# 162 Dropbox Design

Kasemsan and Daven

April 24th, 2017

## 1 High Level Goals

In terms of high level security goals, there are a few things we want to make sure:

1. Protect the integrity of users' files (think of this an ensuring write protection). Concretely, we want to prevent other users or server side errors from causing files to be corrupted or lost.
2. Protect the privacy of users' files (i.e. read protection). In addition to protecting against malicious users gaining unrestricted access to files, we want to make sure there are no loopholes in our server side logic whereby a user is served up another user's file.
3. Provide adequate authentication security, i.e. make it difficult for an attacker to log in as another user, steal their session ID, etc.
4. Keep track of chains of permissions.
    - For example, let's say file A is owned by user A, who then shares read permissions with user B. In this case, you wouldn't necessarily want user B to be able to share file A with user C, or worse, give read/write permissions to a different user.
5. Related to the above, we want to balance computational costs with security.
    - Similar to how services use RSA to set up symmetric keys since symmetric key encryption is significantly faster than RSA (even though, in some cases, RSA might be more secure).

## 2 Authentication

Authentication of users will be handled via usernames and passwords. Usernames will be required to be alphanumeric strings no greater than 40 characters, but no shorter than 15 characters.

## 2.1 users.db

Information pertinent to user verification will be stored in a SQL database for "users". Employing good MVC structure will ensure the persistance of data even when the server isn't running. The 'user' table will contain six columns:

- username
- hash of password
- per-user salt
- session ID
  - when a user logs in, they'll be assigned a random, encrypted session ID, which is what they'll pass in future API calls to verify their identity. This ID will resolve to a username, which is how we'll index into other databases, such as those holding files.

    It's necessary to use the username as an index into tables, instead of the session ID, since the session ID will constantly change (due to logging in/out, timeout, etc). It would be too costly to have to update session IDs across multiple tables.
- timeout
  - create a timestamp when user's session ID is created, then check against this during API calls. After a certain amount of time has elapsed, update session ID.
- pwd
  - for storing current working directory in order to support pwd and cd commands provided in the stencil code

## 2.2 vulnerabilities

As is good practice whenever accepting user input, we'll perform input sanitization via prepared statements to protect against SQL Injection attacks. All passwords will be salted and hashed with per-user, randomized salts using either Go's standard password package scrypt or one of the encryption algorithms (SHA256, MD5, etc.) from crypto. Adding per-user salts will help protect against rainbow table attacks.

A key consideration, which came up in the flag project, was the vulnerability of passing cookies back and forth between client and server; we saw how sniffing traffic could expose cookies passed in plaintext to attackers. To prevent this, we'll encrypt the session ID and limit its passage as an implicit part of API calls. What we mean is that a user will never be expected to know or pass their

session ID as a parameter in an API call to the server. As such, an attacker wishing to forge someone else's session ID would have to **1:** determine the user's session ID in the user table before it expired, **2:** decrypt it and **3:** edit the client binary to pass this particular value to the server.

# 3 File Storage & Access Control

We'll store information about files and the tree structure of the file system in a single table: 'fd', while all uploads will be placed in a read, write, and execute protected folder '/uploads' (i.e. a folder more protected than the '/tmp' folder from the 'handin' project).

## 3.1 fd.db

The 'fd' table will be fairly substantial with eight columns. We might end up breaking this table into two separate tables: 1 for files and 1 for directories. While this might make searching the database faster, it'll cause an increase in space usage since both tables would need numerous duplicate fields. 'fd' will look something like this:

1. checksum
    - This'll be used to identify files. Files will be hashed with SHA256.
2. basename
    - This'll be the path to the file or directory's parent directory. If a user were to try and print out the contents of a directory, the database would be queried according to this basename and all files readable by the user displayed.
3. name
    - name of file or directory
4. owner
    - username of owner of file or directory
5. directory
    - 1: directory, 0: file
6. read permissions
    - list of users who have read permissions.
7. write permissions
    - list of users who have write permissions

8. n-owner

    - number of owners; for example, 3 if three users upload the same file; decreased to 2 if one of the uploaders deletes file from their dropbox.

Information concerning the metadata of files will be stored inside the 'fd' database. The actual files, however, will be placed in a protected '/uploads' folder.

When a file is created, it'll be given a temporary name based on its 'checksum' inside the '/uploads' folder. When files are requested by a user, after the user has been validated by their session ID and the table has been queried to make sure the file exists, the user has readable and writable permissions, etc. the file will be located according to its 'checksum', renamed based on the 'name' field in the database, and moved to a '/tmp' folder accessible only by the user. From here, the user will be able to edit the file.

## 3.2 vulnerabilities and difficulties

### 3.2.1 computational complexity

Key to this design is the guarantee that the system will exit cleanly. If this was not an invariant and we had to protect against sudden crashes, the above design might be too expensive. This is because once a user finishes editing a file, a new 'checksum' for the file needs to be created, the file needs to be copied back into the '/uploads' folder, and if the user is finished, the '/tmp' folder needs to be removed. These operations are computationally substantial such that, if the system had to continually back up files users were working on, it might prove prohibitive.

### 3.2.2 client computation

A concern of ours, too, relates to the path. Let's say a user wants to list the files in their current directory, which is currently "/home/files". To do this, they can make a call to list with "." as path. Or, they might make a call to list with "../files" as path. Or, again, they might make a call to "/////////home/files"; all are valid path calls to print out the contents of the current directory.

If we simply compare the string of the path the user gives with the basename in the 'fd' table, it might not match. For example, maybe the user has read permissions for "/home/files", but when "///////home/files" is used, the string comparison comes back false.

We've yet to decide how we want to handle this. We could rely on the user to provide a simplified version of the path, however, this might introduce too much ambiguity as to what constitutes a 'simplified path'. As such, we'll need to make sure to handle side cases where the path provided by the user is syntactically different from the path in the 'fd' table, but structurally the same.

#### 3.2.3 errors

In handling the file system we'll need to be very careful how we check the database and what errors we return to the user. We don't want to leak information about the files in '/uploads' or allow a user to list files in a directory they don't have 'read' permissions for, open files they don't have 'write' permissions for, cd into a file that's not a directory, etc. There are numerous errors we'll need to diligently check for using our 'fd' table whenever the user tries to navigate the file system.

Additionally, similar to Authentication, since we're using a database and taking user input, we'll need to remember to always sanitize input via prepared statements.

# 4 Sharing

## 4.1 random string identifiers

We considered a couple strategies when it comes to sharing files. First, we considered the approach Facebook uses when it comes to sharing private photos; i.e. create a random string identifier for the photo, with the assumption users without permission wouldn't be able to guess the identifier.

The upside to this is it's computationally inexpensive; sharing files can be done without updating any databases. The downside, though, is it limits granularity; there's no real good way to revoke individual permissions once a user knows the file's identifier.

## 4.2 updates to database

A different approach would be to update the database. In this approach, if a file is shared readable with another user the database is updated giving that user read permissions, but not write permissions. When the user tries to access this file, the file is copied from the '/uploads' folder to the user's '/tmp' folder. Once the user is finished with the file, the file is not copied back into the '/uploads' directory, instead, the '/tmp' folder is simply removed. If a file was shared with read and write permissions, the only difference now would be the file would be copied back into the '/uploads' directory, the checksum updated, and the '/tmp' folder deleted.

With this approach, whenever a user with read permissions to a shared file accesses that file, they'll get the latest version of the file. However, nothing they do to the file will persist. Furthermore, read (and write) access to files can be more finely regulated through changes to the 'fd' table. Computationally this might be more expensive, but it provides more control on how files are shared.

Although we haven't decided completely, I think we're leaning more towards the latter approach. The 'fd' table will control who can read and edit files, with updates easily changing the access permissions of users. In the current implementation, we won't need to create a directory inside the user's root. Instead, files will still be held in a protected '/uploads' directory and only served up to the user upon request.

Furthermore, since the database differentiates between the owner of the file and users who have read/write permissions, we could restrict who can share the file, stipulating only the owner has this permission. Overall, this design protects well against unwanted users gaining access to files, other users passing on shared files, and chains of permissions lasting longer than desired by the owner of the file.

# 5 Deduplication

In order to handle deduplication, we might want to make use of an additional 'sharing' database. This database will hold the file's checksum, the owner of the file, and the users the owner shared the file with.

Identical files will be detected via their checksum; the hash of two files should be identical only if the two files are the same.

If a file is attempting to be uploaded to the file system, but that file's checksum is already in the 'fd' database, another entry will be placed in the 'fd' database, giving an additional owner to the file (along with other necessary fields like 'read', 'write', etc). Additionally, 'n-owners' will be incremented. This is important, since we should only delete files and directories when 'n-owners' equals zero. A new file won't be added to the '/uploads' directory.

As long as the file never changes, both owners and any users they share the file with will refer to the same file. Once a change is made to a file, however, the file will be copied and users differentiated.

Concretely, let's say user A and B both upload the same file. Since this file's checksum is the same, A and B are both marked as owners for the same file. A shares the file with C and B shares the file with D. These updates are added to the 'sharing' database. Still, whenever any of these users open the file, they'll open the same file. Let's say A, however, makes changes to the file. Now, when the file is copied back to '/uploads' it'll have a new checksum. A new entry will be made in the 'fd' database to refer to this new file, and all the users shared the file by A in the 'shared' database updated to have the correct permissions with the new file.

Key to this is keeping track of chains of owners and who those owners share the file with. We don't want, for example, a file that's been deduplicated to be edited

by one user, and then a different user, not related to the first, to see changes. This is why we'll make use of a separate database to clearly differentiate who gets access to what files once a change is made to the original.

# 6 API Calls

In the spec, it mentions client computation and how it's good to expect the client to perform some work, which'll make the logic server side easier. However, when it comes to API calls, in general we feel the more abstraction the better; the less a user can glean about the underlying functionality server side, the fewer attack vectors will be open to them. Here is just a basic list of some of the API calls we anticipate providing functionality for:

- **Login**: will take a username and password and return session ID to the client executable. The user, however, will not know this session ID.
- **List**: will take a path string from the user. Using the user's session ID, it will verify if the user has read permissions for the path specified. If the user does, list will print out all files and directories the user has permissions to view.
- **Cd**: takes a path string. Again, verifying the user with its session ID, cd will first verify first that the path is an actual directory, then that the user has permission to access this directory. If it does, it'll update the user's pwd in the 'user' db. It'll return 0 on success.
- **Upload**: takes a file and path. Upload will do similar verification checks as above and if the user has permissions to edit the directory specified in the path, 'fd' will be updated with the file's information and the file placed in the '/uploads' directory.
- **Download**: takes a path. Will verify that the path exists and that the user has permissions to access this path by consulting the 'fd' table. If the user does have access, the file will be coped out of '/uploads' into the user's root directory.
- **Mkdir**: takes a path. This'll function very similar to upload, except it'll simply make an entry into the 'fd' database without actually adding anything to '/uploads'.
- **Remove**: takes a path. It will verify the user is the owner of the file or directory and delete it.
- **Share**: a file name, another user's username, and the permission you wish to give to the user. Depending on how we implement the above, we might first check to see if the user trying to share a file is the owner of the file.

All API functions will have access to the user's session ID, however, we will not expect the user to include their session ID in calls to the API. If the session times out or the user has not yet logged in, the user will be asked to login first.

# 7 Other

As we implement the above functionality, we'll make extensive use of unit tests as we go. I think I saw a good testing package described here. This will hopefully provide some verification of the correctness of our code as we go.

Also, it's important to note that it's very likely many things will change as we start to implement this. As we look closer at the support code, undoubtedly some of the implementation details will fluctuate. However, much of what we've tried to do is prevent many of the vulnerabilities that were apparent in both 'flag' and 'handin', which we believe we should be able to do following the above outline.