Idea: Client Side Encryption

The following document will go over the design details related to providing database level user data encryption, as outlined in Story ticket UFC-XXX.

Motivation

- People don't trust that I won't go into the database and read their financial transactions.
- People don't trust that the database won't get hacked and their financial transactions will get leaked.
- Personally, I don't care about this (i.e. it's a risk I'm willing to take), but if the customer demands it, so be it (on top of it being a cool feature to implement).
- Since the application has already been designed with offline-first in mind, we can offload the data encryption to a background process to maintain the smooth, optimistic user experience.

A thought occurred to me... if we start encrypting everything, then (morally) we won't be able to use LogRocket (or other such debugging/logging tools), at least without marking a whole bunch of fields as 'sensitive'.

Threat Model

This is kind of along the same lines as the Motivation, just in different terms. Implementing client side encryption will address the following parts of our threat model (from the user's perspective):

- Unauthorized database access
    - That is, hackers breaking in and stealing/leaking data.
    - Or even just unstrusted infrastructure providers.
- Compelled disclosure of user data
    - That is, the government asking for data.
- Selling of user data
    - That is, us (uFincs) selling user data for monetary/other purposes.
- TLS is MITM'd/compromised

    - That is, data travelling over the network is no longer safe because of an untrustworthy data transfer network.

- SQL injection
    - Can't do anything with encrypted data if you only get it from SQL injection.
- Authentication (specifically, JWT) obtained from sources other than an XSS attack
    - If attackers can pull information for other users but *don't* have their login credentials (i.e. their password), then all they're getting is encrypted (useless) data.
    - I'm unaware of a path that results in them getting *only* a JWT *without* XSS, but I'm sure it's possible.

Those are the main ones. Note that the following *are not* part of the threat model that is solved by client side encryption:

- XSS attacks
    - If a malicious actor can run scripts on our frontend, they can just read the data right then and there; encryption isn't going to help with that.
- Malicious/untrusted host (i.e. us)
    - If we (uFincs) becomes compromised, there is technically a path where client side encryption doesn't help. We can always modify the encryption code to send off the user's password (unhashed/unencrypted) and use it for nefarious purposes.
    - This can happen as a result of either me turning to the dark, me hiring someone I really shouldn't have, or us (uFincs) being compelled by (e.g.) a government to backdoor our code (this is, yet related, to being compelled to hand over user data, which I would consider a step down from backdooring).
    - Or the more obvious case where my Google account gets hacked and then malicious actors can just change the code served by the servers. Whoever controls the servers ultimately controls the data, regardless of whether there's encryption or not.
        - Obviously, this would require a user to use the tampered code (if a user stored encrypted data and then never used our service again, they'd technically be fine), but that's kind of a given.

Just so we're clear, an XSS attack is probably the worst possible attack a service like ours could face; since all user data is stored, in plaintext, in the browser, then an XSS attack just takes it all. It doesn't even matter if they could take the JWT from local storage; they can just take the damn data!

Design Brainstorming

- So basically how this would work is that the client (i.e. the browser) would encrypt all of the data before sending it over to the backend.
- Since we are offline-first, we can handle all of the data encryption in a separate process that doesn't block the UI.
- We'll probably need to build some way to use Web Workers that can interact with Redux Saga, since the encryption process will be 'computationally expensive' (relatively speaking, at least).
- Having done some research into various JavaScript crypto libraries, I think using the **Web Crypto API** (a browser standard) is the best way forward.
    - Firstly, it provides much better performance because it uses native code (i.e. C) under the hood, as well as hardware acceleration.
    - Secondly, it's a bloody browser standard! It'd be stupid *not* to use it!
    - However, the API does seem a bit wonky, but we'll make do.
    - I think AES-GCM (Galois Counter Mode) is the way to go for encryption, since it seems like GCM is the most efficient mode of operation.
- The **Key Derivation Scheme** linked in the *Prior Art* section above is probably the most sane way to go about handling keys and encryption.
    - Basically, you have a Data Encryption Key (DEK) and a Key Encryption Key (KEK).
    - You use the DEK to encrypt data, and store the DEK in the database *encrypted with* the KEK.
    - The KEK is regenerated on the client-side, which is used to decrypt the DEK, which can then be used to encrypt/decrypt data.
    - This is important because it enables **password changes** to work without having to re-encrypt all of the data.
        - We simply just re-encrypt the DEK with the new KEK (i.e. new password) and all is good.
        - **Password changes** require that the user enters their existing password.
    - As for **password resets**, we have two options:
        a. Users lose all their data if they forget their password (most secure, least convenient).

    b. Users just follow the normal "Forgot Password" scheme of receiving an email and just changing their password (less secure, more convenient).

        – Obviously, this then relies on the user's *email* not getting hacked, but that's probably fine? I mean, the operation we're running here doesn't *necessitate* these levels of security, but they're nice.

        – Hold up, this just isn't possible. How are we supposed to re-encrypt the DKE with a new KEK if we can't decrypt the DKE to begin with?

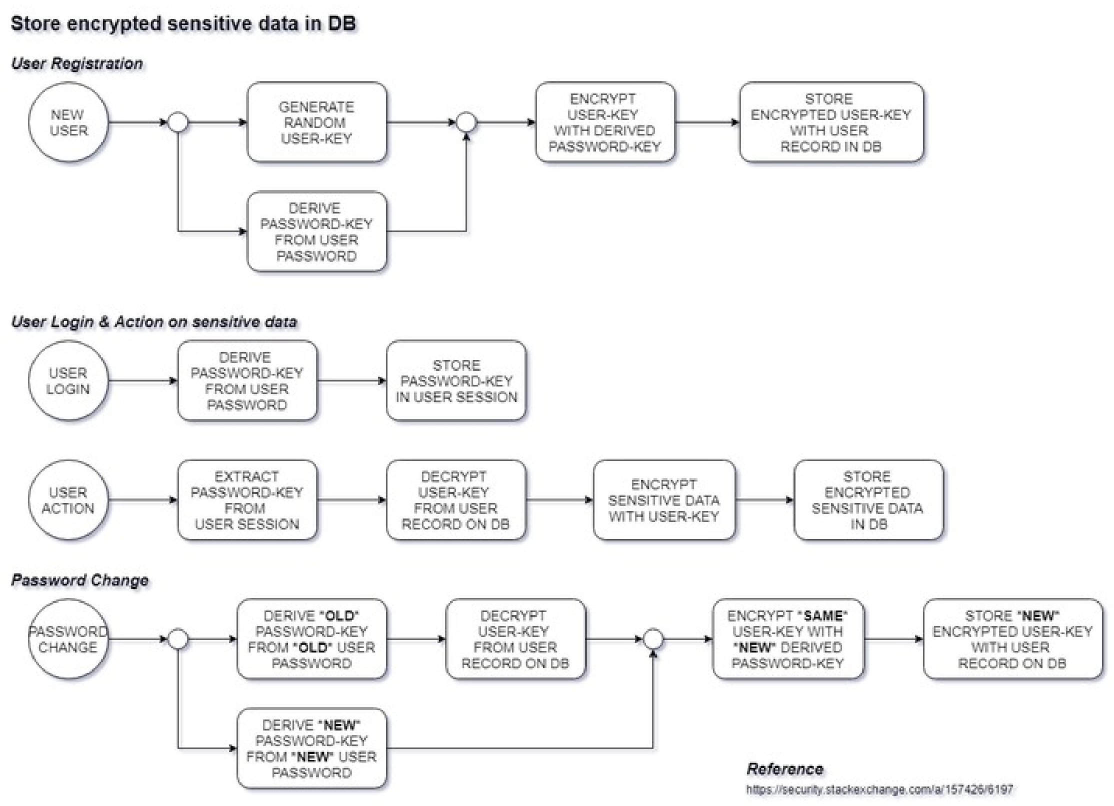
        – Therefore, we *have* to go with **1**.

- As for how to go about actually storing the encrypted data, some thoughts:
    - If we want to re-use the existing database schema that we have, then the easiest way would be to encrypt each field's data *individually* and then just store the encrypted values in the columns in the tables in the database.
        - The fact that we're not obscuring the schema probably doesn't matter considering it could just be derived from the frontend client.
    - The *probably* more secure way (but with way, *way* more work) would be to change to something like a schemaless (cough NoSQL cough) database and just store a dump of encrypted 'json' (or whatever) for each user.
        - But again, *way* more work. I think for our purposes, the field-level encryption should be fine.
    - However, I think the field level encryption might introduce a massive performance penalty (as it should, since this is encryption after all), but like *much* more so than if it was just a blob.
        - Since we have to encrypt/decrypt *every* field, for *every* object, *every* time, the feasibility of this will depend greatly on how fast these operations happen. Cause if we have to do thousands of decryptions when reading all of the user's data, and each decryption is even only a couple of milliseconds (hopefully much less than that), than this could be a big waste of time (both for users, and developers).
- One thing that seems to consistently be brought up about cryptography and web apps is that web apps are fundamentally insecure because a malicious actor could get access to the servers serving the (JavaScript) files and modify them to do any number of things (including siphoning off keys).
    - As such, we're either going to need to accept that this fact and live with it as a risk (most likely), or invent some new way of authenticating files to the browser.

  - For reference, Subresource Integrity seems to be trying to do what we want, I just don't know if it actually accomplishes it.
    - It works by putting a hash on the `script` tags of the JavaScript files in (e.g.) the `index.html`, but theoretically, couldn't a malicious actor that has access to the servers and files just change the files, get a new hash, then change the `index.html` that has the `script` tag and update the hash?
    - That's why I don't think this really solves the problem. Although it does solve *some* problem(s).
  - A thought I had was to inject the necessary file hash from an environment variable into the global vars (like we currently do for other things), but then realized that a malicious actor could also just change the environment variable...
  - Note: While the `CryptoKey` interface has an `extractable` property that prevents the actual key from being read, this seems like a moot point when the malicious JavaScript could just read the user's password that is being used to generate the key and do whatever they want with it.

References

- GitHub - asmcrypto/asmcrypto.js: JavaScript Cryptographic...
- GitHub - diafygi/webcrypto-examples: Web Cryptography API...
- Web Crypto API - Web APIs | MDN
- Comparing Performance of JavaScript Cryptography Libraries
- **Key Derivation Scheme**: encryption - How to encypt sensitive data in database of a...
- Store encrypted user data in database


Store encrypted sensitive data in DB
User Registration
NEW USER
GENERATE RANDOM USER-KEY
DERIVE PASSWORD-KEY FROM USER PASSWORD
ENCRYPT USER-KEY WITH DERIVED PASSWORD-KEY
STORE ENCRYPTED USER-KEY WITH USER RECORD IN DB
User Login & Action on sensitive data
USER LOGIN
DERIVE PASSWORD-KEY FROM USER PASSWORD
STORE PASSWORD-KEY IN USER SESSION
USER ACTION
EXTRACT PASSWORD-KEY FROM USER SESSION
DECRYPT USER-KEY FROM USER RECORD ON DB
ENCRYPT SENSITIVE DATA WITH USER-KEY
STORE ENCRYPTED SENSITIVE DATA IN DB
Password Change
PASSWORD CHANGE
DERIVE "OLD" PASSWORD-KEY FROM "OLD" USER PASSWORD
DECRYPT USER-KEY FROM USER RECORD ON DB
DERIVE "NEW" PASSWORD-KEY FROM "NEW" USER PASSWORD
ENCRYPT "SAME" USER-KEY WITH "NEW" DERIVED PASSWORD-KEY
STORE "NEW" ENCRYPTED USER-KEY WITH USER RECORD ON DB
Reference
https://security.stackexchange.com/a/157426/6197

- How to use Web Worker with create-react-app - Michael Bykovski - Medium
- Speed up your App with Web Workers
- https://stackoverflow.com/questions/13004375/running-window-crypto-getrandomvalues-from-inside-a-web-worker
  - *Can* call Web Crypto API from inside a web worker (see the second comment; the approved one is too old), but since the crypto methods are async, I guess it doesn't really *need* to be run in a worker? Will have to test first to find out.
- Uses the SubtleCrypto interface of the Web Cryptography API...
  - A gist showing how to encrypt/decrypt using the Web Crypto API.
  - Note that it has some problems (particularly with the password hashing).
  - But it *does* show how to convert an ArrayBuffer (Uint8Array) into a base64 encoded string (and points out that TextEncoder/TextDecoder don't mirror each other).
    - According to this: https://stackoverflow.com/questions/50198017/converting-arraybuffer-to-string-then-back-to-arraybuffer-using-textdecoder-text ,

    they don't mirror each other because they're supposed to work with text? i.e. treat each byte as a single character?

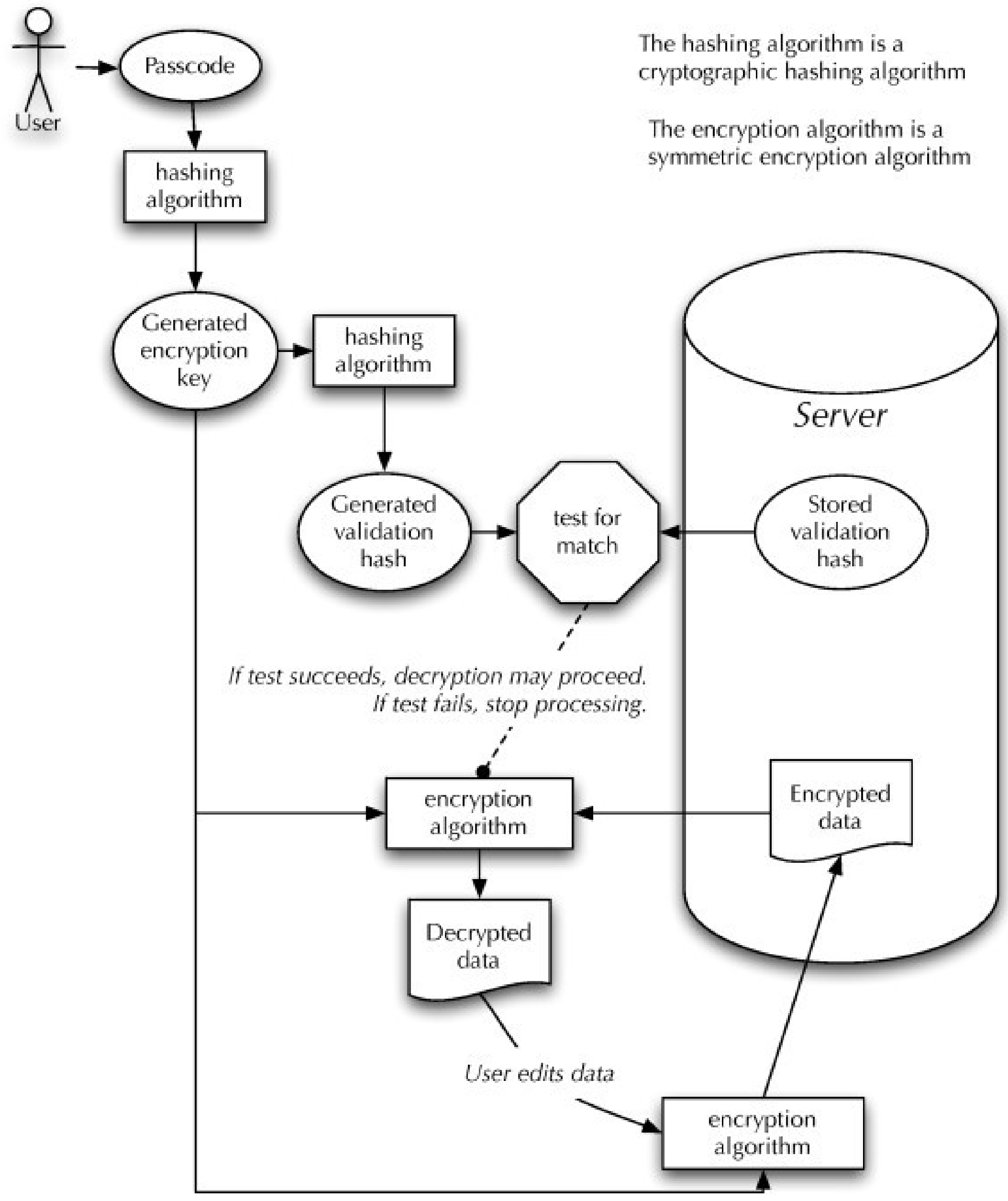
    - Which is basically what the gist is doing?
    - God, this is stupid...

- What's wrong with in-browser cryptography? • Tony Arcieri
  - A great article going over using crypto in the browser.
- SimpleTax: Privacy Policy
  - SimpleTax's old privacy policy
  - Mentions something interesting: they do the encryption on the server, but using the second to last password hash from PBKDF as the key.
  - I wonder if we could do a hybrid scheme where things are encrypted first on the client and then on the server (and if this makes any difference).
    - My first thought is that it wouldn't make any difference in the case where our servers have been taken over (or we are being forced to comply with the government...), because if they control the servers then they could both modify the served JavaScript files and read the password hash from the server.
  - In other notes, this seems like a good privacy policy that we could steal from.
- Encrypted online accounting for freelancers with Sealas (Blog Articles)
  - Oh man, these guys have a treasure trove of blog articles going over ideas for *exactly* what we're trying to do here (offline first, client side encryption, and **an accounting app!!!**)
  - An article on picking a datastore: Offline first, client-side encryption and the quest for a proper datastore
    - tl;dr Postgres and store encrypted data as blobs
  - An article on user authentication: Designing a user system for an encrypted application (Auth Part 0)
    - tl;dr use a data encryption key and a key encryption key
  - An article on their threat model analysis: Sealas Threat Model
    - tl;dr everything they thought of I also thought of, except they didn't acknowledge malicious servers
  - An article on secure cloud computing: Creating a Secure Cloud Infrastructure

- etc, etc
- Like I said, highly coincidentally related to what we're doing.
- However, I couldn't figure out how to use/sign up for their app... it doesn't seem like it's deployed (but the code's on GitHub...).

• Web app with client-side encryption

- Has a really good diagram:


User
Passcode
hashing algorithm
Generated encryption key
hashing algorithm
Generated validation hash
test for match
Stored validation hash
Server
The hashing algorithm is a cryptographic hashing algorithm
The encryption algorithm is a symmetric encryption algorithm
If test succeeds, decryption may proceed. If test fails, stop processing.
encryption algorithm
Encrypted data
Decrypted data
User edits data
encryption algorithm

        - Note: I think this isn't *exactly* what we want. The "test for match" step (where we check the generated encryption key against a known validation hash) is, I believe, just an authentication check (ala a MAC). Since we're using `AES-GCM` (an authenticated encryption scheme), we can skip this step (since it's built in).

- Latacora - Cryptographic Right Answers
- Top 10 Developer Crypto Mistakes
- "OWASP Top 10" for cryptographic mistakes
- An explanation of authenticated encryption with associated data (i.e. AES-GCM): Leaving authentication data blank less secure for AES GCM?
    - Basically, yes, my intuition was correct. The 'associated data' can just be the user's ID. This way, as the answer states, if a user was able to get another user's (encrypted) data *and* secret key into their account, they wouldn't be able to decrypt it because the user IDs are different.
        - Now, obviously, this seems more like of an "accident prevention" feature than an actual prevention measure for a malicious actor. If a malicious actor was able to get a user's encrypted data *and* their secret key (i.e. password), then there's probably little stopping them from also getting their user ID.
- Post on Crypto API sample usage: r/crypto - Is this correct usage of the Crypto API for Javascript?
    - One of the answers here actually brings up the notion of key wrapping algorithms. I was previously unaware these.
    - It seems like they're literally just 'specialized' versions of the regular encryption algorithms that are used specifically for encrypting *keys*.
    - This does appear to be useful for us, since we'll be encrypting the DEK with the KEK; it would appear that using a key wrapping algo (i.e. `AES-KW`) would be 'more' ideal than just plain `AES-256`.
    - Also, it seems like *the* reason to use a key wrapping algo is just because it doesn't require an IV. Since we need to store a salt for the PBKDF2 derivation, it's good that we don't need to *also* store an IV for the DEK encryption.
- More on key wrapping:
    - WebCrypto AES-GCM vs AES-KW for key wrapping
    - This doc page on `wrapKey` seems to have pretty good examples how to do the DEK/KEK derivation: SubtleCrypto.wrapKey()

- This OWASP cheat sheet touches on a number of different cryptographic topics, including key wrapping: Cryptographic Storage - OWASP Cheat Sheet Series
- PBKDF2 and salt
- Use of associated data in `AES-GCM`: Leaving authentication data blank less secure for AES GCM?
- An example of pooling web workers with CRA: gpolyn/react-and-worker-loader
  - A related research term is "thread pooling".
- Storing keys in IndexedDB:
  - Saving Web Crypto Keys using indexedDB
  - https://stackoverflow.com/questions/49478240/where-does-the-webcrypto-api-store-keys
  - Saving Cryptographic Keys in the Browser

Copypasta from Slack Discussions

Blog post on password reset for E2EE apps: https://francoisbest.com/posts/2020/password-reset-for-e2ee-apps

But it's a really pointless solution.

tl;dr create another magic password that the user has to not 'forget' (lose). Which doesn't really improve anything.

--

Post on Crypto API sample usage: https://www.reddit.com/r/crypto/comments/gq1dk5/is_this_correct_usage_of_the_crypto_api_for/

--

I had a shower thought regarding the future roadmap:

I think it makes more sense to do the e2e encrypted stuff before the full blown (CRDT level) offline first stuff. This is, primarily, because having encryption is much more part of uFincs' identity, and it currently doesn't exist at all, whereas we at least have a intermediate implementation of offline first currently.

On top of that, I believe that encryption is a more important selling point for more users than offline first. Especially at the full CRDT/syncing level that I'm shooting for; what we currently have is probably good enough for most users.

Finally, I believe that implementing encryption will be a relatively simpler task (compared to CRDT level offline first), since it'll be primarily implemented as a middleware ontop of our existing infrastructure, whereas the full offline first implementation is a complete rewrite and rearchitecting of our existing infrastructure.

So all that being said, there were a couple other related thoughts I had. One, as part of implementing encryption, I believe we should (or will have to) revamp the app boot process. I believe that the app boot process should be changed such that there is a single saga making all of the 'fetch all' requests, so that we can then issue a single 'app boot' action. This action would contain all of the (processed) server state, and each relevant slice would respond to the action to load up its own piece of state.

This way, we can reduce the number of actions (and thus, renders) that happen during the app boot process to 1 -- everything gets loaded in effectively constant time; no loops.

A nice side effect of this is that we can then implement a nice loading screen for the app boot process. That is, a loading splash screen for the entire app. We can also put 'hip' loading messages here, along with something to indicate that we're decrypting your data. Because... the app boot process will necessarily be the primary place for decrypting the users data from the backend.

However, I think this loading screen should only be used when the user is freshly getting their data from the backend; that is, there is nothing cached client side yet. So once the user has their data, and refreshes the page, they should get right back to the app, but the new data fetches should happen in the background and update the state (more or less) silently once their done.

We *could* go so far as to start incorporating the 'Syncing' indicator that offline first would have introduced. This way we have an indicator for the user for when we're grabbing the latest data and updating the frontend. I would have to think about the verbiage of this indicator, however, since I don't think 'Syncing' would be the right term here.

But yeah, in terms of roadmap, I think once we have CRUD plus import plus a couple of the other smaller UI heavy features implemented, encryption should come before offline first.

And then, before we get to full CRDT level offfline first, we should get to the PWA stuff and make uFincs *really* work 'offline'.

I don't know where the migration of all the Redux stuff to service workers comes in, but it's probably either during or after the CRDT level offline first implementation.

--

I had an idea re: encryption implementation...

I was thinking, could we wrap the Feathers API with a `Proxy` that intercepts all function calls to handle encryption/decryption? Since the API calls already return promises, there's a lot of stuff we could magic away by just proxying the API. For example, we could (I think) even invoke a web worker to perform the encryption/decryption so that it doesn't block the main thread, perform the API call, and then encrypt/decrypt again, before finally resolving the promise to the right value.

It's an interesting idea. I don't think I've really thought *too* much about the exact implementation of how we're going to handle encryption/decryption, but this seems like a viable method.

--

An article on some company's e2e encryption: https://about.misakey.com/cryptography/white-paper.html?pk_campaign=reddit_cedric

Hmm, a great point made in the above article: we can still offer "Forgot my password" functionality, with the caveat that users must be notified that all their data will be lost (since we *can* change their password, we just can't change the password to their *data*).

--

This seems like a good article: Cryptographic Right Answers: https://latacora.micro.blog/2018/04/03/cryptographic-right-answers.html
--

I came across this Stackoverflow post that has a good diagram of the overall encryption scheme: https://stackoverflow.com/questions/15420122/web-app-with-client-side-encryption

--

Can we use Web Workers to parallelize (multi-thread) the decryption process at app boot?

--

In terms of design/architecture, what if the user-facing (i.e. redux) portion of the encryption process was handled by a redux middleware? Any actions that needed to be encrypted would have a `meta` tag saying so, the middleware would intercept these actions, take their payloads and encrypt them, and then re-emit the actions (just without the meta tag). And to be more precise, by "intercept", I mean "take the action and prevent it from reaching the reducers".

To handle which fields of the payload should be encrypted, a schema could be provided at middleware registration to have a map of 'models' to 'fields'. Then, the meta tag for indicating encryption could specify which model this payload corresponds to (e.g. `{meta: {encrypted: "transaction"}}`).

I feel like this kind of design would be very redux-idiomatic and would work well as a packaged library.

Obviously, this is just one portion of the entire encryption/decryption lifecycle, but these are some good thoughts

--

Will need to handle storing the IVs alongside the data, likely just inside the same string with some sort of delimiter (can use `.` or `:`, since they aren't valid base64 chars; for reference, `.` is used in the bcrypt format for our passwords).

Also, will need some way to actually encode/decode the buffers used during the encryption process to base64; some examples: https://gist.github.com/llopv/6257d3eb7024d4889890abe31d5f7f96, https://coolaj86.com/articles/webcrypto-encrypt-and-decrypt-with-aes/ (I don't like this one), https://stackoverflow.com/questions/38677742/cryptokey-arraybuffer-to-base64-and-back