Dropbox

Group Members

- Nathan Andrews (nandrew4) [CS1620 Student]
- Siming Feng (sfeng22) [CS1620 Student]

System Overview

Users, User Creation, and User Authentication

What Makes a User?

All users have the following items:

- A *username*: a unique identifier for the user
- A *password*: a secret string used to authenticate a user
- A _`file_key_generator_key`_: a key used to generate *file keys*, which are in turn used to encrypt the content of a file
- A *public/private key pair* used to encrypt file keys in the user's `owned_files` map
- An _`owned_files_key`_ which is used to encrypt the `owned_files` map
- An _`owned_files`_ map which maps the hash of a filename to the hash of the username of the owner of the file. In the case of a file that was shared, the owner of the file will be the hash of the sender's username.s
- A `shared_files` map which maps the hash of a filename to a secondary map. The secondary map maps the hash of the recipient's username to a the file key for that file. The file key itself is encrypted using the recipient's public key, and thus only the recipient, with their private key, can decrypt the file key.

User Creation

When `create_user` is called, we first create a *password key* using `crypto.PasswordKDF` with the raw password. Using the password key, we make the `file_key_generator_key` and the `owned_files_key` using `crypto.HashKDF` with the password key.

We then generate the public/private key pair, and store empty maps (dictionaries) in the dataserver which are the `owned_files` and the `shared_files`.

We then create a dictionary which represents the user object. This object contains the `file_key_generator_key`, the private key, the `owned_files_key`, and the memlocs of the `owned_files` and `shared_files`. we then serialize this dictionary and encrypt the dictionary using the password key. We then store the dictionary in the dataserver at the memloc created by using the first 16 bytes of the hash of the username.

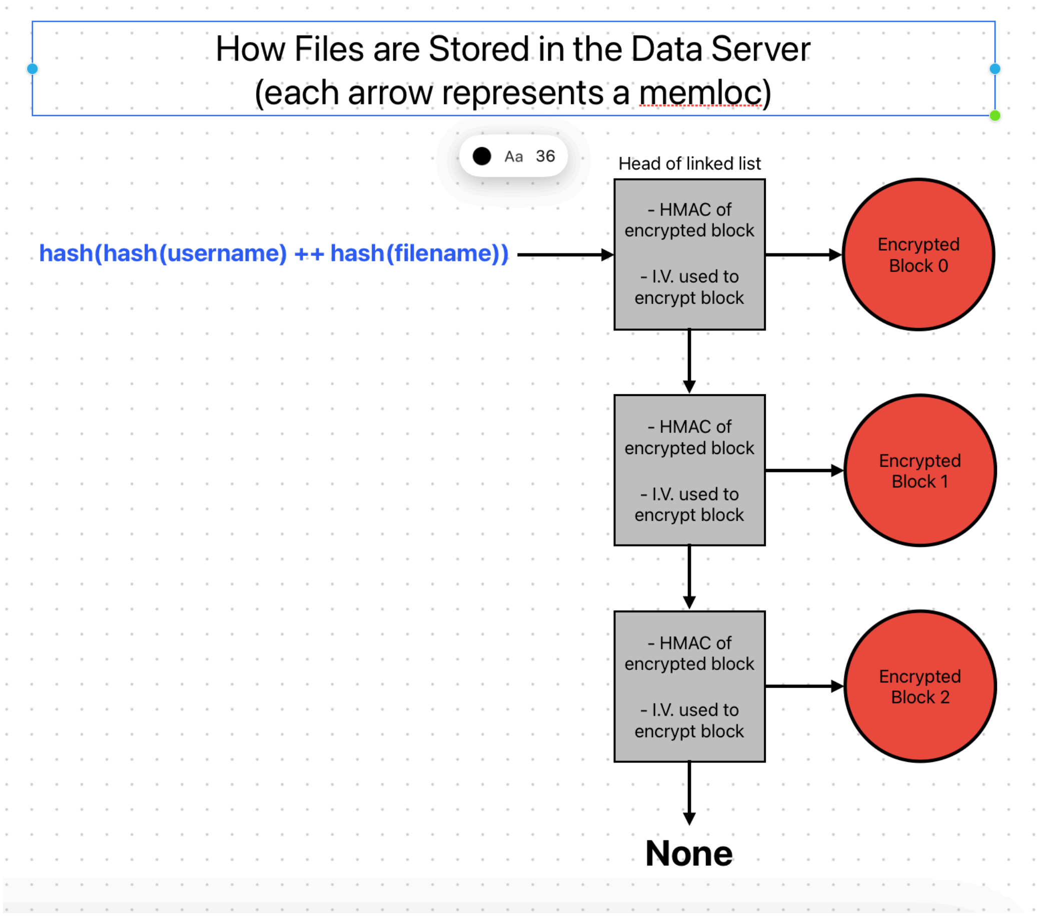
It should be noted that when `create_user` is called, before anything else, we first check if the user exists. To do this, we compute the memloc mentioned before using the hash of the username and check if there exists an entry in the dataserver at this memory location. If so, then we error.

User Authentication
To authenticate a user, first we generate the password key exactly as in `create_user` with `crypto.PasswordKDF` with the raw password. We then get the serialized user dictionary in the dataserver and error if this data does not exist in the dataserver. We then attempt to decrypt the encrypted user dictionary using the password key. If the decryption is successful, then we know that password was correct and the caller is authenticated. At this point we use the items stored in the user dictionary to populate a `User` object and return this object to the caller of `authenticate_user`.

Files

File Representation

File Diagram


How Files are Stored in the Data Server
(each arrow represents a memloc)
Aa 36
Head of linked list
hash(hash(username) ++ hash(filename))
- HMAC of encrypted block
- I.V. used to encrypt block
Encrypted Block 0
- HMAC of encrypted block
- I.V. used to encrypt block
Encrypted Block 1
- HMAC of encrypted block
- I.V. used to encrypt block
Encrypted Block 2
None

file diagram

In our Dropbox implementation, files are represented as a linked list where each node in the linked contains the following elements:

- The memory location of an *encrypted* chunk of files content
- The MAC of the encrypted file content
- The memory location of the next node in the linked list (or None)

It should be noted that there are other fields stored in each node (such as a the memory location of the node itself), however these fields are not important when describing the security of our design.

As listed above, each node in the linked list contains a memory location to an encrypted chunk of file content.

Each chunk of the file is encrypted using symmetric encryption using a random IV and a `file_key` which is generated from the owners `file_key_generator_key`, the file name, and the users the file is shared to. Each chunk is encrypted using the same file key but a different IV.

Each node also stores a MAC of the encrypted file content, as well as the IV that was used to encrypt the content. The MAC is used to verify if either the content, or the MAC itself, were corrupted between the time of creation and the time of access. We also store the IV that was used to encrypt the chunk. Storing the iv is necessary in other functions such as `download_file`.

Checking if a File Exists

The memory location of the head of the file's linked list can be determined if both the username of the owner of the file `owner_name` and the name of the file `filename` are known. The memory location of the head of the linked list can be found by hashing `owner_name` and `filename` separately, concatenating the two resultant hashes, hashing that result, and getting the first 16 bytes of the new hash. The result of calling `memloc.MakeFromBytes` on these 16 bytes will result in the memory location of the head of the linked list of the file named `filename` owned by user `owner_name`.

To determine if a file exists, we query the dataserver. If there is an entry for the memory location of the head, then the file exists, else the file does not exist.

Uploading a New File

If `upload_file` is called and the specified file does not already exist, we do the following:

Divide the file into block-sized chunks. For each chunk `ch`:

1. Generate an random `iv`
2. Encrypt `ch` using symmetric encryption with `iv` to get the ciphertext `c`
3. Create the HMAC `h` of `c` using `crypto.HMAC`
4. Generate a random memory location `memloc_chunk` using `memloc.Make`

5. Generate a random memory location `memloc_node` using `memloc.Make` NOTE: if the chunk is the first chunk of the file, create `memloc_node` using
6. Store `c` in the dataserver as location `memloc_chunk`. `memloc.MakeFromBytes` with the input `file_start_hash`
7. Create a the node for the file (made using a dictionary)
8. Store `memloc_chunk`
9. Store `h`
10. Store the memory location of the next node as None
11. Set the memory location of the previous node's (if it exists) next node as `memloc_node`
12. Store `iv`
13. Store the node in the dataserver at location `memloc_node`
14. Store the hash of the filename in the user's `owned_files` map and update the `owned_files` map stored in the dataserver

Re-Uploading an Existing File (CS1620 Requirement)

If `upload_file` is called and the specified file does exist, we need to ensure that we only upload data proportional to the bytes are different between the new file content and the old file content. We do the following:

1. Get the number of block-sized chunks that make up the new data
2. Using the method described in Checking if a File Exists, get the memory location of the head node of the linked list, `cur_memloc`.
3. If `cur_memloc` is None, and there is still new file content to write to the data server, append the new file content to the end of the link list. If `cur_memloc` is not None, get the `node` stored at `cur_memloc`.
4. Create the HMAC of the current chunk of new file content:
5. Get the IV stored in `node`
6. Encrypt the chunk using symmetric encryption to get the ciphertext `c'`
7. Create the HMAC `h'` of `c'` using `crypto.HMAC`
8. Compare `h'` to `h` retrieved from the dataserver. if `h'` == `h`, then we assume that `c'` == `c` and we do not need to upload `c'`. Move to the next node in the linked list and go to step 3. If there are no more nodes in the list, stop.
9. If `h'` != `h`, then we assume that `c'` != `c` and we need replace `c` with `c'`:
10. Replace `h` with `h'`
11. Replace the data stored at `memloc_chunk` with `c'`
12. If we have reached the end of the new file content, set the next field in the current node to None.

13. Move to the next node in the linked list and go to step 2. If we have reached the end of the linked list, stop.

In essence, to achieve efficient upload, we only upload new data if the HMAC of the encrypted data *already stored* on the dataserver does not match the does not match the HMAC of the encrypted corresponding bytes of the new file content.

Appending to a File

In the case where we need to append to a file, we first get the memory location of the head of the linked list as described in Checking if a File Exists, and then get the stored node. We then iterate through the linked list until we reach the end, and then we use a similar process to uploading a new file, except we set the next field of the last node in the linked list to point to the first node of the file content to append.

Implementation-wise, we have a helper function `append_to_node` which will create a linked list of file content and then append that linked list to the node given by a memory location.

Sharing a File

To share a file with name `filename` with a user with username `recipient`, we first check that file `filename` exists in the dataserver and that the caller of `share_file` owns the file. We then check if `recipient` is a real user in the Dropbox system. We error in the following cases:

- `filename` does not exist
- `filename` does exists but the caller does not own the file
- `recipient` does not exist

Next, we generate a new file_key for the file using `crypto.HashKDF` with the caller's `file_key_generator_key`, and the purpose being the hash of the filename concatenated with the hash of every user the file is shared to (including the newest user). We then encrypt the new file key with every `recipient`'s public key (stored in the keyserver), and update every `recipient`'s entry in the caller's `shared_files`. We then write the updated `shared_files` back to the dataserver.

Also, we save the file data at the beginning of the operation and reupload the data using the new file key at the end.

Receiving a File

Assume that we have just shared a file using the method described in the section Sharing a File. This assumption implies that there exists an entry in the owner of the file's `shared_files` map that contains the file key for `file_name` encrypted with the caller's public key.

To receive a file, we get the `shared_files` of the sender of the file (which is stored in the `owned_files` map), and then find the entry corresponding the the caller for the file `filename`. If not such entry exists, we error. If such an entry does exist, attempt to decrypt the file key with our private key. If the decryption is successful, then we know that we can download the file successfully.

It should be noted that `receive_file` only checks that the caller has access to the specified file, if it exists.

Downloading a File

Any user can down load the encrypted content of a file if they know the username of the owner of the file and the filename of the file. However, to decrypt the file contents, the caller needs the file key for the file. To get the file key, we first check our `owned_files` map to get the owner of `file`. If we own the file, then we generate the file key using the hash of our username and the hash of `filename`.

If we do not own the file, then we go to the `shared_files` of the owner of the file and check if we have an entry in the map. If we have an entry, meaning that there exists the file key encrypted with our public key, then we get the encrypted file key and decrypt it with our private key.

Once we have the file key, we get the head of the linked list for `filename`, and we begin to download the file. For each node in the linked list, we do the following:

1. Compare the stored MAC in the node to the recomputed MAC using the file key and the encrypted data. If the HMACs do not match, then we assume that there is data corruption and we raise an error to the caller
2. Decrypt the node's file content using the file key and append the content to the end of our accumulated file content
3. Move to the next node, or stop if the next node is None

Once we have downloaded, decrypted, and appended all of the file content, we return the content to the user.

Revoking a File
To revoke a file, the owner of the file removes the entry from their `shared_files` that corresponds to the old recipient. The owner then generates a new file key using `crypto.HashKDF` with the owners `file_key_generator_key`, and the purpose being the hash of the filename concatenated with the hash of every user the file is shared to (without the removed user). We then encrypt the new file key with every `recipient`'s public key (stored in the keyserver), and update every `recipient`s entry in the caller's `shared_files`. We then write the updated `shared_files` back to the dataserver.

Also, we save the file data at the beginning of the operation and reupload the data using the new file key at the end.

Security Analysis

Disclaimer
In our actual design, raw filenames are never stored on the dataserver. However, for the sake of explaining the attacks, we will at times assume that the attacker has gained access to the name of a file and, at times, the owner of that file.

Attack 1: Overwriting File Data
In this scenario, the attacker knows that user A owns a file. The attacker gets the head of the linked list for that file and tries to forge new file content by replacing the encrypted file content with their own encrypted file content and the MAC stored in the node with the MAC of the forged file content.

When user A tries to download the file, they will recompute the MAC of the encrypted file content using the file key which the attack does not have access to. The attacker cannot access the file key because the file key is derived from the `file_key_generator_key` which, which the attacker also does not know and thus cannot reconstruct the file key. Because the attacker cannot reconstruct the file key, the attacker's HMAC will not match the recomputed HMAC and user A will therefore be able to determine that the file has been corrupted.

Attack 2: Sharing an Un-Owned Files
In this scenario, the attacker knows that user A and user B exists and that user A owns a file named `aFile`. The attacker tries to share `aFile` with B even through user A has not shared the file.To do this, the attacker would need go to user A's `shared_files` map and add the entry <`hash(aFile)`, <`hash(user B username)`, `EncPubB(file_key_aFile)`>>.

However, the attacker cannot reconstruct the file key because the file key is generated using A's `file_key_generator_key` which the attacker does not know. The attacker could try to store is own fake `fake_key` in place of the file key for `aFile`. In this case, when User A B calls `receive_file` on `aFile`, `receive_file` would try to decrypt only the first block in `aFile` with the `fake_key`. However, since the `fake_key` cannot be the real file key, the decryption will fail and `receive_file` will raise an error indicating that the decrypted file key is fraudulent.

Attack 3: Accessing/Modifying the Stored User Data
In this scenario, the attacker knows the username of user A. In our implementation, each user has some of their private data (such as private keys) stored on the dataserver in a dictionary. This dictionary is stored at the memory location obtained using the first 16 bytes of the hash of the username. Since the attacker knows user A's username, the attacker knows where this sensitive data is stored.

However, this data is encrypted with the password key generated by user A's raw password, which the attacker does not know. To decrypt the data, the attacker would need to try all possible permutations of bytes that equal the key length. In the case of our implementation the key length is 16 bytes, which means that there are 256^{16} possible combinations of password keys, which is an infeasible number of keys to crack.

In the case where the attacker simply overwrites user A's data, when user A tries to authenticate with the correct password, the decryption and deserialization process of the user object will fail, in which case `authenticate_user` will raise an error to the user.

It should be noted that, unfortunately, in our implementation when such an error occurs we are unable to determine if the error is the result of an incorrect password or the result of an attacker corrupting the bytes in the dataserver.

Attack 4: Appending to an Existing File
In this scenario, the attacker knows the username of user A's and that user A owns a file with the name `aFile`. Since the attacker knows the username and filename, the attacker can get the first node in the linked list of `aFile`. The attacker then traverses the linked list until the end of the file. And then appends a new block to the linked list, essentially adding to the file content.

Our design protects against this attack because when user A calls `download_file` on `aFile`, user A will attempt to decrypt the file using the file key, which the attacker does not know. This means that the data the attacker appending to the file (by creating a new node and a corresponding chunk of data), is either unencrypted, or encrypted with the incorrect key. In either case, the decryption will fail, which means that the file content of the new appended chunk has either been tampered with or is fraudulent. In either case, we raise an error to user A, meaning that we can detect that the attacker tampered with `aFile`.