CS6903 Project 2 - Web Based Password Manager using Client Side PGP

- Github Repository: https://github.com/joelbcastillo/CS6903-PGP-Password-Manager.git

Description

This project is a web-based password manager that utilizes PGP to encrypt passwords and share them among authorized users. The application uses a frontend component (kbpgp.js) to allow users to upload their private key into the browser storage only and perform client-side operations for decrypting secrets. Allow users to upload their public key to perform client-side operations for encrypting secrets.

This application is similar to existing application such as 1Password, Bitwarden, and LastPass. All of these applications provide a web interface for users to manage their secrets and share them with team members. However, all accounts for these password managers are protected using a password instead of a public-private key pair that is kept under the end users control. It is most similar to the unix `pass` application, which uses GPG and a command-line interface to store and retrieve secrets.

Requirements

- Keyserver - Organizations deploying this solution will need to have a trusted keyserver to verify user public keys for encryption.
    - Default keyserver is https://keys.openpgp.org

Threat Model

- Private Keys Stored in Browser
    - Threat: Users will be uploading their private key into the sessionstorage for the domain. If the key is not cleared out from sessionstorage after the user logs out (or a specified timeout period) an attacker could access the key and decrypt all of the secrets a user has access to.
    - Mitigation: When a user logs out or their session times out, the PGP Private Key will be deleted from local browser sessionstorage as designed.
- Eavesdropping
    - Threat: It is possible for an attacker / 3rd party to perform a man in the middle attack and capture the traffic from the users browser to the backend server. This traffic, when decrypted, could contain plaintext secrets and metadata that are being sent to be stored in the backend.
    - Mitigation: By encrypting the secrets using PGP on the user's browser, we are able to prevent an attacker from obtaining secrets by performing a man in the middle attack. Instead, the attacker would only see a PGP encrypted string when decrypting TLS traffic.
- Public Key Revocation
    - Threat: Users who are no longer supposed to access data in the secret store will be able to access the secrets as long as the secret is encrypted with their public key.
    - Mitigation:

        - We will provide a simple front-end interface for revoking a user's access to one or more secrets
            - Secrets will be re-encrypted without the revoked user's public key.
- Password Hygiene
    - Threat: If a user no longer has access to the secret, but did at some point, they may have an offline copy that would allow them to use the secret.
    - Mitigation: When a user's access to secrets is revoked, a prompt will be displayed to change the secret. Currently optional, as each organization would need to establish a procedure for rotating secrets.
        - If the secret is not changed, the user would still have had access to the secret when their access was valid.
- 2-Factor Authentication Not Implemented
    - Threat: If a user's private key is compromised, the attacker would gain access to all of the user's secrets
    - Mitigation: N/A - Multi Factor / 2nd Factor authentication is currently out of scope for this project.
- Key Hijacking
    - Threat: It is possible to create a key for a user to spoof their access.
    - Mitigation: N/A - Out of scope for this application. Users should only encrypt secrets for other user's whose identities they have verified through another means.
- Data Tampering
    - Threat: An attacker could intercept the data and tamper with the message in transit.
    - Mitigation: An HMAC will be submitted with the data to ensure data integrity.

Topic Related Notes

Data Origination Spoofing

- We require a users private key for authentication to the web interface. Unless an attacker has control of an authorized users private key, they will not be able to spoof data insertions.

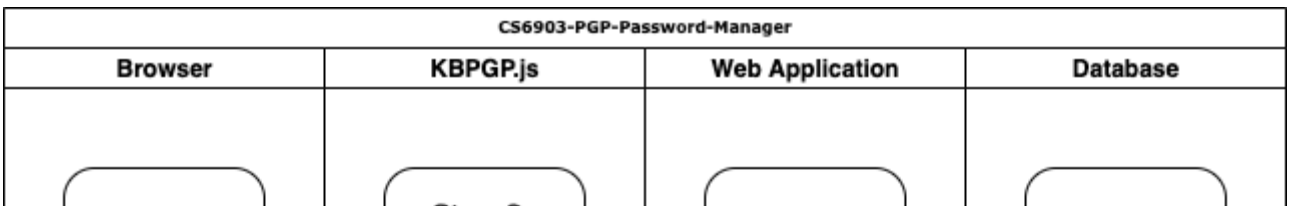
Data Replay

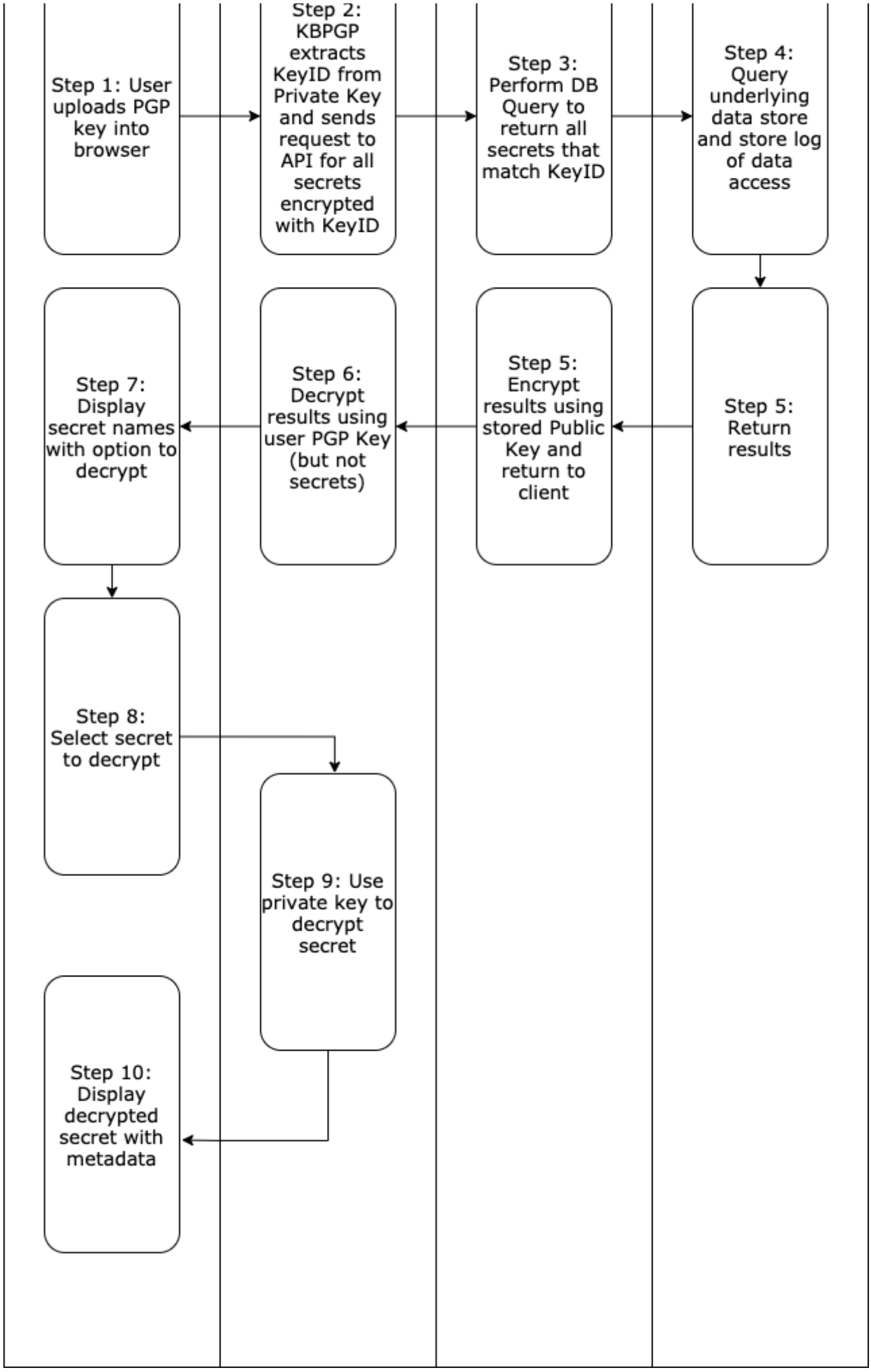
- Data will be transmitted with an HMAC and a timestamp to ensure validity of each API call. The timestamp inside the submitted data package will be checked agains the received timestamp to determine if the message is valid. There will be a grace period of X seconds (where X is most likely < 1 second).

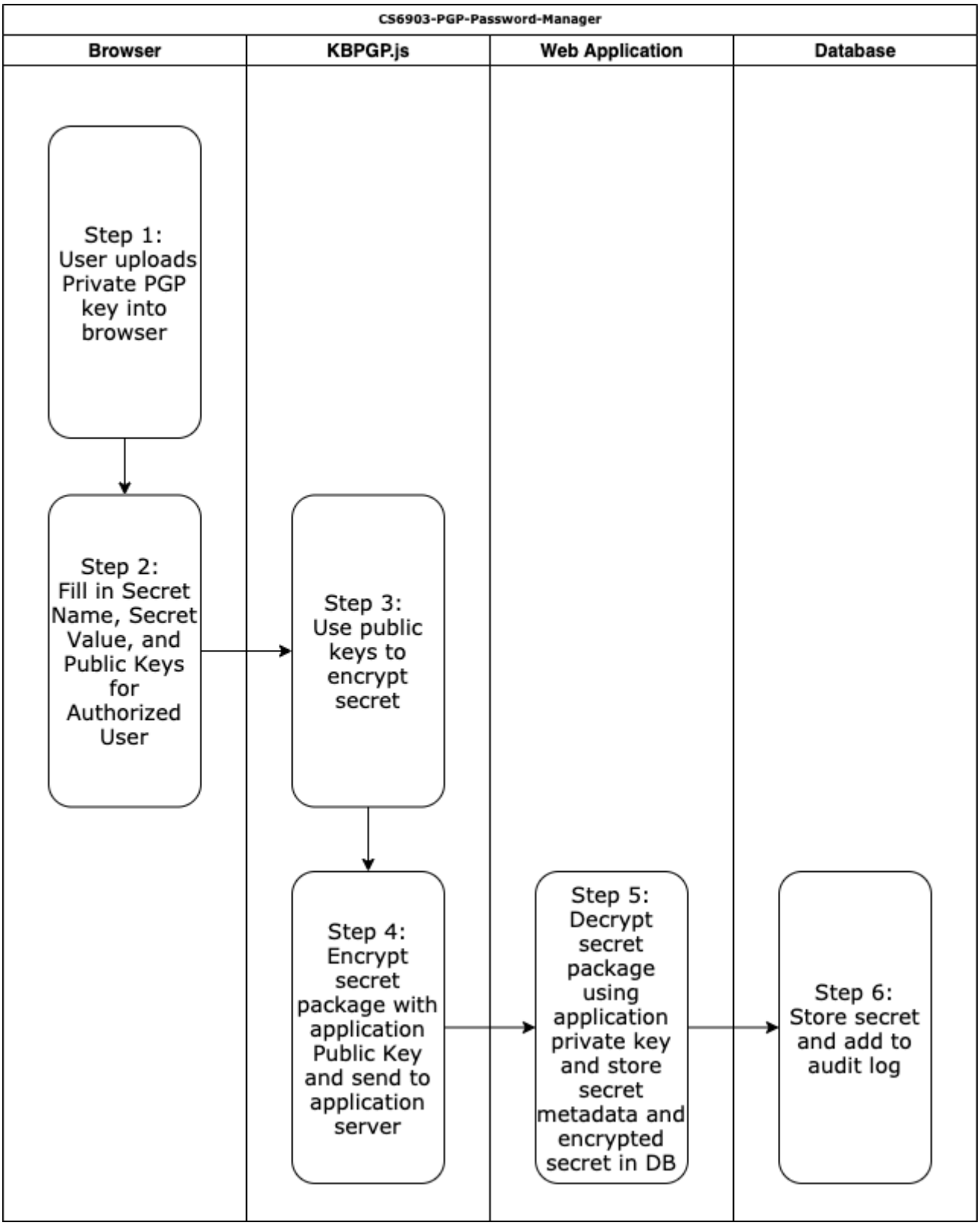
Chosen Plaintext Attacks

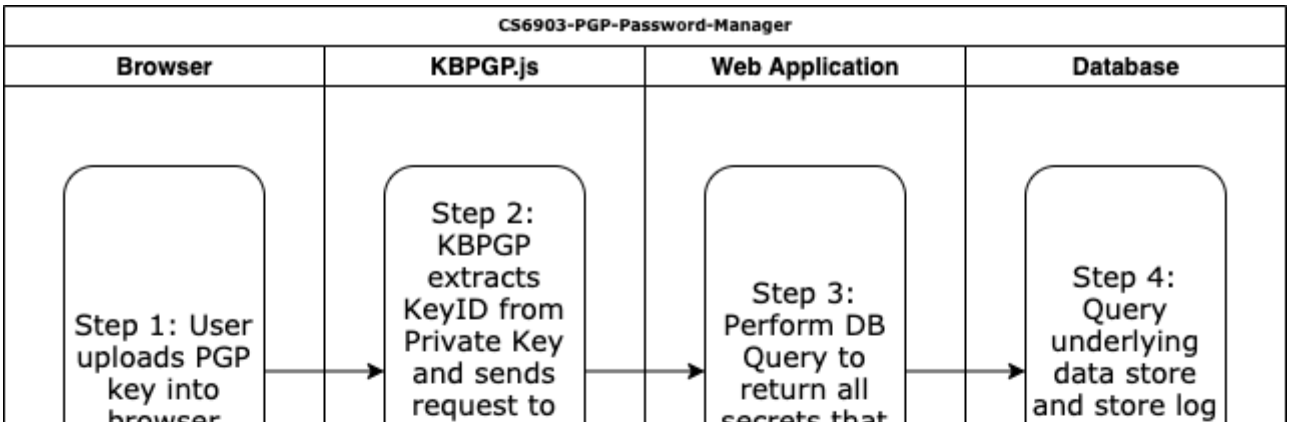
- The output of the GPG encryption algorithm is non-deterministic. As a result, a user would not be able to determine information about the plaintext from the ciphertext

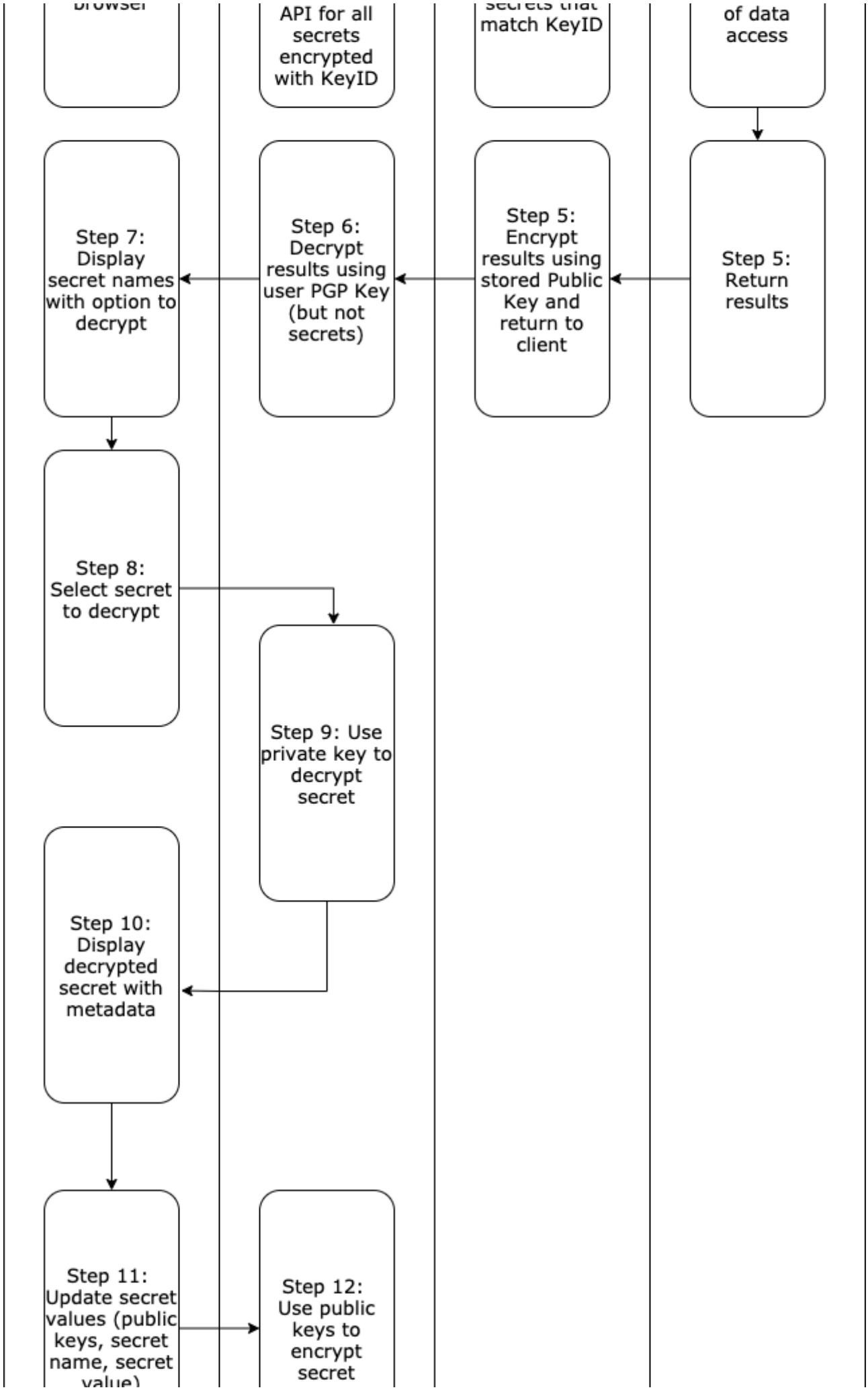
Design

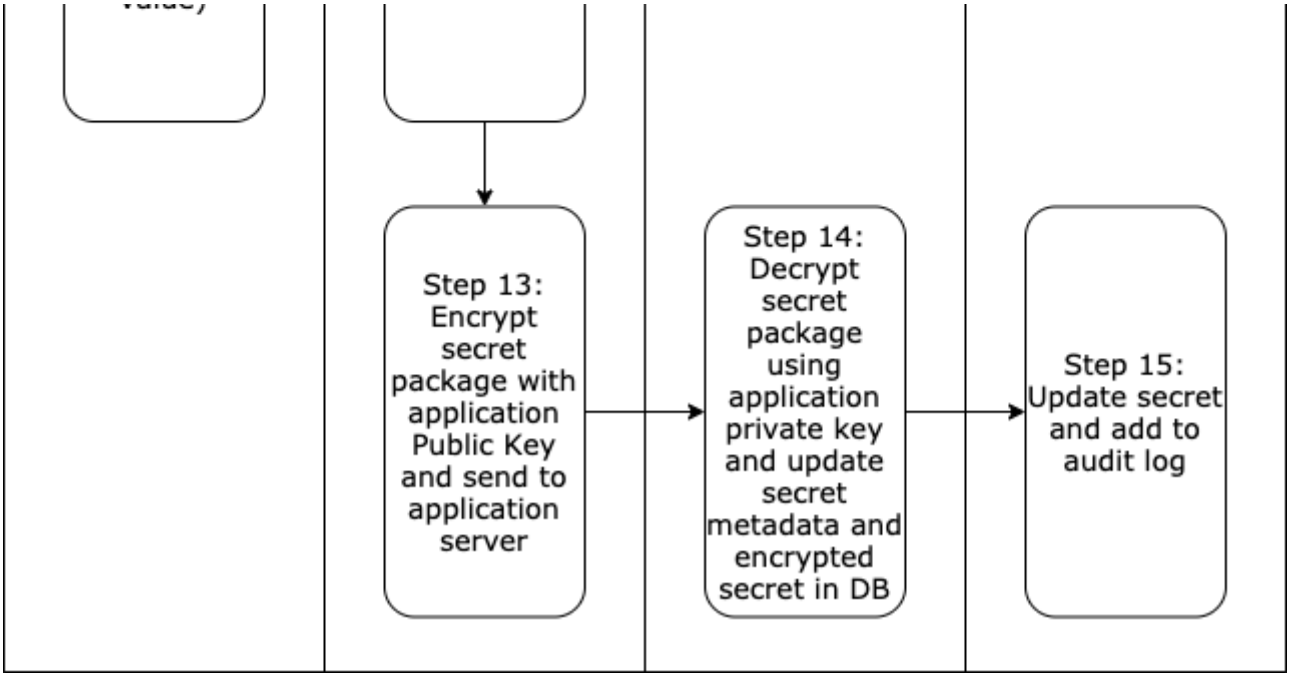



Step 1: User uploads PGP key into browser
Step 2: KBPGP extracts KeyID from Private Key and sends request to API for all secrets encrypted with KeyID
Step 3: Perform DB Query to return all secrets that match KeyID
Step 4: Query underlying data store and store log of data access
Step 5: Return results
Step 5: Encrypt results using stored Public Key and return to client
Step 6: Decrypt results using user PGP Key (but not secrets)
Step 7: Display secret names with option to decrypt
Step 8: Select secret to decrypt
Step 9: Use private key to decrypt secret
Step 10: Display decrypted secret with metadata


CS6903-PGP-Password-Manager
Browser
KBPGP.js
Web Application
Database
Step 1: User uploads Private PGP key into browser
Step 2: Fill in Secret Name, Secret Value, and Public Keys for Authorized User
Step 3: Use public keys to encrypt secret
Step 4: Encrypt secret package with application Public Key and send to application server
Step 5: Decrypt secret package using application private key and store secret metadata and encrypted secret in DB
Step 6: Store secret and add to audit log


CS6903-PGP-Password-Manager
Browser
KBPGP.js
Web Application
Database
Step 1: User uploads PGP key into
Step 2: KBPGP extracts KeyID from Private Key and sends request to
Step 3: Perform DB Query to return all
Step 4: Query underlying data store and store log


browser
API for all secrets encrypted with KeyID
secrets that match KeyID
of data access
Step 7: Display secret names with option to decrypt
Step 6: Decrypt results using user PGP Key (but not secrets)
Step 5: Encrypt results using stored Public Key and return to client
Step 5: Return results
Step 8: Select secret to decrypt
Step 9: Use private key to decrypt secret
Step 10: Display decrypted secret with metadata
Step 11: Update secret values (public keys, secret name, secret value)
Step 12: Use public keys to encrypt secret



Schema

Audit Table

- id - uuid
- timestamp - datetime
- user_id - string
- action_performed - enum
    - Encrypted Secret
    - Decrypted Secret
    - Deleted Secret
- Inputs (JSON)

Secrets Table

- id - uuid
- name - string
- encrypted_value - binary

Users Table

- id - uuid
- key_id - string
- email - string

UsersSecrets Table

- id - uuid
- key_id - string (FK to Users table)
- secret_id - string (FK to secrets table)

API

GET /secret

Args:

- Key ID

Returns:

```
[
  {
    "id": "SECRET_ID",
    "name": "SECRET_NAME",
    "value": "SECRET_VALUE"
  },
]
```

POST /secret

Args:

- Name
- Encryption Key IDs
- Encrypted Value

Returns:

```
{
  "id": "SECRET_ID",
}
```

PUT /secret/<SECRET_ID>

Args:

- Name
- Encryption Key IDs
- Encrypted Value

Returns:

```
{
  "id": "SECRET_ID",
}
```

DELETE /secret/<SECRET_ID>

Args:

Returns:

```
{
  "success": "True"
}
```

Front-End

Functions:

- `load_key(File)`
- `setupKeyManager()`
- `encrypt(key_ids[], secret)`
- `decrypt(privateKey, ciphertext)`
- `modify(secret_id, key_ids[], secret)`
- `delete(secret_id, privateKey)`

Routes

- Login
- Home

Authors

- Ho Yin Kenneth Chan (@kenliya)
- Gary Zhou (@g-zhou)
- Joel Castillo (@joelbcastillo)