CS161 – Spring 2017 — Project 2 Design Doc

Eoin Morgan, SID 24356939, `cs161-adm`
Aditya Iyengar, SID 24816838, `cs161-ajh`

Basic Part II Design

File Types There are 2 types of files stored on the server: "normal" files and "gateway" files (used to share the location of shared files). Every file is uploaded as a `JSON` object containing 4 fields: the value encrypted with a symmetric key (*enc_value*), the IV used in that encryption (IV), a flag for whether the file is a gateway file (*is_gateway*), and a `MAC` computed over the concatenation of the previous 2 fields and the storage location. So if a user uploads a value V to location L, the server stores:

$$AES_{k_e,k_m}(V, I) = E_{k_e}(V)||IV||is\_gateway||\texttt{MAC}_{k_m}(L, IV, is\_gateway, E_{k_e}(V))$$

Here, E_{k_e} represents AES-CBC encryption with AES key k_e and k_m represents our `MAC` key. In the case of a gateway file, the value V will be well-formed `JSON` object containing 4 fields: the location of the file this gateway is pointing to (either a file or another gateway file) (`loc`), the symmetric encryption key (k_e), the symmetric `MAC` key (k_m), and a flag for whether the file at `loc` is a "normal" file (`is_leaf`).

Initialization When the program starts, the user generates two random 32-byte keys ($kpair = k_{dir\_enc}, k_{dir\_mac}$), which the client uploads to '<username>/dir_keys' as the value ($E_{K_{user}}(kpair), SIGN_{K^{-1}_{user}}(kpair)$), where $E_{K_{user}}$ refers to RSA encryption using the user's public key. From this point onwards, the client will always obtain the keys by downloading and verifying them from the server using the user's RSA public/private keypair.

Dictionary Structure This public/private keypair will be used to maintain another file representing the user's file directory. This directory will contain a mapping of file names to file meta-data, with each meta-data record having the following structure:

```
<filename> : {
    'loc' : <file location key on storage server>,
    'k_e' : <sym key used to encrypt the file>,
    'k_m' : <sym key used to compute the MAC on the file>,
    'is_gateway' : <boolean indicates if a gateway file>
    'shared' : [ **Note: empty before a file has been shared for the first time
```

```
        <username>: {
            'loc' : <gateway file location>,
            'k_e': <sym key used to encrypt the gateway file>,
            'k_m': <sym key used to compute the MAC on the gateway file>
        }
    ]
}
```

(Note: When we refer to accessing a user's directory, we mean downloading, verifying the `MAC`*, and decrypting the directory file using the directory keypair (discussed earlier), and possibly re-encrypting and re-*`MAC`*ing the directory and uploading changes.)*

To `upload` a file, the client will add a new entry to the user's directory by randomly generating loc, k_e, and k_m values that associated with the file name and then using those values to encrypt, sign, and then upload it at the generated location on the server.

To `download` a file, the client will look in the user's directory for a directory record with the given name. Then, they will download the file that the record points at. The file's `MAC` is then verified using the corresponding k_m, and decrypted using the corresponding k_e. If that downloaded file is marked as a gateway file, the client will follow its "link" recursively until it finds the original data file or errors out (if the file was tampered with).

To `share` a file, the client will insert a new entry under the 'shared' field of the directory record using randomly generated loc, k_e, and k_m values that are associated with the other user's username. Then, the client uploads the value at *loc* as a gateway file. **This gateway file contains everything the other user needs to locate and decrypt the original file, and is the stored state needed to share files on the server**. Then, the client encrypts the shared file record using the target user's public key, and signs the encrypted message using the user's own private key.

$$message = E_{K_{target\_user}}(gateway\_file\_record)||SIGN_{K_{user}^{-1}}(E_{K_{target\_user}}(gateway\_file\_record))$$

(**Note:** that we can share a file we have received easily by following the same algorithm. The specifications in `download` manage the recursive following of gateway file links.)

To `receive_share` a file from another user, the client first verifies the share message signature using the sender's public key (raising an IntegrityError if it fails). Then, it uses the receiver's private key to decrypt the message, which will contain a file entry for a gateway file. The client adds this record to the user's directory, associated with the provided name.

To `revoke` a user's access to a file, the client finds that file's record in the user's directory. The client (1) pops the user from the `shared` field, (2) updates (AKA reencrypts and re`MAC`s) the file record with a new (randomly generated) location and key pair, and then (3) re-uploads the file using those new parameters. It then iterates through each of the 'shared' records for that file, and for each remaining user (except for the revokee), it uploads gateway

files pointing to the re-uploaded original file along with the new keys needed to access it. **Changing the gateway file changes the state for revocation and ensures that the revokee no longer has access to new updates, thus allowing us to meet all of the revocation requirements**.

Security Analysis

Security Attack #1: MITM When Sharing Access One attack our design would prevent involves a man-in-the-middle attacker modifying the share message after the sender sends the message and before the other person receives it. Such an attack could potentially allow the attacker to have the receiver save the wrong file name, thus invalidating their shared access to the file. However, in our design, we first use a symmetric key to encrypt the filename and then asymmetrically encrypt this symmetric key using the public key of the recipient. We then create an asymmetric signature that involves the encrypted symmetric key, an IV, and the encrypted message and then concatenate this to our encrypted filename to produce our message. Thus, if an MITM attacker were to intercept our message and make any modifications, the receiver would be unable to verify the message signature with their private key and thus would know that it had been tampered with.

Security Attack #2: Accessing A File After Revocation Another attack our design would prevent would involve revokee trying to directly access the latest version of a file they previously had shared access to. Suppose Alice gave Bob shared access and he then wrote down the location of the file along with the keys that were used to encrypt both the message and the MAC; when Alice revokes his access to the file, he could potentially use the location and keys to try to directly access the file on the server. However, our design defends against this by randomly generating new keys and a new location for the file when Alice revokes access from Bob. While these new keys and location are shared with all other individuals who currently have shared access to the file (through a newly-uploaded updated gateway file), Bob will not have access to these new keys and location, and thus will be unable to obtain the newest updates to the file; he will just end up raising an IntegrityError.

Security Attack #3: Decrypting Another User's Shared Messages/Gateway Nodes Because our client sends (potentially many) messages to other users indicating the presence of gateway nodes, an attacker can know that at least a certain number of gateway nodes exist on the file server, pointing to actual shared files. One potential attack would be to decrypt gateway nodes not intended for access by a malicious user. However, this is impossible because each gateway node is encrypted with new private symmetric and MAC keys, known only to the original owner and the shared user. It would be intractable to learn these keys from the encrypted share message (even if the attacker could view many of such messages), because each message is itself encrypted with another key for confidentiality. This message key is encrypted asymmetrically using the recipient's public key, which will not leak any information. Thus, the recipient's public key acts as a trust anchor that allows us to set

up a secure gateway file that an attacker cannot identify or read (because it looks like any other encrypted file on the server).