6.858 Final Project - Encrypted File System

Ciara Kamahele, Tim Mickel, Brandon Vasquez

System design

We implemented our file system with Python for both the client and server. Our server stores the files on the backend with a database (sqlite). The client and server communicate using a JSON-based protocol over regular TCP sockets. The server provides remote procedures to store and load data, and the client never communicates secret details (such as filenames or file contents) to the server in an unencrypted way.

Users can register using the client. When registering, the user generates an RSA public/private key pair and sends the public key to the server where it is distributed to the other clients as soon as possible. A home directory is created for the new user as well.

User representation in the database:

`unique user ID`	`username (public)`	`public key (public)`	`updates []`

When a new user registers or a file is shared with a user, the server pushes an update to the user, for example, with the newly registered user's public key. The client downloads all updates sent while they were offline upon login and updates local stores of keys and data.

All communications between the client and server other than registration messages are signed with RSA (using the user's private key) and the connection to the server will be dropped if a client attempts to falsify their identity by altering their local keys. Thus the server can verify that it is actually talking to the user it thinks, as verified by the server's copy of the user's public key.

Metadata and file structure in the database (name and data are encrypted on the client side, signatures are generated on the client side):

`unique ID by server`	`ACL [ … ] by owner`	`ACL sign by owner`	*`name encrypted`*	`name sign` *`by writer`*	*`data encrypted`*	`data sign` *`by writer`*

Directories are implemented as files - a list of file references (by ID) is stored in the data segment of the file.

When a user creates a file, they are prompted to share the file with other users. They can select which users (by name) they want to share files with, if any, and give them read or read/write access to generate the ACL for the shared file. Files that are created, whether they are directories or files that are uploaded to the server, are encrypted using AES: a random AES key is generated for each new file and is encrypted using the public RSA keys of everyone the creator wants to share the file with. The server then pushes these encrypted keys to their targeted users. However, the server never learns the AES keys, so the server is unable to provide users with files encrypted by different keys (those will be decrypted as garbage).

Users who have write access can both write to files or rename them, which pushes the update to all other users who have access to the file. All changes to the files are signed; the new name and data of the file are signed with the writing user's private key and that is first verified by the server. If the user is not on the ACL, the server will reject the changes uploaded by the user. When other users receive the updated files, the client further checks if the signer is included as a writer on the ACL, and if there is some discrepancy, will alert the user and reject the update. This allows the user to notice changes to files by a malicious server (or changes by a malicious user if the server has a vulnerability). Since this protection is extended to directories (directories are just files), the server cannot modify the user's directory by adding a file unknowingly. If the server removes a file (or makes it unreadable), the client is notified that a file it thought it existed (since it appears in their directory) no longer exists, perhaps indicating tampering.

When deleting files, if a user is the owner of the file, the file is removed from the database. If the user is not an owner (i.e., the owner shared it with the user), then the file is just removed from the users directory but the file still persists on the database. As a result, only the user can permanently delete a file from their database.

Assumptions
In order for our system to be secure, we make a few assumptions. We assume that when a user registers for the first time, the public keys of the other users that are distributed to them are correct. We also assume that every time a new user registers, their public key is correctly distributed to all existing users. Every time a client receives another user's public key (whether upon registration or during a general update push), the client checks each key. If the key is for a user they have no record of, they store the key. If the client already has the key, they check it against their stored copy for that user. If the keys do not match, the client alerts the user that there is a problem with the server. Thus, if the server lied about the public key of a newly registered user initially, the server could pretend to be the other user. However, if the server is compromised at a later point, and attempts to push a falsified version of another user's key, the client will notice the discrepancy.

A similar assumption is made for our treatment of ACLs. When a file is created the user is immediately prompted to create an access control list. A file cannot be created unless the ACL is successfully specified. After a file is created, the ACL is immutable. The file owner is recognized as the signer of the ACL, and the ACL is distributed to all users who can read the file. Thus, if the server initially falsified the ACL and ACL signature before distributing it (for example, the server made a fake ACL and signed the ACL itself to pose as the owner of the file), the clients would save the false ACL and use it for reference in the future. However, the server still would not have the secret AES key for the file, so it wouldn't be able to read or change any information in the file. We assume the server sends the correct ACL the first time, and every subsequent time the client pulls the file, the client checks the file's ACL and ACL owner against its stored record and alerts the user if there is a change.

Future expansions

If we continued this project, the first thing that we would add would be peer-to-peer integration so that we would not have to rely on a semi-trusted third party for key distribution. Distributing the ACLs and secret AES keys for files, as well as directly distributing public keys when new users register, would remove reliance on the server to not falsify any of this information.

Another thing we could add to this project would be stronger consistency guarantees, e.g. fork consistency. In the current implementation, a malicious server could serve two users of the same shared file diverging versions of the file. The server can also hide all changes from a single user by not pushing them to the user. We could solve this problem by adding file history into the signatures (like in the SUNDR file system) and by distributing write notifications peer-to-peer instead of through the server.