Ledger Device for Monero

v0.8

Cédric Mesnil (cedric@ledger.fr)

Ledger SAS

September 26, 2017

Contents

1 License

Author: Cedric Mesnil <cslashm@gmail.com>

License:

2 Introduction

We want to enforce key protection, transaction confidentiality and transaction integrity against some potential malware on the Host. To achieve that we propose to use a Ledger NanoS as a 2nd factor trusted device. Such device has small amount of memory and it is not possible to get the full Monero transaction on it or to built the different proofs. So we need to split the process between Host and 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 $\mathfrak{D}_{\text{in}}$
- 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 $\mathfrak{D}_{\text{out}}$
- 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
- h : 2nd group generator, such $H = h.G$ and h 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
- $\mathfrak{D}_{\text{out}}$ $\mathfrak{D}_{\text{in}}$: first level derivation data

Hash and encryption function:

- $\mathtt{H_p}$: point to point hash function
- $\mathtt{H_s}$: scalar to scalar hash function
- $\mathtt{H_{p\to s}}$: point to scalar hash function
- $\mathtt{AES}$: [k](m) AES encryption of m with key k
- $\mathtt{AES}^{-1}$: [k](c) AES decryption of c with key k

Others:

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

Some helper functions:

- `DeriveDH` : Derivation function: scalar,point $\to$ point
- `DerivePub` : Derivation function: scalar,point $\to$ point
- `DerivePriv` : Derivation function: point, scalar $\to$ scalar

4 State Machine

TBD

5 Provisioning

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. Moreoever, as provisioning is not handled by Monero client, a separate tool must be provided.

5.1 Put keys

5.1.1 Description

Put sender key pairs.

The application shall:

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

5.1.2 Commands

5.1.2.1 Put Keys

Command

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

Command data

Length	Value
20	*a*
20	*A*
20	*b*
20	*B*

Response data

Length	Value

6 Integration

Hereafter are the code integration and application specification. Each step is detailed with client code references and matching device commands.

Note that in this process, the device will first see all input keys, then all output keys, and will have to sign some data related to those seen keys. In order to enforce that signed data is really bound to the input and out keys processed during the first steps, an hash - named $\mathcal{L}$ - is computed during the key processing then verified during the sign process. If the hash does not match, the device will refuse to sign the transaction.

In the same way, we create an hash over commitment - named $\mathcal{C}$ - to ensure that values between commitment validation and signature are the same.

Note that $\mathcal{L}$ is required because the mlsag-prehash does not cover the ephemeral destination key.

6.1 Common commands 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	options +-------+
	var	

When a command/sub-command can be sent repeatedlyn 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 to comme
`0--------`	Last sub command

6.2 Start transaction

6.2.1 Code Reference

The transaction key is generated in cryptonote_tx_utils.cpp line 169.

This generation is simply delegated to NanoS which keeps the secret key. During this step, the NanoS also computes a secret key SPK (Secret Protection Key) to encrypt some confidential data for which the storage is delegated to the Host. Optionally the secret transaction key may be returned encrypted by SPK to be used later. Moreover the secret transaction key is discarded by the token at the end of the transaction and can not be retrieved if not saved by host. If the secret transaction key needs to be saved, the SPK is generated in a deterministic way.

Finally, an optional exchange is done to override the TX public key in case a Sub-Address is used

6.2.2 Description

This is the very first APDU initiating a transaction signature. When receiving this command, the application resets its internal state and aborts any unfinished previous transaction. After resetting, the application generates a new Ed25519 Transaction Key pair (r,R).

If the account parameter is different from zero, it specifies the BIP44 *account* path:

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

The account value shall be greater than *0x80000000*.

If the account parameter is equal to *zero*, and if external keys have been set with the optional *Put Key* command, this keys are used.

If `Keep r` indicator is set, the application derives the protection key *spk* with:

$spk = \text{DeriveAES}(R,a,b)$
$\widetilde{r} = \text{AES}[spk](r)$

If `Keep r` indicator is not set, the application derives a random protection key *spk*.

Finally the application returns R and optionally $\widetilde{r}$

A second command can be sent just after to switch to a destination sub-address. When receiving this command, the application must compute the new transaction public key according to the provided D_{out} sub-address.

$R = r.D_{out}$

The application returns the the new R.

6.2.3 Commands

6.2.3.1 Open Transaction

Command

CLA	INS	P1	P2	LC	data description
00	50	01	00	05	

Command data

Length	Value
01	options
04	account

option encoding

`-------x`	`Keep r` indicator
`-------1`	Private TX key must be returned encrypted
`-------0`	Private TX key must not be returned

Response data

Length	Value
20	R key
20	$\widetilde{r}$ if *Keep r* indicator is set

6.2.3.2 Sub Transaction

Command

CLA	INS	P1	P2	LC	data description
00	50	02	00	00	

Command data

Length	Value
20	Sub address D_{out}

Response data

Length	Value
20	R key

6.3 Process Stealth Payment

6.3.1 Code Reference

6.3.2 Description

This command handles the Stealth Payment Encryption.

The application encrypts the paymentID with the following steps:

compute $\mathfrak{D}_{\text{out}} = \texttt{DeriveDH}(r, A_{out})$
compute $s = \texttt{H}_\texttt{s}(\texttt{PointToBYtes}(\mathfrak{D}_{\text{out}}) \mid \texttt{ENC\_PAYMENT\_ID\_TAIL})$
compute $\widetilde{PayID} = PayID \oplus s\texttt{[0:8]}$

6.3.3 Commands

6.3.3.1 Stealth

Command

CLA	INS	P1	P2	LC	data description
00	52	01	00	41	

Command data

Length	Value
01	option
20	View destination address A_{out}
08	clear payment ID $PayID$

Response data

Length	Value
08	encrypted payment ID $\widetilde{PayID}$

6.4 Process Input Transaction Keys

6.4.1 Code Reference

For each T_{in}, The private spend key is retrieved in the loop cryptonote_tx_utils.cpp line 225 by calling generate_key_image_helper (cryptonote_tx_utils.cpp line 239). The following commands allow to implement generate_key_image_helper in a secure way.

In order not to publish the T_{in} secret spend key x_{in} to the host, the key is returned encrypted by *spk*.

The commands take into account sub-address-v2 by first retrieving the public derivation data, checking if it belongs or not to a sub-address then retrieving the secret key and key image according to that.

6.4.2 Description

For each input T_{in}, the application receives the R_{in} transaction public key.

Once received the application SHALL verify that the public key is valid, i.e the Point is on curve and its order is correct.

After checking the input transaction public key, the application computes the public derivation data $\mathfrak{D}_{\text{in}}$ and returns it.

$\mathfrak{D}_{\text{in}} = \texttt{DeriveDH}(a, R_{in})$

Just after this command, the application shall receive the request to compute the signature key. In other word retrieve for the input transaction the triplet x_{in}, P_{in}, I_{in}. This command takes one argument: the sub-key index. Zero means main spend key, non zero value means sub_key. Thus, the application processes the command this way:

compute $x_{in} = \texttt{DerivePriv}(\mathfrak{D}_{\text{in}}, b)$
if *idx != 0* :
  $x_{in} = x_{in} + \texttt{H}_\texttt{s}(\text{“subAddr”} \mid a \mid idx)$
compute $P_{in} = x_{in}.G$
compute $I_{in} = \texttt{DeriveImg}(x_{in}, P_{in})$
compute $\widetilde{x_{in}} = \texttt{AES}[spk](x_{in})$

Note that the application returns $\widetilde{x_{in}}$, i.e. x_{in} protected by *spk*.

6.4.3 Commands

6.4.3.1 Get Derivation Data

Command

CLA	INS	P1	P2	LC	data description
00	54	01	cnt	21	

Command data

Length	Value
01	option
20	Public input transaction key R_{in}

Response data

Length	Value
20	public input derivation data $\mathfrak{D}_{\text{in}}$

6.4.3.2 Get Input Keys

Command

CLA	INS	P1	P2	LC	data description
00	54	02	cnt	05	

Command data

Length	Value
01	option
04	Sub-key index, 0 means main key
04	real output index

Response data

Length	Value
20	encrypted private input spend key $\widetilde{x_{in}}$
20	public input spend key P_{in}
20	P_{in} key image I_{in}

6.5 Process Output Transaction Keys

6.5.1 Code Reference

For each output transaction, the destination key is computed by calling generate_key_derivation in cryptonote_tx_utils.cpp line 287 and and derive_public_key in cryptonote_tx_utils.cpp line 287

In case of sub-address-v2 a dedicated interaction is done to retrieve the change address. Note here, the derivation data must be kept secret as it is used to blind the amount. So it is returned encrypted to the host host and must be stored in tx as temporary data (associated to the destination key) for the subsequent steps.

6.5.2 Description

Compute either the destination key or the change key.

If destination key is requested, perform the following:

compute $\mathfrak{D}_{\text{out}} = \texttt{DeriveDH}(r, A_{out})$
compute $P_{out} = \texttt{DerivePub}(\mathfrak{D}_{\text{out}}, B_{out})$

If change key is requested, perform the following:

compute $\mathfrak{D}_{\text{out}} = \texttt{DeriveDH}(a, R)$
compute $P_{out} = \texttt{DerivePub}(\mathfrak{D}_{\text{out}}, B)$

Finally:

compute $\widetilde{\mathfrak{D}_{\text{out}}} = \texttt{AES}[spk](\mathfrak{D}_{\text{out}})$
update $\mathcal{L} : \texttt{H}_{\texttt{update}}(A_{out} \mid \mathfrak{D}_{\text{out}} \mid B_{out} \mid P_{out})$

In both cases, return P_{out} and $\widetilde{\mathfrak{D}_{\text{out}}}$.

6.5.2.1 Get Output Keys

Command

CLA	INS	P1	P2	LC	data description
00	56	01	cnt	41	

Command data

Length	Value
01	options
20	Destination view Key A_{out}

Length	Value
20	Destination spend Key B_{out}

option encoding

`-------x-`	Change key request
`-------0-`	Generate destination key
`-------1-`	Generate change key

Response data

Length	Value
20	public destination key P_{out}
20 |	encrypted private derivation data $\widetilde{\mathfrak{D}_{\text{out}}}$

6.6 Perform range proof and blinding

6.6.1 Code Reference

Once T_{in} and T_{out} keys are set up, the genRCT function is called (cryptonote_tx_utils.cpp line 450).

First a commitment C_v to each v amount, and 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 $C_v = k.G + v.H$ (rctSigs.cpp line L589).

Then k and v are blinded by using the $\mathfrak{D}_{\text{out}}$ which is only known in an encrypted form by the host (rctSigs.cpp L597_).

6.6.2 Description

This command receives both the mask value and the amount to blind, plus the encrypted private derivation data computed during the processing of output transaction keys (GOK_).

The application performs the following steps:

compute $\mathfrak{D}_{\text{out}} = \texttt{AES}^{-1}[spk](\widetilde{\mathfrak{D}_{\text{out}}})$
compute $\widetilde{k} = k + \texttt{H}_{\texttt{p}\rightarrow\texttt{s}}(\mathfrak{D}_{\text{out}})$
compute $\widetilde{v} = k + \texttt{H}_{\texttt{s}}(\texttt{H}_{\texttt{p}\rightarrow\texttt{s}}(\mathfrak{D}_{\text{out}}))$
update $\mathcal{L}$: $\texttt{H}_{\texttt{update}}(v \mid k \mid \mathfrak{D}_{\text{out}})$

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

6.6.3 Commands

6.6.3.1 Blind Amount and Mask

Command

CLA	INS	P1	P2	LC	data description
00	58	01	cnt	var	

Command data

Length	Value
01	options
20	value v
20	mask k
20 |	encrypted private derivation data $\widetilde{\mathfrak{D}_{\text{out}}}$

Response data

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

6.7 MLSAG

6.7.1 Code Reference

Interaction overview

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

ProveRctMG : rctSigs.cpp line 361
Call to MLSAG_Gen : rctSigs.cpp line 362
MLSAG_Gen : rctSigs.cpp line 116

At this point the amounts and destination keys must be validated on NanoS. This information is embedded in the message to sign by calling get_pre_mlsag_hash at rctSigs.cpp line 613, 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 pre-hash. The prehash will be kept inside NanoS to ensure its integrity. Any further access to the prehash will be delegated.

Once 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 rctSigs.cpp line 191

Amount and destination validation

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.

NanoS interaction

NanoS operates when manipulating the encrypted input secret key x_{in}, the prehash, the α_{in} secret key and the final $\mathcal{H}$. So the last function to modify is the MLSAG_Gen. The message (prehash $\mathcal{H}$) is held by the NanoS. So the vector initialization must be skipped and the two calls to hash_to_scalar(toHash) must be modified

- init: rctSigs.cpp line 139
- call 1: rctSigs.cpp line 158
- call 2: rctSigs.cpp line 182

The α_{in} , $\alpha_{in}.G$ generation is delegated to NanoS:

- call 1: rctSigs.cpp line 142
- call 2: rctSigs.cpp line 153

As consequence point computation line 144 (rctSigs.cpp line 144) is also delegated. Finally the key Image computation must be delegated to the NanoS: rctSigs.cpp line 148

6.7.2 Description

Part 1: prehash

Validate the destinations and amounts and compute the MLSAG prehash value.

This final part is divided in three steps.

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

finalize $\mathcal{L}$: $\mathtt{H}_{\mathtt{finalize}}()$
update $\mathcal{H}$: $\mathtt{H}_{\mathtt{update}}(header)$

On the second step (SAP_) 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 commend received, do:

compute $\mathfrak{D}_{\text{out}} = \mathtt{DeriveDH}(r, A_{out})$‘
compute $k = \widetilde{k}$ - $\mathtt{H}_{\mathtt{p}\rightarrow\mathtt{s}}(\mathfrak{D}_{\text{out}})$
compute $v = \widetilde{k}$ - $\mathtt{H}_{\mathtt{s}}(\mathtt{H}_{\mathtt{p}\rightarrow\mathtt{s}}(\mathfrak{D}_{\text{out}}))$
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 $\mathcal{L}$: $\mathtt{H}_{\mathtt{update}}(A_{out} \mid B_{out} \mid \mathfrak{D}_{\text{out}} \mid v \mid k \mid \mathfrak{D}_{\text{out}})$

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

Finally the application receives the last part of data (SEN_):

finalize $\mathcal{L}$’ : $\mathtt{H}_{\mathtt{finalize}}()$
check $\mathcal{L} == \mathcal{L}$’

finalize $\mathcal{C}$: $\mathtt{H}_{\mathtt{finalize}}()$
compute $\mathcal{C}$’ = $\mathtt{H}_{\mathtt{finalize}}(commitment_0.Ct|commitment_1.Ct|.....)$ |
check $\mathcal{C} == \mathcal{C}$’

finalize $\mathcal{H}$: $\mathtt{H}_{\mathtt{finalize}}(commitments)$
compute $\mathcal{H} = h(message \mid \mathcal{H} \mid proof)$

Keep $\mathcal{H}$

Part 2: signature

Step 1:

Generate the matrix ring paramaters.

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}]

Step 2:

Compute the last matrice ring parameter

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

Step 3:

Finaly compute all signature:

> 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

6.7.3 Commands

6.7.3.1 Initialize MLSAG-prehash

Command

CLA	INS	P1	P2	LC	data description
00	5A	01	00	var	

Command data

Length	Value
01	options

var	serialized header :
	{
	type
	txnFee
	pseudoOut
	}

6.7.3.2 Update MLSAG-prehash

Command

CLA	INS	P1	P2	LC	data description
00	5A	02	cnt	var	

Command data

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

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

6.7.3.3 Finalize MLSAG-prehash

Command

CLA	INS	P1	P2	LC	data description
00	5A	03	00	var	

Command data

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

var	serialized commitments :
	{
	bytes[32] mask (C_v)
	}

Response data

Length	Value

6.7.3.4 MLSAG prepare

Command

CLA	INS	P1	P2	LC	data description
00	5C	01	cnt	var	

Command data

for real key:

Length	Value
01	options
20	point
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	$\alpha_{in}.G$

6.7.3.5 MLSAG start

Command

CLA	INS	P1	P2	LC	data description

CLA	INS	P1	P2	LC	data description
00	5C	02	00	var	

Command data

Length	Value
01	options
var	inputs

Response data

Length	Value

6.7.3.6 MLSAG sign

Command

CLA	INS	P1	P2	LC	data description
00	5C	03	cnt	var	

Command data

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

Response data

Length	Value
20	signature *ss*

7 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.

8 Annexes

8.1 Helper functions

DeriveDH

Input : r, P
Ouput: $\mathfrak{D}$
Monero: generate_key_derivation

$\mathfrak{D} = r.P$
$\mathfrak{D} = 8.\mathfrak{D}$

DerivePub

input: $\mathfrak{D}, B$
output: P
Monero: derive_public_key

$P = \mathtt{H}_{\mathtt{p}\to\mathtt{s}}(\mathfrak{D}).G + B$

DerivePriv

input: D,b
output: x
Monero: derive_private_key

$x = \mathtt{H}_{\mathtt{p}\to\mathtt{s}}(\mathfrak{D}) + b$

DeriveImg

input: x, P
output: I
Monero:

$I = x_{in}.\mathtt{H}_{\mathtt{p}}(P_{in})$

$\mathtt{H}_{\mathtt{p}\to\mathtt{s}}$

input: D, idx
output: s

$data = point2bytes(D)|varint(idx)$
$s = h(data)$

DeriveAES

This is just a quick proposal. Any other KDF based on said standard may take place here.

input: R, a, b
output: *spk*

$seed = sha256(R|a|b|R)$
$data = sha256(seed)$
$spk = lower16(data)$

8.2 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