IC Vault - Seamless E2E Encryption

1 Abstract

The goal is to enable secure synchronization of data between devices via the IC. We require end-to-end encryption, i.e. the IC does not get to see any cleartext.

For the scope of the hackathon we assume the devices belong to the same user. Furthermore, we assume that the devices are already added to the user's Internet Identity. We want to leverage this fact and thereby demonstrate the power of Internet Identity. By "seamless" we mean that no additional communication between the devices is required, i.e. no scanning of QR codes by one device on another, etc. Each device only communicates directly with the IC. Devices can be remote from each other.

2 Design

Canisters involved are:

- Syncing canister S
- Application canisters A, A',... (for the hackathon we only implement one application canister)

2.1 Syncing canister S

The syncing canister S maintains for each caller principal P an isolated internal storage. Here, P is derived for a user with several devices (same P for all devices of the same user). "Isolated" means that two different callers cannot access each other's storage, neither read nor write. (Though forbidding read access to a different caller is not crucial for security.)

In each caller P's internal storage, S maintains the *device list* `D`. It is a list of pairs

```
(pk, nickname)
```

where `pk` is a public key and `nickname` is a device nickname (a string) such as, e.g., "myphone", "mymacbook" etc.

It will be enforced that nicknames are unique, i.e. there cannot be two different entries in `D` cannot have the same `nickname`. We do not have to enforce uniqueness for `pk`, though it will naturally be the case.

Update call `register_device(pk, nickname) -> Bool`. The pair `(pk, nickname)` is appended to the device list. If there is already an entry with the same nickname then this leads to an error and nothing is appended and the return value is false. Otherwise the return value is true.

Update call `remove_device(nickname) -> Bool`. The entry with the given nickname is removed from the list. Return true if the nickname existed, otherwise false.

Query call `get_devices()`. Returns the list of all nicknames.

In each caller P's internal storage, S maintains the *ciphertext list* `M`. It is a list of pairs `(pk, E)` where `pk` is a public key and `E` is a ciphertext.

Query call `get_unsynced()`. Returns the list of all `pk`'s that appear in `D` but not in `M`.

Query call `isSeeded()`. Returns true if and only if `M` is non-empty.

Query call `get_ciphertext(pk)`. If `pk` does not appear in `D` then error "device not registered". Else if `pk` does not appear in `M` then error "not yet synced". Else returns the `E` from the entry `(pk, E)` in `M`.

Update call `sync([(pk, E)])`. For each item `(pk, E)` in the list that is the argument, do:

- if `pk` does not appear in `D` then skip the item,
- else if `pk` already appears in `M` then skip the item,
- append the item to `M`

Update call `seed(pk, E)`. If `pk` does not appear in `D` or if `M` is non-empty then the request is dropped. Otherwise, initialize `M` with `(pk, E)` as the first entry.

2.2 Browser local storage

The local storage is specific to a given frontend/origin. Each frontend/origin has its own persistent local storage. The local storage contains a secret key `sk` and a device nickname `nickname`.

The shared secret that is used by the application is not stored in the browser's persistent local storage. Instead, it exists only in the syncing canister (encrypted) and in the browser's transient memory (cleartext). To steal the shared secret an attacker has to gain access to the persistent local storage (to steal `sk`) *and* login via II (to obtain the encrypted shared secret). Alternatively, an attacker has to gain access to the persistent local storage (to steal `sk`) *and* collude with a node provider that hosts canister S.

Alternatively, the shared secret could be stored in persistent browser local storage. It would be less secure but the user would not have to log in with II anymore when using the application.

2.3 The process

The process is as follows:

2.3.1 Loading the frontend and login with II

This is nothing new, we just recall how II works:

- User U with browser B loads frontend code C from canister A
- This creates an “origin” O in the browser. The origin O is bound to canister A’s URL from where the frontend code C was loaded.
- The frontend code C re-directs to the II and the II window pops up
- User U enters their identity anchor N
- User U authenticates with the device hardware which is already known to the II
- User U allows the origin to use their II
- Principal P is derived from the identity anchor N and origin O. A different application canister A’ would have a different origin O’ and that would result in a different principal P’.
- II signs a delegation for Principal P to the device key
- II pop-up redirects browser to origin O, passes the delegation to frontend code C.
- C can now make requests as P.

2.3.2 Frontend startup part 1

Part 1 initializes the public key:

- Frontend code C checks if the local storage exists (on first run it doesn’t).
- If “no” then C registers the device:
    - generates a public keypair `(sk,pk)`
    - asks user U for a device nickname (such as “myphone” etc)
    - stores secret key `sk` and `nickname` in local storage
    - calls `register_device(pk,nickname)` on syncing canister S
    - if this returns an error then tells the user that nickname already existed and asks whether the user wants to a) overwrite the existing nickname, or b) choose a new one
        - if a) then calls `unregister_device(nickname)`
        - if b) then reads new nickname from user
        - goes back to the step of calling `register_device`
- If “yes” then C:
    - loads `sk` and `nickname` from local storage
    - calculates `pk`

2.3.3 Frontend startup part 2

Part 2 initializes the shared secret:

- if `isSeeded()` returns false:
    - asks the user whether to seed now and aborts if user says “no”

        - attempts to seed:
            - generates secret `k'`
            - encrypts `k'` with `pk` to get `E'`
            - calls `seed(pk, E')` on S
    - retrieves shared secret:
        - calls `E <- get_ciphertext(pk)`
        - decrypts ciphertext `E` to `k` using `sk`
        - stores `k` in local storage
    - kicks off polling (see below)

Frontend code C can now interact with A using shared secret `k`

2.3.4 Polling to sync

At regular intervals, frontend code C helps to sync other (new) public keys:

- calls `get_unsynced()` to get a list of public keys
- encrypt `k` for all returned public keys to get a list of ciphertexts
- calls `sync` with the list of all returned public keys and the computed ciphertexts

Polling is kicked off at the end of startup part 2, i.e. as soon as the shared secret `k` is available.

EDIT: Calling sync on S requires to be logged in with the II. The delegation is only valid for 30 min. after a user has logged in. Therefore, polling can only continue for that time.

2.3.5 Settings

The frontend code C has a “Settings” tab which displays various information. When opening “Settings” C calls `get_devices()` on S and displays all device nicknames. C also displays the present device’s `nickname`. The user can choose to remove devices by nickname in which case C calls `unregister_device(nickname)` on S.

2.4 Other considerations and notes

As a simple DoS protection we can limit `D` to max. 100 entries.

When local storage is lost, i.e. a device loses sk and nickname, then the next run will automatically re-generate sk, pk, nickname and will re-register the device by calling `register_device` again. If the user has forgotten the device’s nickname and happens to choose a different one than before then there will be a dangling entry in S. We accept that fact. The user can later clean up the list of devices in the “Settings” tab.

2.5 Simple application canister

The simplest application canister maintains a single text field per calling principal P. In other words, the application is a “private clipboard”.

Update call `write(ciphertext)`. Stores the text `ciphertext` under caller P.

Query call `read()`. Returns the text stored under caller P.

The frontend code C allows to enter/view cleartext. It encrypts/decrypts between ciphertext and cleartext with the shared secret `k`. It calls `write/read` on `A` in the background.

Here, `k` can be used as a symmetric key and the encryption can be symmetric. Alternatively, `k` can be used as a secret key of an asymmetric encryption scheme and the encryption can be asymmetric. The latter approach may provide future extendability towards sharing items between different users.

3 Implementation

A single syncing canister can serve multiple application canisters. In the hackathon we’ll write only one application canister.
Syncing canister and application can be written in different languages. In our application, none of them requires cryptographic functions. They are very simple. We use Motoko for both.

The frontend code coming from a single “origin” (A’s asset canister) has to be able to talk to both canisters (A and S). To obtain some degree of code separation and re-usability the part talking to S could be separated into a library.

It may be possible to have two asset canisters per application. One would serve the frontend code talking to S, the other would serve the frontend code talking to A. The two are then re-directing between each other. The point of this is that the first asset canister (with the code talking to S) is re-deployable unchanged by a new application. The new application just needs to redeploy it under a new canister id so that when talking to S it gets its own isolated storage. That way the two applications do not have to trust each other. This approach may also generalize to allowing shared secrets between two mutually trusting applications. Mutually trusting applications would deploy the first asset canister only once and they would all redirect to the same one.

The frontend code requires use of cryptographic libraries and we have to choose a public key encryption scheme for use with S and an encryption scheme (symmetric or asymmetric) for use with A.

4 Applications

- Encrypted clipboard. The simplest application already described above in 2.5.
- Encrypted notes. Store encrypted notes and sync them between devices.
- Plug wallet. The Plug wallet by fleek acts as an “identity provider”. It stores a secret in its local storage. That secret is bound to the user and is used across all apps that the user interacts with (and that integrate with Plug). Installing the Plug wallet on a second device requires that the user re-enters the secret there. Plug wallet could adopt our solution and use II to sync the secret between devices.

The applications can be extended with sharing features between users.

4.1 Password store

Structure: key/value store mapping string to “username:password” pair. Provide ability to add, edit and delete items. To be able to do this easily despite encryption we use a deterministic scheme.

5 Alternative designs

The current design described in section 2 is characterized by the fact that each combination of user and application has its own isolated storage inside S. In particular, for a given user, different applications have isolated storage inside S. As a consequence, a given user maintains a separate device list for each application, each application generates its own secret and each application does its own syncing/polling in the background.

An alternative design places one canister M, the *master application*, between S and A. Canister M is an application canister in the previously established sense. It uses S to sync a single secret between devices. This secret is called the *master secret*. For each application A that wants to use this service, A’s frontend code opens up a pop-up window in which it redirects to M. This is similar to the pop-ups known from the II. Within the pop-up window runs code served by canister M. That code has access to the master secret. The user approves that application A should get access to a derived secret. The code in the pop-up window then derives a per-app secret for A from the master secret and passes it back to A’s frontend code.

The different applications A, A’, … can be mutually untrusting as they receive different shared secrets. But the user has to maintain only one device list and the syncing happens only once.

The user experience consists of a chain of two pop-ups. User goes to A, then M pops up and immediately II pops up. User approves M in the II window, gets redirected to M. User approves A in the M window, gets redirected to A.

6 FAQ

Q. What if the user wants to use different device groups such as using devices 1,2,3 with application A and 1,2 with application B.

A. The user here thinks that application B is higher “value” than A and that device 3 is not trusted enough to handle B. The correct way of solving this is to create two II anchors and don’t add device 3 to the second one.

Q. What if we want to share secrets between different users?

A. The correct way to do that is to let the application canister (A) handle it. As an example, we can extend the clipboard application to allow users to share their clipboard with another user where the other user is specified by the principal that the other user sees when logging in with II.

8 Components

- Canister code S
- Canister code A
- Frontend code for A
    - all the cryptography is here

Appendix

Canister IDs

KV Store canister: uvf7r-liaaa-aaaah-qabnq-cai
Key Sync canister: khpze-daaaa-aaaai-aal6q-cai
Frontend canister: https://xggrc-cyaaa-aaaaj-aaasq-cai.raw.ic0.app/

Candid UI Key Sync:
https://a4gq6-oaaaa-aaaab-qaa4q-cai.raw.ic0.app/?id=khpze-daaaa-aaaai-aal6q-cai

Candid UI KVStore:
https://a4gq6-oaaaa-aaaab-qaa4q-cai.raw.ic0.app/?id=uvf7r-liaaa-aaaah-qabnq-cai

Principals

```
dfx identity get-principal
```

Stefan: xjn6k-yuhju-le64u-3l54i-5alk5-yla6p-ht2sw-mggnw-v6bhp-grkqs-fqe
Timo: mlyaq-3g3y5-lzo54-hoiqq-6zyz7-alxpq-binhp-fbr6x-o6fso-nmnwk-fae
Rüdiger: dqsko-rrmnr-3zdsr-un5vd-qzyht-5phwg-mhbll-srjhh-o3sf5-dhxsi-dae
Helge: 7m7jk-j5gcq-wpc2a-dlvfc-p2yie-zkzpe-veptv-ed5hq-xlh25-c5dyl-lqe
Yvonne-Anne: z5e5z-j3ofi-fjj75-c3d3p-qit4m-c6qyb-xveoq-c6b7b-cfcna-o5ipo-lqe
Thomas: 24e5w-wsxmh-bh5rn-lngip-iustm-evp5e-7bbnh-zjnat-kttu6-ko7ip-fqe

TODOs

- Move clear local store button to Debug/Maintenance, move maintenance to after vault
- Password lists could be sorted alphabetically for usability
- Periodic syncing (in hackathon version explicit syncing by clicking on a button is necessary) to enable dynamic content in the password table on several devices
- Separate key sync calls in frontend to library