Vault: An End-to-End Encrypted File Sharing Service

Justin Stanley

May 11, 2022

1 Problem

Since their inception, file hosting services have become extremely popular among Internet users. People use these services to store important documents, pictures, and other files. These files often need to be shared with other users on the platform. In the modern age of cloud computing, due to data and computation outsourcing much of this data is stored and manipulated on potentially untrustworthy infrastructure. To protect user privacy, sensitive data must be encrypted when it is stored. However, if the encryption keys are stored on the server, a malicious actor can retrieve the keys from the server to decrypt user files. My project moves file encryption to the client side, so all user data is encrypted before it is ever sent to the server. Furthermore, the authentication process also secures the user's password, so a malicious server cannot steal the user's credentials for use on other services. I implemented a file sharing feature, so users can share files with other users on the platform securely, using fingerprint identification to ensure the server cannot intercept shared file data. The project is built from scratch with no reused existing code. I have deployed the project to AWS under an auto-scaling group, so it can scale to a large number of users.

2 Design

The system consists of 3 primary components: **backend**, **frontend** and **storage**.

2.1 Backend

The backend exposes a REST API by which the frontend can interact with the system. The backend also serves the frontend HTML through this API. This component runs on AWS EC2 instances and has access to the S3 secret keys, which it uses directly to upload and download files from S3.

2.2 Frontend

The frontend is an HTML page which users interact with to use the service. A login prompt is initially displayed which the users can login and register with. After a valid authentication token is obtained, the users are directed to a list of their available files. Users can see files which they have uploaded as well as files which have been shared with them. Users can share and download each file available in the list. Finally, users can see their current username and key fingerprint at the bottom of the screen.

2.3 Storage

The service uses S3 to store all user data. At the top level of the bucket is a folder named `users`, which contains a folder for each user designated by their username. Each user folder contains a directory `files` which contains each file owned by or shared to the user. Every user folder contains a file `token` which contains the user's current authorization token and expiration date. Finally, each user folder has a file `info` which contains the user's username, public key and encrypted private key.

3 Implementation

3.1 Backend

The backend is written using Node.js as an asynchronous web application. Express.js[1] is used to serve web requests, and the AWS API is used to make requests to S3. All requests are handled asynchronously using async/await or callback methods.

The REST API is defined as follows:

3.2 Frontend

The frontend is written in Pug which is then compiled to HTML by the backend. The WebCrypto API[5] is used to perform client-side encryption. RSA-OAEP[2] is used for asymmetric encryption, while AES-256[3] is used for symmetric encryption. I heavily referenced the public Mozilla WebCrypto API manual[4] while developing the frontend. The major methods are described below.

3.2.1 Register

To register, a user generates a keypair. A random 16-byte `salt` is generated. The user's password is used along with PBKDF2 to generate a 16-byte `key`. A random 16-byte `iv` is generated. The `iv` and `key` are then used to encrypt the private key with AES-256-CBC[3]. The username, public key, encrypted private key, and `salt`, `iv` are sent to the backend. The server responds with success or failure.

`GET /`	Serves the front-end HTML.
`POST /api/register`	Creates a user account. Expects parameters: `username`, `privateKey`, `publicKey`
`POST /api/preauth`	Starts an auth handshake. Expects parameters: `username` Server responds with `encPrivateKey`, `encNonce`
`POST /api/auth`	Finishes an auth handshake. Expects parameters: `username`, `nonce` Server responds with `username`, `token`, `expiration`
`POST /api/download/<user>/<path>`	Downloads a user file. Expects `token` in request body. Server responds with { `keys, filedata` }
`PUT /api/upload/<user>/<path>`	Uploads a user file. Expects `token` in request body.
`POST /api/key/<user>`	Retrieves a user's public key. Server responds with { `publicKey` }
`POST /api/tree/<user>`	Retrieves a list of a user's files. Expects `token` in body. Server responds with { `publicKey` }

Figure 1: Backend REST API

3.2.2 Login

To login, a client first sends a pre-auth request to the server with their username. The server responds with an encrypted private key, `salt`, `iv`, and an encrypted nonce. The client uses the `salt` and password to generate a `key` as done during register. The `key` is used to decrypt the private key, which is then used to decrypt the nonce. The decrypted nonce is sent back to the server (as proof of ownership of the private key). The server then responds with an authentication token.

3.2.3 Upload

To upload a file, the user generates a random `iv`, `key` and uses them to encrypt the file. The `key` and `iv` are then encrypted with the user's own public key and bundled with the file. The bundle is sent to the server along with an auth token, username and upload path.

3.2.4 Download

To download a file the user simply sends their token along with the desired file path to the server. The server verifies the token is valid and sends the file data. The frontend then retrieves the `key` and `iv` by decrypting the file data header with the user's private key. The keys are then used to decrypt the file, and the user is prompted to save the decrypted file.

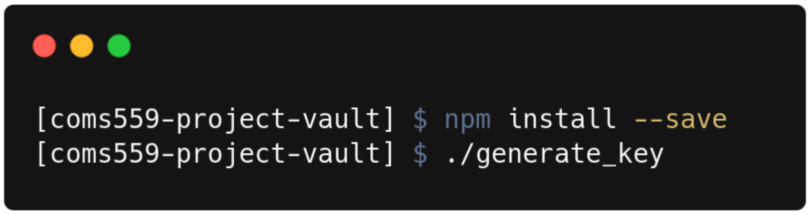
3.2.5 Share

To share a file, the user firsts downloads the file into local memory as in *download*. The user then sends a `get-key` request to retrieve the destination user's public key. The user is shown the destination public key fingerprint to verify it is correct. After confirming, the user re-encrypts the file data as in *upload*, but with the other user's public key. A `share` request is then sent to the server which allows the user to upload files to another's storage. The new file is saved as `shared/<origin_user>/<path>` to avoid overwriting existing files.

4 User guide

4.1 Installation/Requirements

To install and use the application, first ensure your machine has the appropriate AWS configuration so the backend can access S3. On Linux, this file should be located at `$HOME/.aws/credentials`. You must have OpenSSL along with its command line utilities installed. In your S3 dashboard, create a new bucket named `coms559-project-vault`. Proceed to install Node.js by following the instructions for your operating system at `https://nodejs.org`. Clone the project repository from `https://github.com/codeandkey/coms559-project-vault/` if you do not already have it.



```
[coms559-project-vault] $ npm install --save
[coms559-project-vault] $ ./generate_key
```

Figure 2: Installation commands

Run the following commands within the repository folder to download dependencies and initialize SSL keys:

4.2 Starting

After installation is finished, start the backend with the command:
`$ node backend.js`.

4.3 Usage

Once the backend is running, open a web browser supporting the WebCrypto standard (most modern browsers will) and navigate to `http://localhost:8080/`. A prompt will appear allowing you to enter a username and password to register as shown in figure 3. After registering you will be brought to a page with your files and utilities as shown in figure 4. At the bottom of the page you will see your username along with your key fingerprint. You are encouraged to share this fingerprint in advance with users you may share files with. On the left is an "Upload" button which you can click to upload a new file. Once you have uploaded a file you will see the "Download" and "Share" buttons next to the listed file, which you can click to download and share the file with other users respectively. When sharing a file with another user, you will be prompted for their username and then shown their fingerprint. Once you have verified their fingerprint is correct the file will be shared with them.

5 Self-evaluation

I feel most of the major goals of this project were accomplished. All encryption is performed client-side and user files as well as user credentials are protected

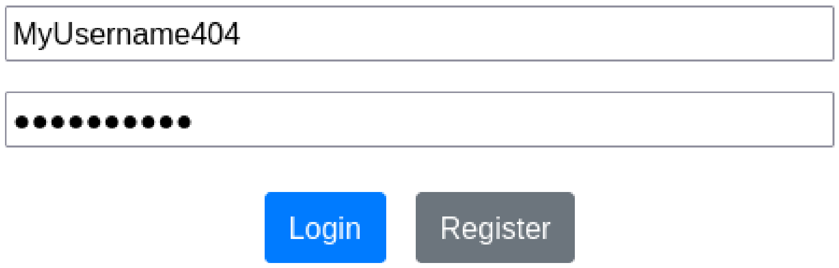
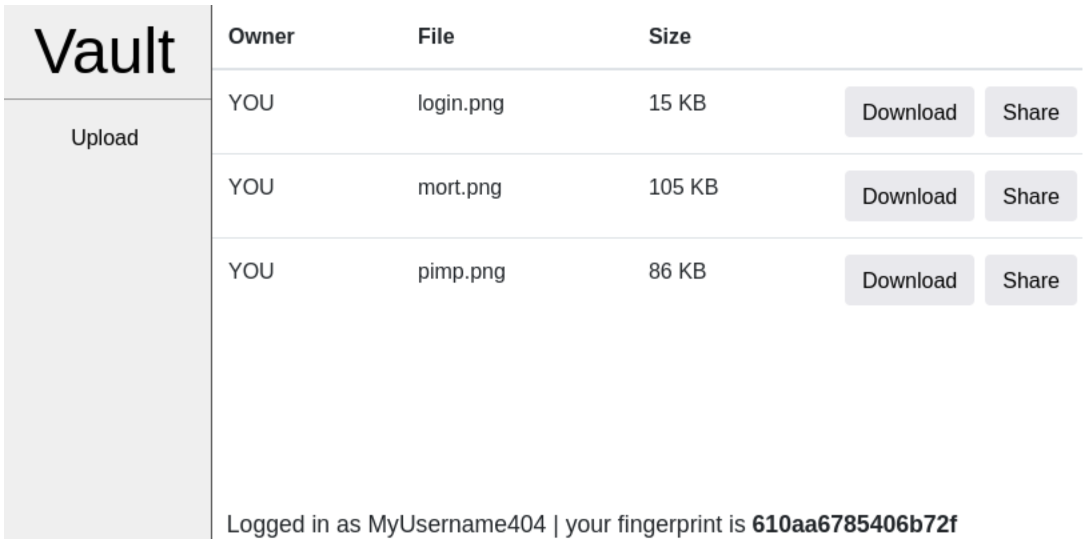


Figure 3: Login page



Owner	File	Size	
YOU	login.png	15 KB	Download Share
YOU	mort.png	105 KB	Download Share
YOU	pimp.png	86 KB	Download Share

Logged in as MyUsername404 | your fingerprint is **610aa6785406b72f**

Figure 4: Main activity page

without any need for server security. However, there are multiple drawbacks to the system as it is:

- The system does not support data streaming. As WebCrypto does not support streaming cryptography, the entire file data must be stored in the browser memory before encrypting or decrypting. Storing large files in RAM is often not acceptable for large files (`> 8GB`). To fix this, it may be necessary to break the file into blocks on the client side, and then encrypt them one at a time before sending them to the server.
- Currently users must share their public key fingerprints to each other outside of the system to guarantee security. I do not see any way around this, and it may be highly inconvienent for users to store each others fingerprints.
- Fingerprints are currently based on the first 8 bytes of the SHA-512 hash of the SPKI-encoded public key. This may not be sufficient- it may be possible for an adversary to easily generate a key with the same fingerprint in a reasonable amount of time, allowing them to impersonate another user.

References

[1] *Fast, unopinionated, minimalist web framework for Node.js.* `https://expressjs.com/`. Accessed: 2022-04-20.

[2] *Optimal Asymmetric Encryption - How to Encrypt with RSA.* `https://cseweb.ucsd.edu/~mihir/papers/oaep.pdf`. Accessed: 2022-04-20.

[3] *The AES-CBC Cipher Algorithm and Its Use with IPsec.* `https://www.ietf.org/rfc/rfc3602.txt`. Accessed: 2022-04-20.

[4] *Web Crypto API.* `https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API`. Accessed: 2022-04-25.

[5] *Web Cryptography API.* `https://www.w3.org/TR/2012/WD-WebCryptoAPI-20120913/`. Accessed: 2022-04-25.