Story UFC-258: Initial Library Implementation

The following document will go over the design of the initial implementation of the encryption library, as outlined in Story ticket UFC-258.

Acceptance Criteria

For reference, the following are the acceptance criteria for this ticket:

- I want all of my personal data (accounts, transactions, import profiles, etc) to be encrypted using my 'password', so that the server (aka uFincs, aka the operators of uFincs) never have access to my raw finances.
- The encryption/decryption processes should work in a background thread so that the UI thread isn't too heavily impacted. It should also, ideally, be multi-threaded for even faster performance.
- The encryption process needs to be more-or-less a self contained library, so that we can open source it.

Design Brainstorming

Migrating Existing Users

So as part of implementing the library, we're gonna need some way to migrate users who have existing non-encrypted accounts.

Since the EDEK and KEK Salt get passed back from the backend on login regardless if they exist or not (they can be null), I was just thinking that the the login saga could call out to some 'encryption' saga to handle setting up the keys for the user.

The only catch is how to handle actually encrypting the user's data. Currently, we do fetches as part of the app boot process; however, these fetches will have to have the 'decrypt' meta tag added them. As such, how should we handle fetching and then *encrypting*, while also not accidentally storing encrypted values in the Redux store?

Hmm... I suppose we could kind of hack it? What I'm thinking is that the `fetchAll request` action can take in an param that dictates whether the data should be encrypted or decrypted. It'll default to decrypted, but we can pass in encrypted as part of the migration process.

Then, the migration process can call these fetch effects. The data will be fetched and processed as usual, and then the `set` actions will have the encrypt tag applied to them. The `set` actions will get picked up by the encryption middleware and have their values

encrypted. The important/niggly part is that the actions will then hit the reducers, thus storing encrypted data in the redux store.

The good thing that I realized, however, was that since this is all happening before the user has been redirected to the app, there shouldn't be any problems with this, because there won't be any components that are using the (encrypted) data. As such, we should be able to just `take` actions as part of the migration process to get the data, then clear out the stores before finishing.

Good, good, this might work.

Update: It did not work. The `transactionsDateIndex` listens for these actions and was not happy about finding an invalid date. Shit.

Well, I guess we could just use custom encryption actions or something... I mean, it's a flag so whatever...

Also, I can't figure out how to do multi patches. Turns out multi *updates* aren't a thing, and multi patches seems to only be a thing in the context of "I want to patch multiple records with the same new value", not "I want to patch multiple records with different values for each record". Which is a pain. But whatever, in theory nobody besides me will ever trigger this code.

Other Schema Updates

Yeah, so I only *just* realized that we're gonna need to change the column types on other fields... like, `amount` can't be a bigint, it has now has to be a string. Same with `date`.

So... that'll make the migration *even more awkward*. Cause then we can't even really do it client side, can we? I mean, I guess, as part of the actual sequelize migration file, we could just convert all `amount` fields to strings, leave dates as is, and it'd *probably* be fine? Even without any of the encryption stuff, because amounts have the handling in place for strings regardless, and dates don't need any special handling. So... it *should* be fine?

Ugh, I hate. I hate it all.

But it's gonna be *sooo* worth it... right?

On second thought, there really isn't any point in encrypting the date field is there? Because the `createdAt` and `updatedAt` fields aren't going to be encrypted, and they leak just as much data.

- BTW, I decided not to encrypt the timestamp fields because 1. we weren't even using them on the import profile/mapping models, and 2. they can somewhat be considered to be auto-generated (at least, in some instances) and I don't want to have to deal with taking full control of them. So no point in encrypting them then.

Then I guess it's only the number fields that we have to change over then.

Error Handling

There are currently three main encryption related error cases that we need to handle:

i. When we fail to unwrap the EDEK (at login).

ii. When we fail to read the encryption keys from storage during app boot (post login).

iii. When an encryption/decryption event fails (as part of app boot data fetching or general app usage).

The following sections will discuss/brainstorm ideas for how to handle these failure cases.

EDEK Unwrap Failure (Login)

This failure happens when, during the login process (but before the app boot process), the encryption middleware fails to be initialized because of one of three things:

- Wrong password
- Wrong EDEK
- Wrong KEK Salt

We *basically* don't need to care about the wrong password case, because that's handled by the act of getting the access token from the Backend. If the Backend doesn't give us our JWT, then our password/email must have been wrong, and that doesn't really fall into the purview of the encryption middleware.

However, there is the theoretical case where a user manages to login, get their JWT, EDEK, and KEK Salt, but the EDEK and KEK Salt are for someone else. As such, *technically*, the 'password' is what's wrong, since it won't be able to unwrap the EDEK (even though the EDEK and KEK Salt are 'valid' and 'untampered').

The second and third cases (wrong EDEK/KEK salt) stem from the EDEK/KEK salt received from the Backend being tampered with; that is, the user's password no longer decrypts the EDEK. This would be *really* bad. Like, catastrophically bad (at least, from the user's perspective), because if either value is tampered with, then we can't get their DEK, which means all their data is now permanently encrypted (i.e. lost).

So there are two things we need to deal with here:

i. How to resolve this situation from our end (i.e. data recovery)

ii. How to handle this situation from the UI/UX perspective (i.e. what error messages to present the user)

On the first point, I believe the only reasonable solution we have to this is to restore the users's EDEK/KEK Salt from a backup. I mean, we don't currently have the monitoring in

place to know how/who/what changed/corrupted the values, so restoring from backup seems like our only option.

On the second point, I think the best thing we can do is to direct the user to refresh the page and try logging in again; if that doesn't work, contact support.

Key Storage Retrieval Failure (Post Login)

I think this is a much simpler failure case to handle.

To describe the error, basically, after a user logs in, if they refresh the page, then their DEK should be reloaded from storage (IndexedDB). In theory, if they log in and the storage gets cleared, then they try to refresh the page, they won't have their encryption key. As such, they won't be able to operate on their data.

As for how this would practically happen, I can only make the assumption that browser storage is effectively ephemeral and that we can't rely on the keys always being there once logged in.

As such, the solution is simple: just kick the user out of the app and tell them to login again. Done.

Encryption/Decryption Failure (General Usage)

This failure case is much, much more tricky to handle properly. Let's split it up into the two sub-cases: encryption and decryption.

Encryption

The encryption failure case can happen as result of the following:

i. The payload is seriously fucked up. Like, I can't even simulate what could cause an error. The `TextEncoder` can seemingly encode anything (including `undefined`, `null`, and non-string values). So... ok, I guess this isn't a cause of encryption failure.

ii. The DEK.. isn't.. available?

OK, I guess encryption is just a *lot* harder to screw up. Especially if we have all the checks and balances in place that come before an encryption event is even allowed to happen.

In any case, if the encryption were to fail *for some reason*, what should happen?

Uhhh... rollback? But that would require us to further integrate the encryption process into the request handling process.

I think we could do something like the following: if we generate an error while trying to encrypt a payload, we can attach an `error` field to the action (as it conforms to FSA actions). Then, we can modify the `effectSagaWrapper` of the `OfflineRequestSlice` to check for any errors on the action and emit an `effectFailure` when that happens.

Yeah, that could work. The user won't know *why* things are failing, but at least the app won't completely break!

Decryption

The decryption failure case is *much* more likely to happen (because it has happened before!). Here's how things can hit the shitter:

i. The DEK isn't available (duh).

ii. The DEK doesn't work on the ciphertext.

iii. The user's ID is wrong (associated data is wrong).

iv. The iv/ciphertext is corrupted/poorly formatted.

The 3rd case is the one I've experienced the most. If a field gets improperly encrypted (or somehow doesn't get encrypted *at all*), then the iv/ciphertext splitting algorithm will find that one or the other part is `undefined`. Which will then cause the string decoder to blow up because `atob` specifically has throws an error on `undefined` (but not `null`...).

Obviously, all of the times I experienced this problem was because I had improperly implemented something else unrelated, but it just goes to show that it's *possible*.

So how do we handle a decryption failure?

Uhh... good question. Well, considering decryption (currently) only applies to `fetchAll` effects (which don't have a rollback), well, we can't exactly rollback.

I guess we have two options: we can either go for progressive failure or halt-the-world failure.

- In progressive failure, we note an error decrypting a single field but still decrypt everything else (assuming everything else also decrypts fine). Then, we notify the user that there was a problem decrypting some of their data, but they can continue using the app.
- In halt-the-world failure, we completely halt the app to tell the user that their data couldn't be decrypted.

In either case, what is the user supposed to do with this knowledge? Contact support?

And what are we, the 'support', supposed to do about it? At best, users (or an automated error process) would be able to give us the ID of the record, the type of the record, and maybe the dates (`date` for transaction, else `createdAt/updatedAt`). What would we even do with this information?

The best we can really do is a visual check to see if the record is obviously corrupted (e.g. it wasn't encrypted in the first place). Otherwise, we can really only throw out the data.

An idea I had was, at least as a placeholder solution before support deals with things, was to mark un-decryptable records as such (i.e. add a 'corrupted' column to every row) so that the decryption process can at least ignore those records. And then maybe we can give the user an interface for inspecting this data, with the option of throwing them away.

- But this is an advanced use case that is out of scope of the current story. Maybe later.

I guess for now we'll just have an error toast that says "Sorry, something went wrong. Contact support."

Tasks

- Build the initial version of the library.
- Integrate it into the app.