Technical Specification

Problem Statement

Given a full-stack CRUD application. We must guarantee that data inserted into the backend cannot be tampered with without detection, and furthermore, that if tampering has been detected, we must be able to revert the changes and restore the data to its correct state.

To address the concerns, we must clearly state the vulnerabilities present in our current standing application.

Front-End

As this is the major point of communication for our data, we must consider how data is being communicated between the front-end and backend. For this implementation, we will assume that in a production environment, all communications will be completed through some form of OAUTH, and an https connection.

Back-End

The backend is the most critical aspect of this problem. The largest security concern is that an attacker could possibly get access to our production database. This means the attacker could modify or delete our data without hesitation. This solution will predominantly focus on solving the backend security concerns, and allow the front-end to receive validated data.

Scope

The following are definitions of user stories which will be solved given this implementation. Anything not found in this list can be considered out of scope

for this project.

- As a user, I expect to be able to retrieve my stored data on load.
- As a user, I expect to modify, update, and upload new data to the application.
- As a user, I expect to be able to validate that my data has not been tampered with.
- As a user, I expect that my data will be recoverable given that tampering has occurred.

Solution

On reading the problem statement, I had many thoughts for possible solutions. I will outline my two most considered options.

Unique Hash

This solution will generate a unique hash for incoming user data. If at any point, the data in the database changes, the hash that is in the record will NOT match the hashed data. This can be validated at any time by the user, and guarantees no data tampering.

Data Tampering

Consider a user loads the page, the backend will generate a hash for the stored record, and check it against the stored hash. Ex. a mock record.

Hash	Data
e163g178	“Hello World”

If at any point the data is modified, the hash in the record will not match the data. Ex.

Hash	Data
e163g178	“Goodbye Friend”

In this case **“Goodbye Friend”** does not hash to **e163g178.** And therefore we can detect that data has been tampered.

Data Recovery

Given that we have detected a hash mismatch at any point, we know that the record in the database is invalid.

We must then guarantee that there is a backup for all data in the database. There are many solutions for remote backups.

1. **Remote Database**

   This essentially serves as a clone of our database. As soon as new data is added to the database, the backend will duplicate it in a remote database, adding a layer of security.

   Concerns:

   - If the attacker has access to the database, they could also theoretically gain access to the clone. At this point both options are compromised.
   - There is no incremental record of the data in the remote clone other than the log. If the database log is ever compromised, we lost the record of all history for any record.

2. **WORM (Write Once Read Many)**

   This database implementation does not allow replacements of records. Every new record is appended to the end of the list. The most recent entry is considered to be correct. Rollbacks are easy as all variations of the data exist in the database. Data deletions are handled by appending a tag to an old record, for example “Flagged for deletion”, whereby the database can delete the record after a predetermined retention policy has been met.

The following is an example of the WORM database.

Hash	Data	Flag
e163g178	“Goodbye Friend”	Delete
72h1a67s	“Bequest Rules!”	Latest

As you can see, only the latest addition to the record is considered valid. And old records are not able to be modified.

Concerns:

- Database Size: If there are a large number of records, and significant modifications, the records will grow quickly. Although this can be greatly mitigated with a well implemented retention policy.

For our particular problem, the WORM database guarantees a much better record of all the modifications to the data. There cannot be any removals and therefore we have a record of every state of the database. If tampering is detected, the WORM database can retrieve the latest record instantly.

Digital Signatures with Asymmetric Cryptography

The major concern with the hashing solution is that a perpetrator could theoretically generate a matching hash for any data they replace in the record. For example, given an initial database state.

Hash	Data
e163g178	“Hello World”

We will consider the scenario where the attacker wants to modify the record. They could simply generate a new hash for his new data. Ex.

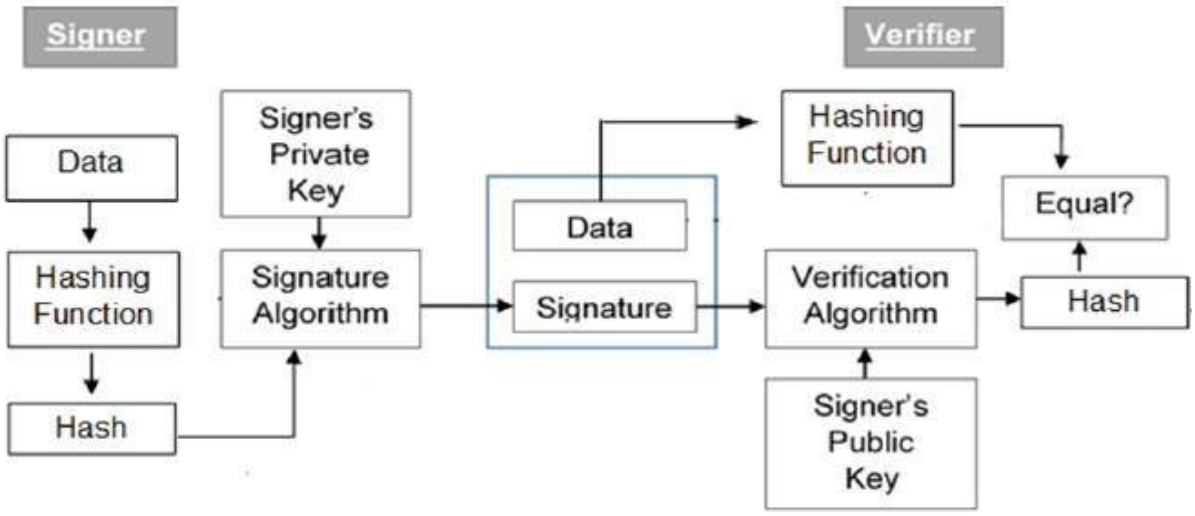
“Hello World” → “Hello Arin” , and a matching hash is generated with the same algorithm as our backend. Therefore the record could be entirely replaced to the following. This is because the hashing algorithm is reproducible.

Hash	Data
ab72uahh	“Hello Arin”

On validation, the backend would have no way of knowing the data has been tampered with unless it references a backup.

This problem led me to considering a technique called Asymmetric Cryptography.

Data Tampering



This diagram outlines a data flow for the solution.

1. We start by generating one **PRIVATE** key which is only accessible by the server. And one **PUBLIC** key which can be freely distributed to any one.
2. When a user submits data, the backend immediately hashes and signs the data with a **PRIVATE** key. This key is stored in a Key Management Service (KMS) which is remote to the server, or in the environment where it cannot be read.

3. All records are stored with both the plaintext data, along with the signature of the data generated by the key.
4. On retrieval, the user gets both the signature AND the data. And can verify the data with it's signature using the **PUBLIC** key.

This solution guarantees that even if the attacker has complete admin access to the database, they CANNOT modify the record as they will be unable to generate a valid signature. This can be verified by any party with the public key. It is critical to note that the private key cannot be accessed by any attacker. The trusted KMS service is fundamental to the signature being secure.

The only case to consider is on data deletion. If data is completely deleted from the database, we still must refer to a backup. This is because, even though the attacker cannot modify the records due to an invalid signature, they can still completely wipe the record. The user would retrieve from the database, and there would be nothing to show. The data recovery portion of the solution is integral to solving deletion issues. This ensures that there is a record to compare to for data mismatch and backup.

Data Recovery

For this solution, we will also use the WORM database solution as stated above for data recovery.

Specifically, when a user retrieves data from the backend, it must be compared to the WORM database, this ensures that tampering AND deletion are correctly accounted for.

1. A user provides data to be stored.
2. The server will sign this data, and store the record along with the signature generated with the **PRIVATE** key.
3. The record gets sent and appended to the WORM database. And subsequently flags the previous record for deletion.

4. At this point, we will assume the attacker has wiped the record from the server. In this case, the signature cannot be validated as there is no record. Therefore, we compare to the WORM database.
5. We see that the latest action (The data deletion) is not validated in the WORM database. Therefore we know there was unauthorized deletion in the backend database.
6. We simply roll back the backend database using the WORM database. Our record is now valid.
7. The user retrieves the data, validates the signature with a **PUBLIC** key, and the flow is complete.

In an optimal flow, the user would never even know the data was tampered. As the record could be compared on retrieval instantly, and the data recovered as needed. Furthermore, one could assign a periodic worker to validate that backend records match WORM database records.

Implementation

Based on the two suggested solutions above, the most complete is the solution which uses digital signatures. I will implement the solution using digital signatures, as well as a WORM datastore for data backups.

For the scope of this project. I will focus on the cryptographic aspects. This means I will implement the frontend and backend without the use of 3rd party services (Such as AWS databases, KMS services, etc..). Those services will be mocked on the server side.

Specifically:

1. The **private** key will be stored in a server side variable called **KMS_PRIVATE_KEY.** This mocks the retrieval of the private key from a key management service.
2. The **public** key will be passed back to the frontend on every retrieval so the frontend can validate the date. This mocks a public API that can provide the public key.

3. The data store will be mocked with a server side variable called **WORM_DATASTORE.** And will fulfill the requirements of a cloud WORM datastore.

Otherwise, all aspects of the application will work as expected. Digital signing and verification will take place, as well as checks for data deletion.

The frontend implements three features.

1. Data Retrieval

   This is handled by a useEffect() which fetches from the backend on page load. The data retrieved contains the **data**, the **signature**, and the **public** key.

2. Data Updating

   This is handled by the update button. When it is pressed, the value in the textbox is sent to the backend. It then expects to receive the latest data from the backend. This is used to get the latest **signature** and **public** key from the backend.

3. Data Verification

   This is handled by the verify button. When it is pressed, the frontend uses the **signature** and **public** key to validate that the data has not been tampered with.

The backend implements the following:

1. Get Route

   This is the core of the security. When a user fetches data, the backend first validates that the signature is correct. If it isn't, we immediately refer to the latest WORM backup. Therefore, the user will never even know if data tampering occurred. The database is updated to the latest backup, and the user retrieves their correct data.

2. Post Route

   This is where the user passes new data to the backend. The data is immediately signed using a private key, and the signature is stored in the

record. Furthermore, the WORM database is appended to so we have a backup of the data.

3. Delete Route

   This is purely for the demonstration, this route demonstrates the capability to fix data tampering and restore the correct state of the database. To test this, simply open up postman or submit a HTTP Delete request through any method of your liking to: ‘localhost:8080/delete’. This deletes the backend record of the data, and you will find that on retrieval, the backend detects the tampering and restores from the WORM storage.