Appendix: Design Requirements Cheat Sheet

This is a non-comprehensive summary of the design requirements. It is not a substitute for reading the spec. All information on this page is also on the spec.

General Requirements

- Atomic Operations
  - Only one function is called at a time.
  - We won't quit the program in the middle of a function call.
  - Malicious action will only happen in between function calls.
- Stateless Design
  - You cannot use global variables (except for basic constants).
  - You cannot store persistent information in the code's local memory.
  - Instead, you must store persistent data on Datastore or Keystore.
- Keystore
  - You can only store public keys (no hashes, no structs, no files, etc.) on Keystore.
- Datastore Adversary
  - They can list out and read all values on Datastore.
  - They can modify Datastore in between function calls.
  - They can take snapshots of Datastore at any time and compare differences between snapshots.
  - They are not a user in the system, and will not collude with other users.
- Error Handling
  - When a function cannot execute correctly, return a non-nil error.
  - You do not need to recover from errors.
  - When malicious action occurs, you can either return a non-nil error, or complete the function correctly (despite the malicious action).
  - After a non-nil error occurs, all subsequent functions have undefined behavior, as long as you do not violate any security requirements.

- Your code should never panic (crash). Always return non-nil errors instead.

Library Functions

- Keystore
    - `KeystoreSet`: Stores a name-value pair.
    - `KeyStoreGet`: Reads a value given a name.
- Datastore
    - `DatastoreSet`: Stores a name-value pair.
    - `DatastoreGet`: Reads a value given a name.
    - `DatastoreDelete`: Deletes a name-value pair.
- UUID
    - `uuid.New`: A random, unique UUID.
    - `uuid.FromBytes`: Converts a 16-byte slice into a UUID.
- JSON
    - `json.Marshal`: Converts a struct into a byte array. Only converts struct fields that are capitalized.
    - `json.Unmarshal`: Converts a marshalled byte array back into a struct.
- Crypto: Miscellaneous
    - `RandomBytes`: Generates the requested number of random bytes.
    - `Hash`: 64-byte hash of arbitrary-length data.
- Crypto: Symmetric-key, confidentiality
    - `SymEnc`: AES-CTR encryption. You provide IV and key. Outputted ciphertext contains IV.
    - `SymDec`: AES-CTR decryption. Panics (avoid this) if the ciphertext is too short to decrypt.
- Crypto: Symmetric-key, integrity
    - `HMACEval`: 64-byte HMAC of arbitrary-length data.
    - `HMACEqual`: Compares two HMACs/hashes for equality.
- Crypto: Public-key, confidentiality
    - `PKEKeyGen`: New, random, public/private key pair.
    - `PKEEnc`: Encrypts with the public key. Cannot support long plaintext (consider a hybrid encryption helper function).
    - `PKEDec`: Decrypts with the private key.
- Crypto: Public-key, integrity

    - `DSKeyGen`: New, random, public/private key pair.
    - `DSSign`: Signs with the private key.
    - `DSVerify`: Checks if signature is valid on message, using the public key.
- Crypto: Key Derivation
    - `HashKDF`: Hashes together source key and purpose. Lets you derive multiple keys from one source key.
    - `PBKDF`: Slow hash on a password and salt. Output can be used as a symmetric key.

Users and User Authentication

- `InitUser`
    - Error if an user with the same username exists.
    - Error if an empty username is provided.
- `GetUser`
    - Error if the user does not exist.
    - Error if the password is incorrect.
- Usernames and Passwords
    - Usernames are unique and case-sensitive, and are any string of 1 or more characters.
    - Passwords are not unique, and are any string of 0 or more characters.
    - Brute-force password attacks are possible when using fast hashes, but not when using PBKDF.
- Multiple devices
    - If the constructor is called multiple times for the same user, multiple User objects exist on different devices. All changes from one device must be reflected on all other devices immediately (i.e. on the next function call).

File Operations

- `User.StoreFile`
    - Stores a new file if the filename doesn’t exist in the namespace.
    - Overwrites the existing file if the filename exists in the namespace.
- `User.LoadFile`
    - Error if filename doesn’t exist in namespace.
- Files

    - Confidentiality is required on file contents, filenames, and length of filenames (and any data that could be used to learn those three values). Confidentiality is not required on anything else.
    - You must detect if a file has been tampered with.
    - Filenames are any string of 0 or more characters.
    - Namespace: Different users could use the same filename, but they could refer to different files.
- `User.AppendToFile`
    - You don't need to check the integrity of the file before appending.
    - Error if filename doesn't exist in namespace.
    - Bandwidth: The total amount of data uploaded with DatastoreGet, and downloaded with DatastoreSet, must be a constant. Bandwidth can only scale with the size of the data being appended. Compute, time, space, etc. does not matter for efficiency, only bandwidth.

Sharing and Revoking

- Sharing and Revoking
    - The user who first creates a file with `StoreFile` is the owner.
    - The owner, and anybody who has accepted an invitation, should be able to access the file (load, store, append, create invitations).
    - If a user changes the file, all users with access must see the changes immediately (i.e. on the next function call).
    - You cannot create copies of the file.
- `User.CreateInvitation`
    - Stores information for the recipient on Datastore and returns the UUID of that information.
    - Error if filename doesn't exist in namespace, or if recipient user doesn't exist.
- `User.AcceptInvitation`
    - The UUID returned from `CreateInvitation` is securely passed to the recipient.
    - The recipient can choose their own filename (possibly different) for the shared file in their own namespace.
    - Error if the chosen filename already exists.
    - Error if the invitation UUID is wrong, or unable to be processed correctly.
    - Error if the invitation UUID is invalidated from a revoke.
    - Undefined if the invitation UUID has already been accepted before.

- `User.RevokeAccess`
  - Revokes access from the target user, and all the users that the target user shared the file with (either directly or indirectly).
  - Only the owner can call this (undefined otherwise).
  - Can only be called on a user the owner directly shared the file with (undefined otherwise).
  - Could be called either before or after the user accepts the invitation.
  - Revoked users should get an error when trying to access the file (e.g. load, store, etc.).
  - Revoked users should be unable to regain access, even if they maliciously use Datastore.
  - Non-revoked users should be able to continue accessing the file without needing to re-accept any invitations.
  - Error if filename doesn't exist, or target user doesn't have access.
- Revoked User Adversary
  - They will only try to regain access to the file. They won't tamper with files they still have access to.
  - They could try to call functions with different arguments.
  - They could directly read/modify Datastore, but cannot list all UUIDs being used.
  - They can write down any values (i.e. local variables your code computed) before getting access revoked.
  - They should be unable to read the latest file, modify the file without being detected, or deduce when future updates are happening.
