}(commitments)$
> compute $\mathcal{H} = \texttt{keccak}(message \mid \mathcal{H} \mid proof)$

Keep $\mathcal{H}$

Command

CLA	INS	P1	P2	LC	data description
00	82	03	00	var	

Command data

not last:

Length	Value
01	options

20	one serialized commitment :
	{
	bytes[32] mask (C_v)
	}

last:

Length	Value
01	options
20	message (rctSig.message)
20	proof (proof range hash)

Response data

Length	Value

8.2.5 MLSAG

8.2.5.1 MLSAG prepare

Description

Generate the matrix ring parameters:

generate α_{in} ,
compute $\alpha_{in}.G$
if real key:
    check the order of H_i
    compute $x_{in} = \mathtt{AES}^{-1}[spk](\widetilde{x_{in}})$
    compute $II_{in} = x_{in}.H_i$
    compute $\alpha_{in}.H_i$
    compute $\widetilde{\alpha_{in}} = \mathtt{AES}[spk](\alpha_{in})$

return $\widetilde{\alpha_{in}}$, $\alpha_{in}.G$ [$\alpha_{in}.H_i$, II_{in}]

Command

CLA	INS	P1	P2	LC	data description
00	84	01	cnt	var	

Command data

for real key:

Length	Value
01	options
20	point

Length	Value
20	secret spend key $\widetilde{x_{in}}$

for random ring key

Length	Value
01	options

Response data

for real key:

Length	Value
20	$\alpha_{in}.H_i$
20	$\alpha_{in}.G$
20	II_{in}
20	encrypted α_{in} : $\widetilde{\alpha_{in}}$

for random ring key

Length	Value
20	$\alpha_{in}.H_i$
20	$\alpha_{in}.G$

8.2.5.2 MLSAG hash

Description

Compute the last matrix ring parameter:

> replace the first 32 bytes of `inputs` by the previously computed MLSAG-prehash
> compute c = `keccak`(`inputs`)

Command

CLA	INS	P1	P2	LC	data description
00	84	02	00	var	

Command data

Length	Value
01	options
var	inputs

Response data

Length	Value

8.2.5.3 MLSAG sign

Description

Finally compute all signatures:

compute $\alpha_{in} = \texttt{AES}^{-1}[spk](\widetilde{\alpha_{in}})$
compute $x_{in} = \texttt{AES}^{-1}[spk](\widetilde{x_{in}})$
compute $ss = (\alpha_{in}$ - c * x_{in}) % l

return ss

Command

CLA	INS	P1	P2	LC	data description
00	84	03	cnt	var	

Command data

Length	Value
01	options
20	$\widetilde{x_{in}}$
20	$\widetilde{\alpha_{in}}$

Response data

Length	Value
20	signature ss

9 Conclusion

This draft note explains how to protect Monero transactions of the official client with a NanoS. According to the latest SDK, the necessary RAM for global data is evaluated to around 0.8 Kilobytes for a transaction with one output and 1,7 Kilobytes for a transaction with ten outputs. The proposed NanoS interaction should be enhanced with a strong state machine to avoid multiple requests for the same data and limit any potential cryptanalysis.

9.1 References

[1] https://github.com/monero-project/monero/tree/v0.10.3.1
[2] https://github.com/monero-project/monero/pull/2056
[3] https://github.com/kenshi84/monero/tree/subaddress-v2
[4] https://www.reddit.com/r/Monero/comments/6invis/ledger_hardware_wallet_monero_integration
[5] https://github.com/moneroexamples