Note: Decentralised Recovery (DeRec)
Date: 21 August 2024
Author: Rupp

Introduction

Decentralised recovery is a novel technique to recover secrets from a group of agents who keep shares of that secret. The overall mechanism is very simple: a secret - or a set of secrets - S is cryptographically split into N shares : $S_1, \ldots, S_N$ by a user $\mathcal{U}$. Each secret $S_i, 1 \leq i \leq N$ is then given to an *agent* $\mathcal{A}_i, 1 \leq i \leq N'$ which is called a *helper. The choice of the helpers is defined by* $\mathcal{U}$.
A specific function
$\Phi : \{S_i, 1 \leq i \leq N\} \rightarrow \{\mathcal{A}_i, 1 \leq i \leq N'\}$ maps secret shares to helpers. Once a user decides to share the secret among agents, we call it a *sharer*.

The secret sharing technique is based on the Shamir Secret Splitting Scheme (SSSS) which is described in [1] and [2]. The secret that will be share in such case will be the key K used to cipher all the passwords and other credentials that $\mathcal{U}$ wishes to backup.

The sharer instals an application on a compatible device, the chosen helpers do the same and when the configuration is ready, the sharer meets each helper *in person* and they pair their respective devices. By pairing, they open a secure channel which is resistant to tampering. Once the secure channel is open, the share as well as additional data are sent to the device of the helper. The process is completed when the sharer has paired and shared secrets with a quorum of helpers known as the threshold t and defined in SSSS. Of course the sharer can share with more helpers, e.g they can share with $N' > t$ helpers. This is even recommended in order to have more choice during recovery. In the next sections, we will detail each of the cryptographic protocols and algorithms involved in decentralised recovery, we will analyse the security model and we will propose options where the security could be improved by using a javacard applet and a smartcard such as the smartcard wallet.

The Shamir Secret Sharing Scheme (SSSS)

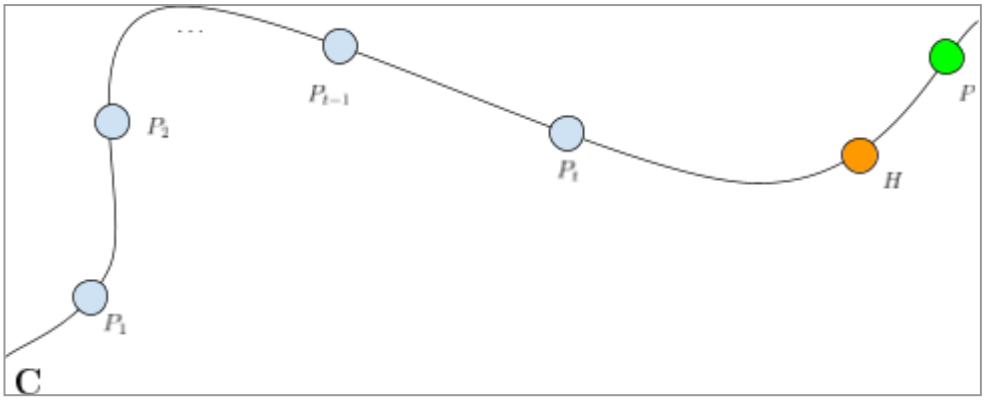
SSSS has never been really standardised, There is no RFC, ISO or similar norms that describes it. The only attempt to create a standard from it can be found in [2].

The most simple way to share a secret would be to directly split it into parts as a stream of bytes. There are obvious caveats in this approach and even with an incomplete amount of share an attacker could “guess” the remaining parts using several techniques, therefore it’s not feasible.

The approach proposed by Shamir is to hide the secret as a point on a secret polynomial curve C over a finite field F_q. The shares are other points on this curve. The only way to obtain the secret is to reconstruct the secret curve and the only way to reconstruct the curve is to have the quorum of points (shares) to do it.

The sharing process is as follows:

- *Define a polynomial curve C over a finite field F_q represented by a polynomial function $f(x) = \sum_{1 \leq i \leq t-1} a_i x^i, a_i \in F_q, x \in F_q$;*
- *Define a point P on this curve representing the secret;*
- *Determine a quorum of points $P_i, 1 \leq i \leq t$ over the curve allowing the reconstruction of the curve;*
- *Discard the secret curve C and distribute each point P_i, to a specific helper.*
- *Additionally a hash of the secret can be stored in point H and distributed to the helpers to check the validity of the reconstruction.*



The x-abscisse of the points must be defined. Usually P is at abscisse $x = 0$ and the shares are at abscisse $x(P_i) = i$ (using an enumeration of the field with indexes $1, 2, \ldots, i, \ldots, q$) , the field F_q is the field with $q = 256$ elements. However since SSSS isn't standardised, different conventions may be used depending on the implementations. The amount of shares is obviously limited by q. For $q = 256$ there can be at most 255 shares (or 254 if the hash is used) . The degree $t - 1$ of the polynomial function f determines the quorum t of shares.

The recovery simply uses the well-known Lagrange interpolation formula from the shares.

$$f(x) = \sum_{j=1}^{j=t} y_j \prod_{k=1, k \neq j}^{k=t} \frac{x - x_k}{x_j - x_k}$$

Where the shares are : $P(i) = (x_i, y_i), 1 \leq i \leq t$

Once the secret polynomial function f has been recovered, the secret is also easily recovered.

To check that the sharing is secure, let's consider the worst-case scenario when all the shares needed to reconstruct the polynomial but one are known by an attacker. Since the value of a secret is an element of F_q it

can take only but q values. If $t-1$ shares are known there still remain q polynomials which are all valid candidates for reconstruction, this means that this doesn't bring more information than random guessing of the secret. If no shares are known the amount of possible polynomials is equal to q^t.

In computer implementations, q is taken to be 256 which allows to map an element of F_q to a byte. In order to split a stream of bytes, it's enough to repeat the sharing process for each byte. To fix the ideas, if we want to split-share the byte stream $b^1, b^2, ..., b^k, ...$ we define first a quorum t and then we randomly share each byte b^k seen as an element of $F_q = F_{256}$, we obtain t shares $P_1^k, P_2^k, \ldots, P_t^k$ represented by their ordinates (coded as one byte):
$y(P_1^k), y(P_2^k), \ldots, y(P_t^k)$
(only ordinate since their abscisses are fixed by convention). At the end of the process, the stream of bytes is shared by the t streams of bytes:

$y(P_1^1), y(P_1^2), \ldots, y(P_1^k), \ldots$
....
$y(P_t^1), y(P_t^2), \ldots, y(P_t^k), \ldots$

Pairing Protocol for Decentralised Recovery

During the transmission of a secret share to a helper, it is needed to perform a mutual authentication between the helper and the sharer. The base authentication is done *only* in-person, e.g via a visual contact. Once this first authentication is done, the sharer and the helper must pair each other's devices (e.g a device where the DeRec application has been installed) . The description of the pairing protocol in [3] is clearly incomplete so, in order to fully describe it, we have to reverse-engineer it from the source code.

The source code can be found in the repository https://github.com/derecalliance/cryptography/blob/main/src/secure_channel . It is composed of three files written in RUST :

- *sign.rs*
- *encrypt.rs*
- *mod.rs*

The principle is extremely simple:

- Two SECP256K1 key pairs are generated: K_{sign} and K_{enc} used respectively for signature and encryption.
- Assuming two participants, helper and sharer, in that context. They generate both their signatures and encryption keys $K_{sign}^{helper}, K_{enc}^{helper}$ for the helper and $K_{sign}^{sharer}, K_{enc}^{sharer}$ for the sharer.
- One of the participants - let us assume it's the sharer - initiates the secure channel by providing their public keys for signature and encryption to the helper. $(K^{PUB})_{sign}^{sharer}, (K^{PUB})_{enc}^{sharer}$ and assuming the helper

does the symmetrical operation, e.g provides their public keys for signature and encryption to the sharer. $(K^{PUB})^{helper}_{sign}, (K^{PUB})^{helper}_{enc}$ (this is not part of the API but this has to be done in some ways for the API to function)

- The sharer signs a message m with his private key for signature $(K^{PRIV})^{sharer}_{sign}$ with the ECDSA signature algorithm, and the result $sign(m)$ is concatenated with the message to form a new message $m' = m||sign(m)$.
- This new message is then encrypted by the helper's public key for encryption $(K^{PUB})^{helper}_{enc}$ with the ECIES encryption algorithm (Integrated Encryption for Elliptic Curves) . the final message $m"$ is transmitted to the helper.
- Once the message $m"$ is received , the helper decrypts it using his private key for encryption $(K^{PRIV})^{helper}_{sign}$, then presumably knowing the size and position of the signature, they extract the signature from the deciphered message m' and verify it via the public key of the sharer for signature $(K^{PUB})^{helper}_{sign}$.
- If the signature is verified, the message m is extracted and processed by the Helper.

There are several caveats in this API :

1) The key generation is done via a random generator `generate_encryption_key()` located in `encrypt.rs`. This function generates both encryption keys and signature keys used for the secure channel. The random generator is `rand_chacha::rand_core::OsRng`. It is based on the chacha stream cipher. It can be seeded for better security but the DeRec implementation does not use that feature. Despite being considered cryptographically secure, it remains a

pseudo-random number generator (PRNG) whose security cannot be compared with the true random generators (TRNG) usually present in secure devices. The API should have offered the ability to plug external random number generators.

2) Concatenation of the raw signature with the plaintext message is considered as a bad practice and can lead to several security issues. This has been addressed by cryptographers by the *keyblock* structure which has been standardised in ISO and PCI. E.g a “header” must be added .
3) The API doesn’t provide any mechanism for key exchange such as Diffie-Hellman. It also doesn’t provide elements of randomness such as nonces, which apparently has to be done at the level of the message building itself.
4) The use of ECIES is generally discouraged, while this is a valid and secure encryption scheme, it cannot really cipher adequately large amounts of data. Using it incorrectly with data that are larger than the encryption key can also create security issues. Since no checks are done on the message size to encrypt, this could be seen as a flaw in the protocol.

	$(K^{PUB})^{helper}_{sign}$	$(K^{PRIV})^{helper}_{sign}$
helper	Send to sharer	Sign message
sharer	Verify message	-

	$(K^{PUB})^{sharer}_{sign}$	$(K^{PRIV})^{sharer}_{sign}$
helper	Verify message	-
sharer	Send to helper	Sign message

	$(K^{PUB})^{helper}_{enc}$	$(K^{PRIV})^{helper}_{enc}$
helper	Send to sharer	Decrypt message
sharer	Encrypt message	-

	$(K^{PUB})^{sharer}_{enc}$	$(K^{PRIV})^{sharer}_{enc}$
helper	Encrypt message	-
sharer	Send to helper	Decrypt message

[3] gives some description of the overall pairing system but contains some contradictions (issues raised on github). At the current version of the DeRec pairing system, public keys are (apparently) mutually exchanged via QR-codes only. This is a valid system in itself as long as users can check that their own public keys are correct, e.g. that they know these keys and can visually check their correctness. With a QR code it's difficult but not impossible for a trained eye to check the correctness of the data, otherwise a malicious application could display other public keys (public key poisoning).

SSSS implementation

There is *one notable specificity* in the SSSS implementation by DeRec: the value of q is not $2^8 = 256$ but instead:

q=400240955522166739341778982573590415655688281
993900788533205813612403165049083786444268762912901566403789427255978 7.

This is a prime number used as the base field for the BLS_12_381 signatures (present in some cryptocurrencies like Zcash for instance). An element of F_q represents then 381 bits, which allows less repetition of the polynomial generation process for sharing one secret. Potentially this has been chosen as a ZKP-friendly environment for future features. Apart from that, the secret is defined by $f(0)$.

SSSS implementation can be found at : https://github.com/derecalliance/cryptography/tree/main/src/secret_sharing .

It is composed of four files written in RUST :

- `shamir.rs`
- `utils.rs`
- `vss.rs`
- `mod.rs`

`shamir.rs` contains the “classical” SSSS implementation that we already described. `vss.rs` contains the specific DeRec logic on top of the “classical” SSSS.The two other files are unimportant.

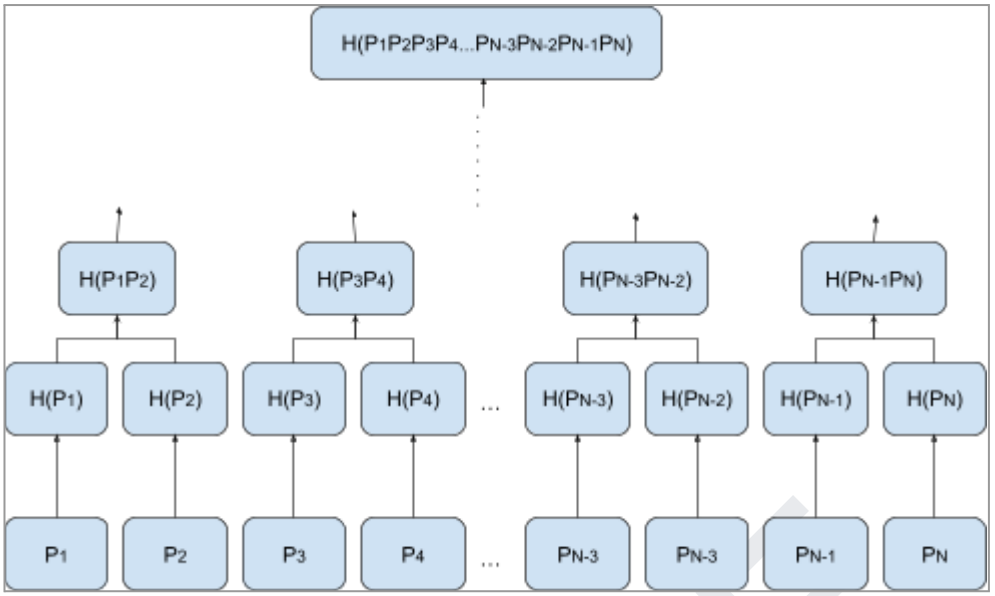
The DeRec specific logic uses a Merkle tree with a tree depth equal to seven.

A secret share is defined as a `VSSShare` structure as follows:

```
pub struct VSSShare {
    /// we use the x-coordinate to uniquely
identify shares
    pub x: Vec<u8>,
    /// we use the y-coordinate as the share
    pub y: Vec<u8>,
    /// AES encryption of the secret message
    pub encrypted_secret: Vec<u8>,
    /// Merkle-root commitment to all shares
    pub commitment: Vec<u8>,
    /// bottom-up Merkle authentication path
    /// bool denotes isLeft, while vec<u8> is
the SHA-384 hash
    pub merkle_path: Vec<(bool, Vec<u8>)>
}
```

The abscisse (e.g x-component) determines the number of the share (from 1 to t) . Besides the Shamir share itself (y) , a `VSSShare` contains additionally:

- The secret data encrypted with AES-GCM by the key $K = f(0)$ (the key k is the share as mentioned before)
- A commitment vector
- A path through a Merkle hash



(note: in this Merkel tree illustration, we note P_i the shares but in the context of DeRec these are in reality the VSSShare structures which are stored.

The concept is to store all the shares in the bottom of a Merkle tree. The commitment vector is used for the authenticated data in the AES-GCM algorithm (AES-GCM is AEAD) . As mentioned in [3] a serious caveat is that the Merkle tree that corresponds to the share must be available to the user, which is certainly not guaranteed. In order to fix this issue, DeRec stores the root of the Merkel tree (e.g. H(P1P2P3P4...PN-3PN-2PN-1PN) in the above illustration). In terms of design this is extremely debatable to store the root of a Merkel tree within the leafs of that same Merkel tree and relates more to the quick fix of a design flaw than something thoroughly thought.

The Merkle path contained in each share allows obtaining a Merkel proof (of opening or of insertion) by navigating from the share to the root and verifying that the end hashes are identical. The hash used is SHA-384.

Since the only reliable knowledge of the Merkel Tree is via the (common) root hash contained in each share, some

specific process must be in place to determine the “real” Merkel Tree in case of incorrect, erroneous, corrupted or malicious shares. A simple “majority rule” is then used (e.g the right Merkel Tree is the one that the majority of the shares contain)

General mechanism of the Decentralised Recovery

[3] describes the DeRec system rather at a high-level.
The description of the pairing system is contradictory and there is no precise information about how helpers and sharers should pair. The current protocol proposes to use QR codes in order to mutually exchange keys, then using them to cipher and sign data as explained in the sections above. Note that there is no code sample or API available in order to use QR code to pair. In such configuration “pairing” isn’t exactly the right term since there is no real cryptographic mutual authentication at the start but rather a basic key exchange without any cryptographic mutual key agreement. (note: there are some attempts to describe some mutual authentication for pairing in [3] involving a transport URI but it’s so vague that it cannot really be used at that stage)

The recovery in itself, after pairing is simple. The sharer will attempt to retrieve as many shares as it is needed to recover the secret from the helpers. Each time a new share is obtained, the sharer client software will try to recreate the share. It’s worth noting that there is apparently no information in the shares about the threshold value. E.g nobody knows exactly the value of the quorum t unless that value was written, stored or remembered by some of the participants. The Merkle tree and the AES-GCM safeguard against errors and make sure that only the right recovered key will be able to decipher the secrets (which

are stored as a copy in each share - the secret can be stored in only one share as this seems permitted by DeRec).

The distribution function of the shares, e.g Φ depends also on the implementations of each DeRec software and it cannot be assumed that the sharer (or any helper) knows that distribution function. As a consequence the recovery process is a lot about trial and error.

[3] proposes a variety of high-level future scenarios in which the pairing *could* be automated via third parties, banks etc … Additionally it is mentioned that the authentication between the sharer and the helpers is to the discretion of the implementations.

Software Implementations

DeRec offers an API made only of interfaces, thus giving a great level of freedom to implementers, designers and developers. The interfaces are defined in : https://github.com/derecalliance/api-java/tree/main as Java interfaces.

Any software offering Decentralised Recovery must therefore target these interfaces. We note that :

1) DeRec does not propose an universal interface but only a Java one. However this can be easily converted into a variety of languages (C++, C#, Rust….) .
2) The cryptographic libraries are fixed and written in Rust.

The functionalities to implement are :

- Pairing process , secure channel, key exchanges (must use the RUST encryption and signature API)
- Distribution of shares to helpers
- Encapsulation of the sharing process
- Verification that helpers are still active and have stored the right shares (by using a challenge-response mechanism with a hash and a nonce over the shares, assuming the shares are stored in the sharer application which is not in recovery mode)
- Recovery mode with eventual additional verifications of the helpers and sharer

Security Analysis of the Decentralised Recovery

The DeRec system is based on SSSS which is considered as an excellent and very secure mechanism for sharing secrets and used in X9.24 for instance.
However all the concepts provided by DeRec have been already implemented by financial organisations in the context of key management. For obvious reasons a bank cannot afford to lose their secrets and use sophisticated systems for storing and recovering them in case of problems. The helpers are then named “Key Custodians”. DeRec brings such a concept to the “general audience” and provides a smart and easy way to recover secrets. But there is one caveat: not everybody is trained in cryptography and key management such as banks key custodians and therefore the choice of the helpers could lead to dramatic situations in terms of security.

Here are some of the main caveats of DeRec:

1) The wrong choice of the helpers can lead to the impossibility of recovery, because of disagreements,

social conditions, changes in life etc … (think of the situation where a notary must look for the whole list of heirs in order to proceed with a legacy…)

2) Helpers could collude easily especially if the secrets are linked to an important amount of money, in such case, given the fact that worldwide communications are easy, it would be enough that a quorum of helpers collude in order to steal secrets and access potentially some vault. It would be easy to develop softwares to help such people to collude and unlock secrets.
3) Over time, helpers will change their devices, buying more sophisticated models which means that after a year probably most of the helpers would have no more access to the share and would need to re-pair. That situation can be extremely inconvenient and lead to an ‘extinction' of the amount of helpers.
4) Since in most cases shares would be stored outside a cryptographic secure boundary, malwares could collect them into some databases where malevolent persons could re-assemble them and use or sell secrets. It’s worth noting that banks only store shares in KLDs, SCDs, HSMs etc …
5) Verification that shares are actively kept means a cryptographic “ping”, e.g. permanent data transfer between the sharer and the helpers involving a hash of the share concatenated with a public nonce which also could become inconvenient and hard to implement.

Since implementers have all freedom to act, these caveats could be solved by an adequate design.

Improvement of security by using smartcards

Alternatively to using smartphones/devices, smartcards could be used as they fit the requirement of SCDs (Secure Cryptographic Devices) required by X9.24 secret splitting and secret sharing. In such a configuration, the smartcards would be used by the helpers, they would pair - via NFC - with the sharer's device and store the DeRec shares structures.
During recovery, the same process can happen and the smartcard is paired with the sharer which is in recovery mode and the shares structures are sent to the Helper's device. The only features that could not be directly implemented are the checks but an application could easily provide a proof of this by asking the helper to present the card "from time to time" (twice a year for instance) . The card doesn't have to implement SSSS (this would be implemented in the sharer's applications) but "only" to pair, store the DeRec share and eventually answer to a response-challenge proving that it stores the share.
A smartcard is a small object that can easily be stored, carried away and protected by a PIN for instance. It has a very long life and is shockproof. Therefore making it the ideal candidate for a universal Derec device for Helpers.

Conclusion

DeRec is a good idea with a good design with some few flaws and caveats that can be corrected by implementers.

Smartcards should naturally find their place as the recovery device provided to helpers. One could imagine a commercial set sold with ten or more cards for helpers and an application for sharers that can be downloaded from google play or apple store for instance. It's interesting to note that no network or blockchain is required at all to make it work!

References

[1] How to Share a Secret, Adi Shamir Massachusetts Institute of Technology

[2] SLIP-0039 (https://github.com/satoshilabs/slips/blob/master/slip-0039.md)

[3] The DeRec Protocol, DeRec alliance (https://github.com/DeRecalliance/protocol/blob/main/protocol.md)