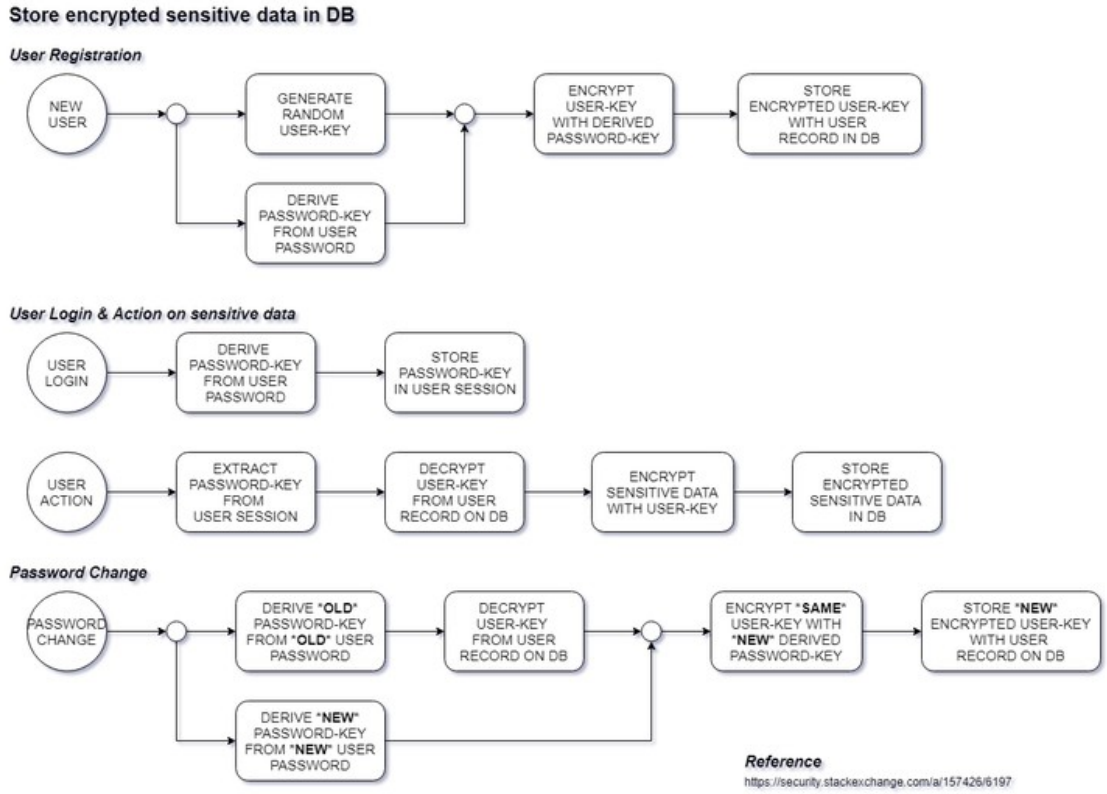
Library Design

The following will describe a design of the `redux-e2e-encryption` library. For more details on how this library will be used and why it is being created, see Epic UFC-257: Client Side Encryption and Idea: Client Side Encryption.

Data Flows

Here we're going to elaborate on how the encryption process affects the key data flows of uFincs.

Everything can more or less be summarized the following diagram:



- Store encrypted user data in database

This is additionally illustrated through the following code samples:
SubtleCrypto.wrapKey()

Signup

The following will demonstrate how the encryption process affects the sign up process.

Frontend:

i. The user provides the email and password for their new account in the Sign Up form.

ii. They acknowledge that losing their password will result in the loss of all their data.

iii. They submit the form.

iv. Generate a new `AES-GCM` key using `generateKey`. This will be the user's Data Encryption Key (DEK).

   - The DEK should be set as extractable; this is because it will need to be exported and stored in the database.

v. Generate a salt to be used for generating the Key Encryption Key (KEK).

   - Keep this salt around, cause we're gonna need to store it in the database.

vi. Derive the user's Key Encryption Key (KEK) using their password and the salt from step 5.

   - This is will be done with `importKey` to generate the key material using the user's password and then `deriveKey` to create an `AES-KW` key (from SubtleCrypto.wrapKey(), this is known as the 'wrapping key') using `PBKDF2`. Note

   - The KEK should *not* be set as extractable; it should be derived each time the user logs in and stored using the WebCrypto storage.

vii. Use the KEK to encrypt (wrap) the DEK using `wrapKey` to get the encrypted DEK (EDEK).

viii. Send the request to the create the user account to the backend, including the user's email, password, EDEK, and KEK salt.

   - Will need to encode the EDEK and KEK salt into base64 first.

Backend:

i. Accept the request to create the user, and store the EDEK in the users table.

Login

The following will demonstrate how the encryption process affects the login process.

Frontend:

i. The user provides their email and password in the Login form.

ii. They submit the form.

Backend:

i. Accept the request to login.

ii. Lookup the users EDEK and send it along with the auth JWT to the frontend if the login was successful.

Frontend:

i. Show a loading screen to the user if the login was successful.

ii. Derive the user's KEK using their password (see the Signup flow above for the process).

iii. Use the KEK to decrypt (unwrap) the EDEK using `unwrapKey` to get the DEK.

iv. Fetch the rest of the user's resources (transactions, accounts, etc).

v. Decrypt all of the user's resources using the DEK and store the results in Redux.

Resource Requests

The following will demonstrate how the encryption process affects the resource request process, through an example use case with transactions.

Frontend:

i. The user fills out the transaction form.

ii. They submit the form.

iii. The `request` action to create the transaction is emitted, along with the meta tag indicating this action should be encrypted.

iv. The `redux-e2e-encryption` middleware picks up the action.

v. The middleware encrypts the fields of the payload.

vi. The middleware re-emits the `request`, but with the encrypted payload and without the meta tag.

vii. The `request` action is picked up by the associated offline request slice sagas and handled normally.

WAIT, WAIT THIS DOESN'T WORK.

If we operate on the `request` actions of *offline* request slices, then we'll end up storing the encrypted data in the Redux store as well. Why? Because we're offline first! The `request` payload goes straight to the store before being sent to the Backend.

So what do we have to do? Annotate the `effect` actions instead, since those are the ones that get sent to the backend.

However, a consequence of this is that we'll need to modify the encryption middleware slightly: it needs to put the original payload *back* on the action, just under a different key. So, after encryption, `payload` will be the encrypted payload and `originalPayload` will be the original.

The reason we need to do this is so that the offline request slices can take the original (unencrypted payloads) and use it for the client-side rollback process. If we only have the encrypted payloads, then the rollback process can't work.

- **Update**: Actually, it seems like we *won't* need to worry about changing the rollback process, because the `rollbackData` is derived from the `commit`, not the `effect`.

So, here's the revised flow:

Frontend:

i. The user fills out the transaction form.

ii. They submit the form.

iii. The `request` action to create the transaction is emitted and handled normally.

iv. The `effect` action is emitted, along with the meta tag indicating this action should be encrypted.

v. The `redux-e2e-encryption` middleware picks up the `effect` action.

vi. The middleware encrypts the `payload` and puts the encrypted payload on the `payload` key; it puts the original (unencrypted) payload on the `originalPayload` key.

   – The exact details of the encryption process will be described as part of the design of `redux-e2e-encryption`.

vii. The middleware re-emits the `effect`, but with the encrypted payload and without the meta tag.

viii. The `effect` action is picked up by the associated offline request slice sagas and handled normally.

ix. If the `effect` needs to be rolled back, the rollback process uses the `originalPayload`.

Changing Password

The following will demonstrate how the encryption process affects the password change (*not* password reset) process.

Frontend:

i. The user fills out the password change form with their old password and their new password.

ii. Derive the user's old KEK using their old password and old salt.

iii. Derive a new KEK using the new password and a new salt.

iv. Use the old KEK to decrypt the EDEK to get the DEK.

    - Somewhere in this process, we need to make sure the user's old password is correct. This can come in the form of authenticating to the Backend to refresh the JWT, or some part of the KEK derivation/EDEK decryption process failing.

v. Encrypt (wrap) the DEK with the new KEK to get a new EDEK.

vi. Send the new EDEK, new KEK salt, and new password to the Backend.

Backend:

i. Update the user's password, EDEK, and KEK salt.

Migrating Existing Data to Encryption

I am currently the only active user of uFincs (at least, as far as I know). As such, I could probably get away with just manually encrypting my own data and just running the crypto stuff as if there was never any need to migrate.

However, for external consumers of the library, they probably won't this luxury. As such, we need to provide some way to migrate existing non-encrypted data to encrypted data.

I wonder if this could be as simple as just using something like a flag column on the user's account for indicating whether they've been moved over to encrypted data or not. This way, the frontend could know whether or not to perform the encryption migration. That's one part of it -- knowing whether or not to perform the migration.

- Actually, instead of having a dedicated flag, couldn't we just check for the presence of the EDEK/KEK salt? If we don't have both of these things, then the user *couldn't* have been using encryption yet. Therefore, this could be the trigger for the encryption migration.

The other part is the actual migration. We'll need to provide some interface for just manually encrypting a bunch of data so that it can then be posted back to the Backend.

Here's how I'm envisioning the data flow:

i. User logs in.

ii. Backend sends back the user's information; the EDEK and salt are empty.

iii. Frontend receives the empty salt and EDEK.

iv. Frontend triggers the 'signup' flow to create the salt and EDEK.

v. Frontend requests all of the user's data.

vi. Frontend encrypts all of the user's data and posts it back to the Backend.

In practice, since the library middleware is the 'interface' for consumers, this process will really be handled by a series of actions.

Alternative Interfaces

That last sentence from above ("since the library middleware is the 'interface' for consumers") made me realize something: using the middleware as the single interface for interacting with the encryption process really only works for *us* (specifically, us, the people who use sagas).

Think about it: we pass through actions, encrypt their payloads, and then fire off new actions. If we weren't using sagas, what would we do with these encrypted actions? They'd just hit reducers, which is pointless, cause we don't want to store encrypted data in the store.

If we look at this from the perspective from a `redux-thunk` user, where are all of their fetch requests are done in thunks, this middleware pattern is *utterly* useless. If they're using `fetch` (or whatever other request library) directly, then they'd want to be invoking our `crypto` module directly; they'd be imperatively handling the encryption process.

Of course, that's like the worst case scenario. An improvement would be, instead of using `fetch` directly, they had something like an `api` class that wrapped all of their HTTP requests (ala our Feathers API). Then we'd be able to do something like wrap the API in a proxy to handle the data encryption/decryption.

The point is this: depending on how people use Redux, there are multiple different ways we could expose the encryption process to users.

Obviously, we'll implement it for our needs first and then go from there. Just something to keep in mind as we design and build.

Exported Interface

The following are the exports of the library. That is, this is the user-facing portion of the library:

i. `createEncryptionMiddleware`

   - The main export; this is what users use to generate the middleware that they can then register on their store.

ii. `crypto`

- A utility export; exports the `crypto` module (see below) so that users can make use of our crypto API if they so desire.

Modules

There are 2 main modules for the library:

i. The `crypto` module, which wraps the `WebCrypto` interface when used client-side (and wraps the Node `crypto` module when used server-side?).

ii. The `middleware` module, which create the middleware that is used by Redux to intercept and encrypt actions, and leverages the `crypto` module to do this.

Then there's the secondary modules:

i. The `schema` module, which enables parsing action payloads (objects/arrays) to know which fields to encrypt/decrypt.

ii. The `worker` module, which handles taking the actions from the middleware and performing the actual work (from parsing with the `schema` module to encrypting/decrypting with the `crypto` module).

Design Brainstorming

The following will be any brainstorming required to design the library.

Middleware Module

So I was doing some research on *how* to actually write a Redux middleware and it seems to be pretty straightforward. Basically, it's just a function with the following signature:

```
store => next => action => result
```

I mostly already knew this. But what I was most curious about was what `result` could be. Like, can we do async calls in a middleware? It *seems* like we can. For example:

- From `redux-promise`: redux-utilities/redux-promise

Additionally, it also seems pretty easy to integrate web workers into a middleware. You basically just send a message to the worker from the middleware and the worker is responsible for emitting a *message* (which would be the action) that the middleware can then pick up for dispatch. This way, the worker doesn't need access to the store functions (i.e. `dispatch`) directly. For example:

- From `redux-worker-middleware`: keyz/redux-worker-middleware

I feel like the architecture of `redux-worker-middleware` is more along the lines of what we want.

Also, if we're going to be using web workers, we probably want some sort of pooling, so that we can *really* multi-thread this bad boy.

Initializing the Middleware

There will be two stages to the middleware initilization: 1. when the middleware is created for registration on the store, and 2. when the user logs in.

i. Middleware Creation
   - The model schema should be provided.
   - The web worker pool should be created.

ii. User Authentication
   - A Redux action should be dispatched upon the successful authentication of a user.
   - This action will contain the user's ID, plaintext password, EDEK, and KEK salt.
   - The action will be picked up by the middleware and used to get derive DEK.
   - A message is then posted to the Web Workers containing the DEK and user ID, so that they can cache them for data encryption/decryption.
     - This means the worker process is going to need the same concept of action types that Redux uses.

Handling Encryption/Decryption

Encryption/decryption should be handled by applying different meta tags to the actions. This is what the middleware will use to differentiate actions it should/should not deal with.

Additionally, since we're doing field level encryption, the tag should specify a 'model' to use for encrypting/decrypting the payload. This means the consumer will need to configure a schema of models specifying which fields to encrypt.

Actions

The middleware will need a set of dedicated actions for performing various things (e.g. passing in the keys). As such, here's the list of actions we'll need:

- `redux-e2e-encryption/LOGIN`

Crypto Module

The `crypto` module will have the following interface:

- `generateKEKSalt() → salt`

- `generateKEK(password, salt) → kek`
- `generateDEK() → dek`
- `wrapDEK(dek, kek) → edek`
- `unwrapDEK(edek, kek) → dek`
- `*encrypt(dek, plaintext, associatedData) → ciphertext`
- `*decrypt(dek, ciphertext, associatedData) → plaintext`
- `*generateKeysForNewUser(password) → kek, dek, salt, edek`
- `*generateKeysForExistingUser(password, salt, edek) → dek`

* These are the public methods.

Storing Keys

Looks like we're gonna need to deal with IndexedDB to store the keys between refreshes (unless we want to store the raw password in localStorage and derive the keys each time).

This isn't going to be simple.

Also, don't forget we need to clear the keys when logging out.

Also, maybe a way of 'passing' the keys to the Web Workers is instead by letting them look them up in IndexedDB.

Schema Module

So the `schema` module starts by by receiving the schema from the creation of the middleware.

This schema should be a map of model names to an array of field names that will be encrypted/decrypted. For example:

```
{
    account: ["id", "name", "openingBalance", ...],
    transaction: ["id", "description", "date", "amount", ...]
}
```

We should be very strict about the structure of this schema. For example, I don't want there to be any way to implicitly specify 'all fields of a model'; no, *you're* specifying them. Otherwise, it gets ignored.

Once we have this schema, we then need to be able to use it. Essentially, we need to be able to take a given payload and a function and apply the function to the payload using the schema.

The catch is that the payload could be in one of many forms: an object that is directly a model from the schema, an object which is keyed by IDs and maps to models, or an array of models (of course, there are more forms the payload *could* take, but those'll be the ones we support).

Which means we need some way of specifying, in the action `meta` tag, both the model *and* the form of the payload.

How about we provide some methods that consumers can use to generate whatever string format we're gonna use?

So, for a string format, we could just go really simple like `single-${model}`, `array-${model}`, and `map-${model}`. Then we could have functions like:

- `single(model: string)`
- `arrayOf(model: string)`
- `mapOf(model: string)`

For the consumer, I think these methods should be exposed statically, so that they don't need to have access to the instantiated schema at the action level.

Speaking of schema instantiation, we need a `createSchema` method that.. just takes in the schema and passes it right back out. Maybe with some TypeScript validation? But that's really it.

So far, our public interface for the `schema` module is:

- `single(model: string)`
- `arrayOf(model: string)`
- `mapOf(model: string)`
- `createSchema(schema: Object)`

Payload Manipulation

Internally, we need to take the schema that the user provides and use it to instantiate a class to parse the payloads. Essentially, we'll only need a single method:

- `applyToPayload(payloadShape: string, payload: any, function: (fieldValue: string) → string)`

This will take the payload and the payload shape that was extracted from the meta tag, then apply the given function to the payload using the payload shape and schema to transform all of the specified field values.

Boom, good to go.

More Complex Schemas

Hmm, seems like our attempt to keep the schema relatively simple with only flat fields might not be enough... See, the Import Profiles currently bundles together the Mappings for creation all at once. As such, if we want to properly encrypt everything, we need to support specifying more complex schema types.

I'm thinking this could be solved by allowing users to specify that a field is an array of another model in the schema. I think the syntax for this could be a literal array with a single element -- the name of the model. For example:

```
{
    importProfile: {
        ...
        importProfileMappings: ["importProfileMapping"]
    },
    importProfileMapping: {...}
}
```

- Need to be careful with recursion -- make sure it's properly accounted and tested for.

Also, I've just learned of **json-schema** (JSON Schema). I wonder if this would be useful for us?

Worker Module

Update: OK, so I kinda didn't really design the module before implementing it... but, I did want to drop something here:

If we want to easily test multiple workers vs 1, just accidentally forget to add the `+ 1` when using the `currentWorkerIndex`. God dammit.

Or just change the size of the pool; that works too.

Benchmarking Crypto

So I did a couple of spikes to test out web workers, redux middleware, and the web crypto API.

I combined everything together to see if web workers would actually help speed up the encryption process.

I only tested encryption (not decryption) but the results were promising: not perfectly linear scaling, but close enough. By throwing 4-8 threads (web workers) at a large number of messages to encrypt, I was able to get speedups around 2-4 times over just encrypting the messages linearly.

However, there was *definitely* some overhead for calling into a web worker vs just calling the encryption locally. As such, it'd probably be most optimal to reserve a web worker pool for large encryption/decryption tasks (e.g. initial app boot or transactions import).

However, realistically, it'd just add developer overhead to switch between using web workers vs not using web workers, so it'd probably be fine to just always use web workers. The marginal cost of a single encryption call is essentially negligible (<0.5 ms) and the overhead introduced by the web workers is on the same order of magnitude, so I believe the simplicity of always using the web workers outweighs the marginal improvements in performance.

For the record, here's some numbers from benchmarking 1,000,000 strings of 10 bytes each:

- Main thread only: ~60 seconds
- 4-8 web workers: ~20 seconds

Obviously, this was kind of a stress test; 1,000,000 strings is about 100,000 transactions (since there are 10 fields on a transaction object, if we were to encrypt everything). Additionally, 10 bytes each is very realistic, if likely slightly above average.. But it's a good reference point.

Follow Up (during UFC-259, password change form)

So I noticed something recently: Firefox is *wayyyy* slower to login than Chrome. Like, more than 2.5 times slower (3s on Chrome, ~8s on Firefox). And then I noticed that it was the same when changing passwords. Why??

Well, it seems weird. First, WebCrypto operations (or at least, what we do for `generateKEK`, which is mostly a key derivation operation) are *wayyy* slower on Firefox. Roughly, 3 times slower! (400ms for a `generateKEK` on Chrome, 1200ms on Firefox). Hmm.

OK, that's one thing. I realized that initializing all of the Web Workers at login was actually happening sequentially, so on Firefox, it would initialize all 6 in a row at 1200ms = 7.2s. Then on Chrome, all 6 in a row would be 6 * 400ms = 2.4s.

OK, fine. I'll parallelize the worker initialization just like we do with encryption/decryption. This is where the *weird* shit happens.

Parallelizing on Firefox works out fine. All 6 `generateKEK` calls happen in parallel, they all take ~1200ms, the end result is a roughly 3s login time. Just like Chrome.

But parallelizing on Chrome is *completely* different. What happens is that the first `generateKEK` operation takes the 400ms, but then each subsequent one slows down by a factor of... one `generateKEK` operation! That is, the first is 400ms, then the next is 800ms, then 1200ms, etc. The end result is that there's no speedup *at all*! It still takes 3s to login! WTF!!!

I honestly have no idea what's going on here; I can only guess that this has something to do with how WebCrypto is implemented under the hood and how it defers between the two browsers.

In the end, I'm just happy that we can get Firefox's speed up to par with Chrome's.

... OK, after a bit of research, I think the problem I'm experiencing can be explained by this: 615133 - chromium - An open-source project to help move the...

tl;dr *all* WebCrypto operations (in Chrome) get posted to a *single* background thread, *regardless* of where they're coming from. As a result, running all 6 `generateKEK` calls will happen synchronously, *no matter what*, because of the asynchronous nature of WebCrypto and this one implementation detail in Chrome. Here's the profiling at the `initKeys` worker level:

```
initKeys: 2319.763916015625 ms                    worker.ts:26
initKeys: 2772.653076171875 ms                    worker.ts:26
initKeys: 2773.344970703125 ms                    worker.ts:26
initKeys: 2774.5400390625 ms                      worker.ts:26
initKeys: 2775.60498046875 ms                     worker.ts:26
initKeys: 2776.89697265625 ms                     worker.ts:26
WORKER POOL: 2778.23583984375 ms                  workerPool.ts:59
>
```

And here's the profiling at the `generateKEK` level:

```
generateKEK: 454.95703125 ms                      crypto.ts:298
generateKEK: 908.240234375 ms                     crypto.ts:298
generateKEK: 1362.843017578125 ms                 crypto.ts:298
generateKEK: 1816.73828125 ms                     crypto.ts:298
generateKEK: 2268.29296875 ms                     crypto.ts:298
generateKEK: 2727.1591796875 ms                   crypto.ts:298
WORKER POOL: 2740.098876953125 ms                 workerPool.ts:59
>
```

... WTF CHROME

Also, it's worth noting that another performance characteristic I saw in Firefox (when profiling `initKeys` at the worker level) is that the first 4 operations happen in the same amount of time, but then the subsequent 2 are slower by the amount of time of a single operation. Here:

```
initKeys: 1358.2ms - timer ended                    …834985f980bf7da7b6.worker.js:5997:11
initKeys: 1359.04ms - timer ended                   …834985f980bf7da7b6.worker.js:5997:11
initKeys: 1359.05ms - timer ended                   …834985f980bf7da7b6.worker.js:5997:11
initKeys: 1377.83ms - timer ended                   …834985f980bf7da7b6.worker.js:5997:11
initKeys: 2574.63ms - timer ended                   …834985f980bf7da7b6.worker.js:5997:11
initKeys: 2585.96ms - timer ended                   …834985f980bf7da7b6.worker.js:5997:11
WORKER POOL: 2588ms - timer ended                                        workerPool.ts:59
»
```

I think this can be explained by the fact that Firefox uses a 4 thread pool for WebCrypto operations, as opposed to Chrome's 1 thread. This is further evidenced by the following: 449009 - chromium - An open-source project to help move the...

OK, well I guess this at least explains why we saw (and continue to see) very non-linear scaling when using workers with WebCrypto -- all we're really parallelizing is our own user land code; all of the crypto code isn't being sped up *at all* (at least, in Chrome).

That's really disappointing.

One thing that a commenter points out in one of the above Chromium threads is that it might actually be faster to use a non-WebCrypto library (i.e. a synchronous crypto library) with multiple Web Workers, since that would actually be parallel crypto processing. Something to keep in mind.

In the mean time, I've taken these learnings to realize that we could just initialize the keys in the main thread and then instruct the workers to pick them up from storage to speed up the login process (at least, when we have access to IndexedDB). Pretty cool!