NanoS integration into Monero

(Ledger SAS, cedric@ledger.fr, Draft Note v0.6)

Table of Contents

I. Short 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 anticipate next client feature and propose solution to integrate the PR2056 for subaddress. This proposal is based on kenshi84 fork, branch subaddress-v2.

II. Notations and definitions

Upper case letter	point
Lower case letter	scalar
(a,A), (b,B)	signer main view/spend key pair
(c,C), (d,D)	signer sub view/spend key pair
A_{out}, B_{out}	receiver main view/spend key
C_{out}, D_{out}	receiver sub view/spend key
v	amount to send/spent
k	secret amount mask factor
C	commitment to *a* with *x* such C = kG+vH
H	2nd group generator, such G=h.H and h is unknown
$DeriveDH$	Derivation function: scalar,point $\mapsto$ point
$\mathrm{DerivePub}$	Derivation function: scalar,point $\mapsto$ point
$\mathrm{DerivePriv}$	Derivation function: point,scalar $\mapsto$ scalar
$H_{\mathrm{p}}, H_{\mathrm{s}}, H_{\mathrm{p}\to\mathrm{s}}$	hash function (H_{p}: point to point, H_{s}: scalar to scalar, $H_{\mathrm{p}\to\mathrm{s}}$: point to scalar)
sha256	sha256 hash function
low16B	lower 16 bytes function
high16B	higher 16 bytes function
m	message to sign
$\overline{\overline{\mathrm{d}}}$	encrypted data d, non decryptable by Host.
DeriveAES	Derivation function: ... $\mapsto$ AES 128 key
AES[k](d)	AES based encryption with integrity. (d data to encrypt and protect with k key)
AES^{-1}[k]($\overline{\overline{\mathrm{d}}}$)	AES based decryption with integrity check. ($\overline{\overline{\mathrm{d}}}$ data to decrypt and to verify with k key)
EPIT	8 bits constant ENCRYPTED_PAYMENT_ID_TAIL

III. Reminders

Here we shortly describe the process to build a Monero transaction in official client v0.10.3.1

The transaction is build in the construct_tx_and_get_tx_key function (https://github.com/monero-project/monero/blob/v0.10.3.1/src/cryptonote_core/cryptonote_tx_utils.cpp#L159) . This will be our start point.

Steps:

1. Generate a TX key pair (r,R)
2. Process Stealth Payment ID
3. For each input T_{in} to spent:
   retrieve the spend key (x_i,P_i) from R_{in} and b
   Compute the key image *I* of x_i
4. For each output T_{out} :
   compute the output public key from A_{out} and r
5. For each output T_{out} :
   compute the range proof and blind the amount
6. compute the confidential ring signature with involved x_{in}
7. Return TX

IV. Goals

Goals summary:

Secret Protection:

- secret key account (a,b)
- secret key transaction r
- per transaction spend key xi

Integrity protection:

- amount
- destination

Support:

- Old ring transaction
- Ring Confidential Transaction (MoneroRCT)
- Sub Address v2

Integration:

- Official Monero client (web clients later)

V. Nanos Integration

V.1. Step 1: TX key

The transaction key is generated here https://github.com/monero-project/monero/blob/v0.10.3.1/src/cryptonote_core/cryptonote_tx_utils.cpp#L169

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 secret transaction key may be return encrypted by SPK for later purpose. Indeed the secret transaction key is lost by the token at the end of transaction and can not be retrieved if not

saved by host. If secret transaction key need to be saved, the SPK is generated in a deterministic way.

Finally, an optional exchange is done to override he TX public key in case of Sub-Address.

Host		Nanos
Data:		Data: (a,A), (b,B)
Request TX key		
	→ keep	

		Generate TX key pair r,R
		compute $\bar{\bar{r}}$ = empty
		if keep:
		compute k_r = *DeriveAES*(R,a,b)
		compute $\bar{\bar{r}}$ = *AES*[k_r](r)
		else:
		compute k_r = random(16)

	← R, $\bar{\bar{r}}$	
Request Sub TX key (optional)	D_{out} →	
		Re-Compute R = r.D_{out} if keep: compute k_r = *DeriveAES*(R,a,b) compute $\bar{\bar{r}}$ = *AES*[k_r](r)
	← R, $\bar{\bar{r}}$	
Data: R		Data: (a,A), (b,B) r, k_r

V.2. Step 2: Process Stealth Payment ID

Host		Nanos
Data: R, P_{in}, $\tilde{x}_{in}$, I_{in}		Data: (a,A), (b,B) r, k_r
Request encrypted paymendID		
	→ payID, A_{out},	
		compute $\mathcal{D}$ =*DeriveDH*(r,A_{out}) compute key s = H_s(*point2bytes*($\mathcal{D}$)||EPIT)) compute $\widetilde{payID}$ = payID ⊕ s
	← $\widetilde{payID}$	$\widetilde{payID}$
Data: R, P_{in}, $\tilde{x}_{in}$, I_{in}, P_{out}, $\widetilde{D}_{out}$		Data: (a,A), (b,B) r, k_r

V.3. Step 3: spend key

Spend keys for each T_{in} are retrieved in the loop line #225 by calling generate_key_image_helper (https://github.com/monero-project/monero/blob/v0.10.3.1/src/cryptonote_core/cryptonote_tx_utils.cpp#L239) . The following sequence is the equivalent of the one in generate_key_image_helper (https://github.com/monero-project/monero/blob/v0.10.3.1/src/cryptonote_basic/cryptonote_format_utils.cpp#L132) . In order not to publish the T_{in} spend key x_{in} to the host, the key is returned encrypted by a session key.

This sequence take into account subaddress-v2 by first retrieving the public derivation data, checking if it belong or not to a subaddress and retrieving the secret key and key image according to that.

Host		Nanos
Data: R T_{in}		Data: (a,A), (b,B) r, k_r
Request $\mathcal{D}_{in}$ derivation data		
	→ R_{in}	
		Check the R_{in} order compute $\mathcal{D}_{in}$ = *DeriveDH*(a,R_{in})
	← $\mathcal{D}_{in}$	
Request key	→ idx	
		compute private key x_{in} = DerivePriv($\mathcal{D}$, b) if idx != 0: x_{in} = x_{in} + H_s("subAddr" | a | idx) compute public key P_{in}= x_{in} . P_{in} compute key image I_{in} = DeriveImg(x_{in},P_{in}) compute $\overline{x}_{in}$ = *AES*[k_r](x_{in})
	← P_{in}, $\overline{\overline{x}}_{in}$, I_{in}	
Data: R, P_{in}, $\overline{\overline{x}}_{in}$, I_{in}		Data: (a,A), (b,B) r, k_r

V.4. Step 4: destination key

The computation destination key is performed by calling crypto::generate_key_derivation (https://github.com/monero-project/monero/blob/v0.10.3.1/src/cryptonote_core/cryptonote_tx_utils.cpp#L278) and crypto::derive_public_key (https://github.com/monero-project/monero/blob/v0.10.3.1/src/cryptonote_core/cryptonote_tx_utils.cpp#L287). The first one provide an intermediate value D used to blind mask and amount of the confidential commitment C.

In case of subaddress-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. The encrypted derivation data is returned to the host and must be stored in tx as temporary data (associated to the destination key) for the subsequent steps.

Host		Nanos
Data: R, P_{in}, $\tilde{\tilde{x}}_{in}$, I_{in}		Data: (a,A), (b,B) r, R, k_r
Request destination key A_{out}, B_{out} can be either main address or sub address		
	→ A_{out}, B_{out}	
		compute $\mathcal{D}$ = *DeriveDH*(r, A_{out}) compute P_{out} = *DerivePub*($\mathcal{D}$, B_{out}) compute $\tilde{\tilde{\mathcal{D}}}_{out}$ = AES[k_r]($\mathcal{D}$)
	← P_{out}, $\tilde{\tilde{\mathcal{D}}}_{out}$	
Request change key		
	→	
		compute $\mathcal{D}$ =*DeriveDH*(a, R) compute P_{out} = *DerivePub*($\mathcal{D}$, B) compute $\tilde{\tilde{\mathcal{D}}}_{out}$= AES[$k_r$]($\mathcal{D}$)
	← P_{out}, $\tilde{\tilde{\mathcal{D}}}_{out}$	
Data: R, P_{in}, $\tilde{\tilde{x}}_{in}$, I_{in}, P_{out}, $\tilde{\tilde{D}}_{out}$		Data: (a,A), (b,B) r, R, k_r

V.5. Step 5: range proof and blinding

Once T_{in} and T_{out} are set up, the genRCT function is called (https://github.com/monero-project/monero/blob/v0.10.3.1/src/cryptonote_core/cryptonote_tx_utils.cpp#L450). First a commitment *C* to each $v_{out,}$ and associated range proof are computed to ensure the *v* amount confidentiality. The commitment and its range proof does not imply any secret and generate *C*,*k* such *C* = *k*.G+*v*.H, where *v* is the real amount. (https://github.com/monero-project/monero/blob/v0.10.3.1/src/ringct/rctSigs.cpp#L589)
Second, *k* and *v* are blinded by using the D_{out} which is only known in an encrypted form by the host. (https://github.com/monero-project/monero/blob/v0.10.3.1/src/ringct/rctSigs.cpp#L597)
This blinding can be delegated as follow.

Host		Nanos
Data: R, P_{in}, $\widetilde{x}_{in}$, I_{in}, $\widetilde{D}_{out,}$, k_{out}, v_{out}		Data: (a,A), (b,B) r, R, k_r
Request blinded mask and amount		
	→ k,v, $\widetilde{\mathcal{D}}_{out}$	
		compute $\mathcal{D} = AES^{-1}[k_r](\widetilde{\mathcal{D}}_{out})$ compute $\overline{k} = k + H_{p \to s}(\mathcal{D})$ compute $\overline{v} = k + H_s(H_{p \to s}(\mathcal{D}))$
	← $\overline{k}$, $\overline{v}$	
Data: R, P_{in}, $\widetilde{x}_{in}$, I_{in},P_{out}, $\widetilde{D}_{out}$, k_{out}, v_{out}, $\widetilde{k}_{out}$, $\widetilde{v}_{out}$		Data: (a,A), (b,B) r, R, k_r

V.6. Step 6: RCT

The little bit tricky part!

V.6.1. Raw explanation

V.6.1.1. Interaction Overview

After all commitments have been setup, the ring signature operates. The ring confidential signature is performed by calling proveRctMG which call MLSAG_Gen

ProveRctMG : https://github.com/monero-project/monero/blob/v0.10.3.1/src/ringct/rctSigs.cpp#L613,
Call to MLSAG_Gen : https://github.com/monero-project/monero/blob/v0.10.3.1/src/ringct/rctSigs.cpp#L362
MLSAG_Gen : https://github.com/monero-project/monero/blob/v0.10.3.1/src/ringct/rctSigs.cpp#L116

At this point we need to validate amount and destination key on NanoS. Those information are embedded in the message to sign by calling get_pre_mlsag_hash line rctSigs.cpp#L613, 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_i, aG_i. All α_i must be kept secret to protect the x_{in} keys. Moreover we must avoid signing arbitrary values during the final loop https://github.com/monero-project/monero/blob/v0.10.3.1/src/ringct/rctSigs.cpp#L191

V.6.1.2. Amount and destination validation

During *rv* serialisation, NanoS receives a list of tuple <P_{out}, $\overline{k}$, $\overline{v}$>. In order to do that we need to approve the orignal destination address A_{out}, which is not recoverable from P_{out}. Here the only

solution is to pass the orignal destination with the *rv.* (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), and recompute P_{out} and check it matches the one in *rv*. If user validate A_{out}, and P_{out} matches, then P_{out} is validated
Finally, as we now hold D_{out}, we can unblind $\tilde{\tilde{k}}$ and $\tilde{\tilde{v}}$ and validate that C = kG+vH, and ask the user to validate the amount *v.* If both are ok, this TX_{out} is validated.

V.6.2. NanoS interaction

NanoS operates when manipulating the encrypted input secret key at step 2, the prehash, the α_i secret key and the final c value (see step 5.1). So the last function to modify is the MLSAG_Gen.

The message (prehash mslsag) 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: https://github.com/monero-project/monero/blob/v0.10.3.1/src/ringct/rctSigs.cpp#L139
call 1: https://github.com/monero-project/monero/blob/v0.10.3.1/src/ringct/rctSigs.cpp#L158
call 2: https://github.com/monero-project/monero/blob/v0.10.3.1/src/ringct/rctSigs.cpp#L182

The α_i, aG_i generation is delegated to NanoS:
call 1: https://github.com/monero-project/monero/blob/v0.10.3.1/src/ringct/rctSigs.cpp#L142
call 2: https://github.com/monero-project/monero/blob/v0.10.3.1/src/ringct/rctSigs.cpp#L153
As consequence point computation line 144 is also delegated

Finally the key Image computation must be delegated to the NanoS:
https://github.com/monero-project/monero/blob/v0.10.3.1/src/ringct/rctSigs.cpp#L148

V.6.2.1. MLSAG- Prehash

In the following delegation, the NanoS only needs to keep mask and amount of rv.ecdhInfo structure which takes 64 bytes per destination.

Host		Nanos
Data: R, P_{in}, $\widetilde{x}_{in}$, I_{in}, $\widetilde{D}_{out}$, rv, k_{out}, v_{out}, $\widetilde{k}_{out}$, $\widetilde{v}_{out}$		Data: (a,A), (b,B) r, R, k_r
start rv serialisation		
	→ rv.type, rv.txnFee, rv.pseudoOut, rv.ecdhInfo	
		Init *H* update H with inputs Keep ecdhInfo
	←	
for each ctkey in rv.outPk send ctkey, real destination address and request validation		
	→ $ctkey_i$, (A_i,B_i), $\widetilde{\mathcal{D}}_i$ (3)	
		Compute $\mathcal{D}$ =$AES^{-1}[k_r](\widetilde{\mathcal{D}}_i)$ compute P_{out}= *DerivePub*($\mathcal{D}$, B_{out}) compute(1) k = ecdhInfo.mask - $H_{p\to s}(\mathcal{D})$ compute(1) v = ecdhInfo.amount - $H_s(H_{p\to s}(\mathcal{D}))$ check(2) $ctkey_i$.mask == k.G+v.H check P_{out} == $ctkey_i$.dest Request user to validate A_i , B_i, v If checks passed and user has validated : update H with $ctkey_i$ else reject the transaction
	←	
end rv serialisation		
	→ rv.rangeSigs, rv.MGs	
		finalize H with inputs => mlsag_prehash
	← ZERO_HASH	
Data: R, P_{in}, $\widetilde{x}_{in}$, I_{in},P_{out}, $\widetilde{D}_{out}$, k_{out}, v_{out}, $\widetilde{k}_{out}$, $\widetilde{v}_{out}$		Data: (a,A), (b,B) r, R, k_r, mlsag_prehash

Note 1: ecdhInfo.mask is $\widetilde{k}$, ecdhInfo.amount is $\widetilde{v}$
Note 2: $ctkey_i$.mask is commitment C
Note 3: $\widetilde{\mathcal{D}}_i$ is $\widetilde{\mathcal{D}}_{out}$ computed at step 4

V.6.2.2. MLSAG- signature

The last step is the signature of the matrix and prehash.

Remenber that all private input keys are encrypted by the NanoS, so xx[i] contains $\widetilde{x}_i$ and alpha[i] will contain $\widetilde{\alpha}_i$

Host		Nanos
Data: R, P_{in}, $\tilde{\tilde{x}}_{in}$, I_{in}, $\tilde{\tilde{D}}_{out}$, rv, k_{out}, v_{out}, $\tilde{\tilde{k}}_{out}$, $\tilde{v}_{out}$		Data: (a,A), (b,B) r, k_r, mlsag_prehash
Request αH_i, αG_i, II_i		
	→ H_i, $\bar{x}_i$	
		check the order of H_i generate α_i, αG_i compute $x_{in} = AES^{-1}[k_r](\bar{x}_i)$ compute $II_i = x_i.Hi$ compute $\alpha H_i = \alpha_i.Hi$ compute $\bar{\alpha}_i = AES[k_r](\alpha_i)$
	← $\bar{\alpha}_i$, αH_i, αG_i, II_i	
Request αG_i		
	→	
		generate α_i, αG_i
	← $\bar{\alpha}_i$, αG_i	
Data: R, P_{in}, $\tilde{\tilde{x}}_{in}$, I_{in},P_{out}, $\tilde{\tilde{D}}_{out}$, k_{out}, v_{out}, $\tilde{\tilde{k}}_{out}$, $\tilde{v}_{out}$, $\bar{\alpha}_i$		Data: (a,A), (b,B) r, R, k_r, mlsag_prehash

Then the hash_to_scalar must be fully delegated

Host		Nanos
Data: R, P_{in}, $\tilde{\tilde{x}}_{in}$, I_{in}, $\tilde{\tilde{D}}_{out}$, rv, k_{out}, v_{out}, $\tilde{\tilde{k}}_{out}$, $\tilde{v}_{out}$, $\bar{\alpha}_i$		Data: (a,A), (b,B) r, R, k_r, mlsag_prehash
Serialize toHash		
	→ $tohash_{bytes}[]$	
		Set $tohash_{bytes}[0:32]$ = mlsag_prehash compute c = $H(tohash_{bytes}[])$ keep c
	← c	
Data: R, P_{in}, $\tilde{\tilde{x}}_{in}$, I_{in},P_{out}, $\tilde{\tilde{D}}_{out}$, k_{out}, v_{out}, $\tilde{\tilde{k}}_{out}$, $\tilde{v}_{out}$, $\bar{\alpha}_i$		Data: (a,A), (b,B) r, R, k_r, mlsag_prehash, c

Finally the last mixup https://github.com/monero-project/monero/blob/v0.10.3.1/src/ringct/rctSigs.cpp#L191 is also delegated. Here it is important to use the last c value generated by the NanoS. Indeed the c value is a hash of data which contains the prehash as its first 32bytes. This enforces that the final c signed value cannot be forced by the Host and matches the previously user validated amount and destination.

Host		Nanos
Data: R, P_{in}, $\tilde{x}_{in}$, I_{in}, $\tilde{D}_{out}$, rv, k_{out}, v_{out}, $\tilde{k}_{out}$, $\tilde{v}_{out}$, $\bar{\alpha}_i$		Data: (a,A), (b,B) r, k_r, mlsag_prehash
Request ss[i]		
	$\rightarrow \bar{x}_i$, $\bar{\alpha}_i$	
		compute $\alpha_j = AES^{-1}[k_r](\bar{\alpha}_j)$ compute $x_j = AES^{-1}[k_r](\bar{x}_j)$ compute ss = $(\alpha_i - c * \bar{x}_j) \% l$
	$\leftarrow$ ss	
Data: R, P_{in}, $\tilde{x}_{in}$, I_{in},P_{out}, $\tilde{D}_{out}$, k_{out}, v_{out}, $\tilde{k}_{out}$, $\tilde{v}_{out}$, $\bar{\alpha}_i$		Data: (a,A), (b,B) r, R, k_r, mlsag_prehash

VI. Conclusion

This draft note explains how to protect Monero transactions of the official client with a NanoS. According to the last 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.

VII. Annexes

VII.1. Helper functions

DeriveDH
*Input :*r , P
Ouput: D
Monero: generate_key_derivation

$$D = r.P$$
$$D = 8.D$$

DerivePub
input: D,B
output: P
Monero: derive_public_key

$$P = H_{p \rightarrow s}(D).G+B$$

DerivePriv
input: D,b

output: x
Monero: derive_private_key

$x = H_{p \to s}(D)+b$

DeriveImg
input: x,P
output: I
Monero: derive_private_key

$I = x_{in} . H_p(P_{in})$

D2s
input: D, idx
output: s
Monero: derive_private_key

data = *point2bytes*(D) || *scalar2bytes*(idx)
s = H_s(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

hkey = sha256(“AES”|R|seed)
data = hmac_sha256[hkey](sha512(R))
spk = lower16(data)

seed shall be retrievable from a and b or master seed. Example:
seed = sha256(sha256(a)|sha256(b))

VII.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/