CS161 Fall 2017 Project 2 Design Document

Morgan Rae Reschenberg

Part 1: System Design

User initialisation
Each user is initialised as a struct which stores their Username, Password, and RSA private key. Their public key is put on the keystore where {username : publickey}. The struct is CFB encrypted with a PBKD key, HMAC'd using a new PBKD key, and saved on the datastore at a hashed value so no information is leaked to the datastore.

Store File
We make a new SharingRecord struct to "share" the file with ourselves. We then also make a new Node struct and save both the node and the sharing record by marshaling and then CFB encrypting the data to a hashed location dependent on filename. The Node file is HMAC'd and the record is RSA-Signed with our private key.

Node Structure
Only one node is created per stored file. For each "append" (including the initial store call), a node has a list of: Reference to append (uniquely random to each append), key to decrypt the append (also uniquely random to each append), and SHA of append contents for integrity check

Sharing Record
Each user will have their own sharing record for each file that is shared with them. This structure keeps track of the shared file's Node. A sharing record has the following fields: location of node, location of node HMAC, and keys to decrypt node, and verify node HMAC (keys are randomly derived). The record is encrypted with CFB + random key (this key is shared with the user via RSAEncrypt + RSASign/Verify)

Loading a File from the Datastore
Given a filename, we verify and decrypt our sharingRecord key and then use that to verify, decrypt, and unmarshal our sharingRecord for the file we're loading. Then, we use the sharingRecord to access the file node and, assuming access hasn't been revoked, we verify and decrypt all appends for the file. The total file content (including appends) is returned.

Modifying/Appending a File
Given a filename, we verify and decrypt our sharingRecord key and then use that to verify, decrypt, and unmarshal our sharingRecord for the file we're modifying. Then, we use the sharingRecord to access the file node and add to our modification lists: a new random key, our append contents, and a verification hash for the append. We save the node and the sharingRecord back to the datastore in the same location from whence they came.

Sharing a File
Given a filename, we verify and decrypt our sharingRecord key and then use that to verify, decrypt, and unmarshal our sharingRecord for the file we're sharing. We make the recipient their own CFB-encrypted

SharingRecord with a new random sharingRecord key. We deliver the key via RSA. We also sign the encrypted value and the sharingRecord with our private key for integrity/authentication.

Receiving a File
First, the recipient verifies the sender's signature on their sharingRecord key using the sender's public key from the Keystore. If the record key is intact, they verify their SharingRecord. If both are intact, they decrypt, unmarshal and attempt to load the file.

Revoking access of a file
Given a filename, we verify and decrypt our sharingRecord key and then use that to verify, decrypt, and unmarshal our SharingRecord for the file we're changing permissions on. We then access the file's node and change both the node encryption key and the node HMAC key; we re-encrypt the node, recalculate the HMAC and update the new keys in our sharingRecord only.

Part 2: Security Analysis

Attack 1: Malicious File Sharing:

Setup: A user receives two sharing notifications for a file of the same name. One share is from the intended sender and the other is from a malicious party intending to spoof the user with a malicious file.

Protections: Because we sign each sharingRecord with the private key of the sender, the recipient is able to distinguish between the malicious sharer and the intended sharer before opening either file.

Attack 2: Modifying file contents on Data Store:

Setup: An attacker tries to modify the file contents of **any** file on the datastore

Protections: If they modify a user stored file, when a user attempts to load a file, a change will be detected when we verify the hashes of each append (including the initial call to store). The hashes are CFB encrypted in our file Node, so an attacker unable to spoof them even though the hash functions themselves are public.

If an attacker modifies a JSON'd Node, we'll detect it during the HMAC verification stage before we unmarshal. If they modify a sharingRecord, we'll detect it when we recompute and verify the signature on the file before decrypting or unmarshalling. Because we sign with our private key, which is unknown to an attacker, the signature cannot be faked.

Attack 3: Attempt to access unshared files

Setup: An attacker attempts to load a file for which they don't have permission.

Protections: If the attacker had previous access to the file but no longer has access, there was an intermediate call to revoke. After a call to revoke, the attacker can still locate the file's node, but can't decrypt it because a call to revoke changes both the encryption and HMAC key for the node. This will result in an error during the HMAC verification phase and the attacker will be returned nil.

If the attacker never had access to a file: they are unable to distinguish that file's node or a sharingRecord for that file from any other file on the datastore. Even if they could, the attacker still couldn't decrypt them. Because all our keys are randomly generated and there is no IV reuse (+ all IV's are randomly generated) any attempt to break encryption is IND-CPA and highly improbable.

Defining Undefined Behaviour

User Initialisation

If at any point the HMAC we re-calculate doesn't match the existing HMAC, we stop trying to perform the indicated operation and return to the user with an error.

Storing a File on the Datastore

If a user tries to create multiple files with the same name, the old file is deleted. All users that had access previously lose access except the owner.

Loading a File from the Datastore

If a user tries to load a file that doesn't exist, the call to load file errors and no file is returned. If the sharing record doesn't exist for this user, we return an error and a nil file reference. If any file append has been modified or corrupted, an error is raised and a nil file reference is returned

Modifying/Appending a File

If a user tries to modify a file that doesn't exist (or one they don't have permission to access), the call to append file errors and no file is returned.

Sharing a File

If a user tries to share a file that doesn't exist (or one they don't have permission to access), the call to share file errors and no file is returned.

Revoking a File

Any user which has access to a file is able to call the revocation method. This results in all users *except the one who called revoke* losing access to the file. This user doesn't have to be the original owner of the file.