DarkFS - An Encrypted File System

Team: Arjun Narayanan, Yuta

1. Motivation

In many software applications, we want to store files in a remote, untrusted file server. With an untrusted file server, we have a number of potential problems that we deal with. We don't want the file server to see plaintext data or filenames. We also need to detect whenever the file server tampers with data.

Beyond protection against an untrusted file server, we need to support multiple users and file sharing. Users should only be able to read/write the files that they have proper permissions for.

2. Threat Model

Server: The server may be malicious. It could read filenames and file contents that it stores. It could also tamper with the files. For example, It might inject it's own content into a file. Or it could swap one file's contents with those of another file.

Other Users: Without proper enforcement of permissions, a user could read/write a file he does not have access to.

3. Brief Overview

The DarkFS client is written in python. We use Dropbox to represent the untrusted file server. The client is run from the command-line and has a command-line interface.

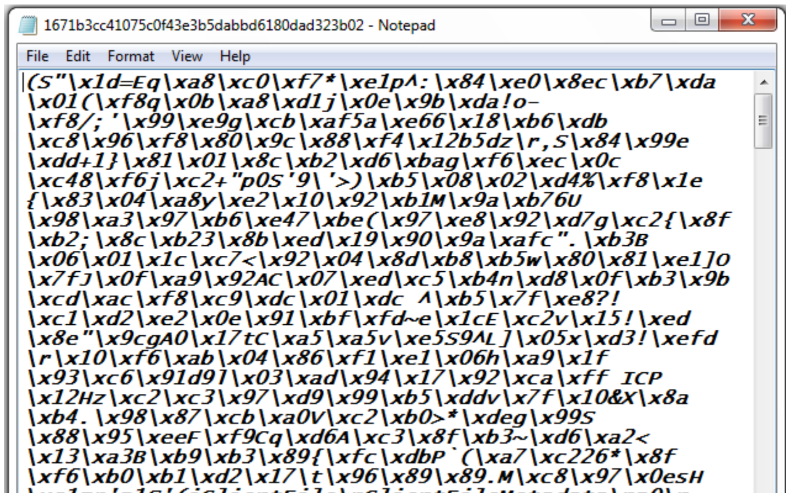
For security, we mainly use RSA and AES encryption and RSA signing. RSA and AES encryption are used to hide file contents from people who don't have permission to read the file. RSA signing is used to make sure that our files are not tampered with and are written only by users with proper permissions.

4. Server

The server is very simple. It stores and retrieves files. As mentioned before, we use Dropbox for our remote server. We have created a Dropbox account just for this filesystem. All of the clients will connect to this Dropbox instance.

A file on the server has four parts: encrypted filename, encrypted contents, metadata, and a sign file. The most important part of the metadata is the permissions for the file. This is one of our key design decisions. The permissions for a file are stored directly in the file. This means that we do not need any sort of centralized permissions store. We will explain how we enforce these decentralized permissions later. The role of the sign file will also be explained later.

For a given file, the server never sees the actual filename or contents. It only ever sees the encrypted filename and encrypted contents. As we can see in the image below, the server view of a file shows an encrypted filename (beginning with 1671b) and encrypted contents.



```
(S"\x1d=Eq\xa8\xc0\xf7*\xe1p^:\x84\xe0\x8ec\xb7\xda
\x01(\xf8q\x0b\xa8\xd1j\x0e\x9b\xda!o-
\xf8/;'\x99\xe9g\xcb\xaf5a\xe66\x18\xb6\xdb
\xc8\x96\xf8\x80\x9c\x88\xf4\x12b5dz\r,S\x84\x99e
\xdd+1}\x81\x01\x8c\xb2\xd6\xbag\xf6\xec\x0c
\xc48\xf6j\xc2+"p0S'9\'>)\xb5\x08\x02\xd4%\xf8\x1e
{\x83\x04\xa8y\xe2\x10\x92\xb1M\x9a\xb76U
\x98\xa3\x97\xb6\xe47\xbe(\x97\xe8\x92\xd7g\xc2{\x8f
\xb2;\x8c\xb23\x8b\xed\x19\x90\x9a\xafc". \xb3B
\x06\x01\x1c\xc7<\x92\x04\x8d\xb8\xb5w\x80\x81\xe1]O
\x7fJ\x0f\xa9\x92AC\x07\xed\xc5\xb4n\xd8\x0f\xb3\x9b
\xcd\xac\xf8\xc9\xdc\x01\xdc ^\xb5\x7f\xe8?!
\xc1\xd2\xe2\x0e\x91\xbf\xfd~e\x1cE\xc2v\x15!\xed
\x8e"\x9cgA0\x17tC\xa5\xa5v\xe5S9^L]\x05x\xd3!\xefd
\r\x10\xf6\xab\x04\x86\xf1\xe1\x06h\xa9\x1f
\x93\xc6\x91d91\x03\xad\x94\x17\x92\xca\xff ICP
\x12Hz\xc2\xc3\x97\xd9\x99\xb5\xddv\x7f\x10&X\x8a
\xb4. \x98\x87\xcb\xa0V\xc2\xb0>*\xdeg\x99S
\x88\x95\xeeF\xf9Cq\xd6A\xc3\x8f\xb3~\xd6\xa2<
\x13\xa3B\xb9\xb3\x89{\xfc\xdbP`(\xa7\xc226*\x8f
\xf6\xb0\xb1\xd2\x17\t\x96\x89\x89.M\xc8\x97\x0esH
```

Figure 1. Screen shot of encrypted file in Notepad.

5. DarkFS Client

The DarkFS client serves as the interface for all users. All security and integrity checks are performed on the client side.

5.1 API

edit_file [filename]
- Open or edit a local file. The edited file is not uploaded to the server.

upload_file [filename]
- Upload a file to the server. DarkFS will ask the user to enter the permissions for the file, specified as (user, ‘read’) or (user, ‘write), where user is the person you are sharing the file with. You can share a file with as many users as you want.

update_file [filename]
- Upload the local copy and replace any existing version on the server. The owner can also update file sharing permissions.

download_file [filename]
- Download file from the server and replace local copy if one exists.

delete_file [filename]
- Delete a file on the server. Also delete the local copy if it exists.

rename_file [oldFilename] [newFilename]

- Rename a file on the server. For safety, the user should re-download a file after renaming it

ls

- List all files in a user's local DarkFS directory. Can be downloaded (possibly shared) files or files the user has created but not yet uploaded.

list_files

- List user's files on the remote server. Includes files that have been shared with the user.

5.2 Cryptographic keys and their purpose

Every user of DarkFS has an RSA public key and an RSA private key. The private key is stored locally on the user's machine. We assume each user knows the public keys of all users. The public keys can be shared securely using PGP.

Each file is associated with its own randomly generated AES key. When a file is created, we generate a random AES key for that file. These keys are persisted in the a client-side database. A user needs this key to be able to decrypt the data and actually read the file. Thus, we call this the file_read_key. The file_read_key represents a READ permission.

Each file is associated with its own RSA public/private key pair. These are also randomly generated when the file is created. Call the public key file_public_key and the private key file_write_key. When a user tries to update a file, we generate an RSA signature from the file_write_key and the contents of the file. This is the signFile mentioned in Section 4. Before allowing the update, DarkFS uses the file_public_key to check that the signature is valid. By valid, we mean that the new file contents were generated using file_write_key and not another private key. This means that only users with the file_write_key can update a file. Thus file_write_key represents a WRITE permission.

Encryption Key	Type	Purpose
User public/private key	RSA	- Identify different users - Allows encryption of file_read_key and file_write_key when sharing files
file_read_key	AES	- Encrypt and decrypt the contents of a file. - READ capability
file_public_key/file_write_key	RSA	- WRITE capability

Figure 2. Cryptographic keys in our system

5.3 File sharing

We stated earlier that the permissions are stored directly in the file. The format of the permissions is a hash map mapping shared users to the keys required. For a given shared user, the value of the hash map will contain a file_read_key if the shared user is given READ permission, and both a file_read_key and a file_write_key if the user is given WRITE permission. These keys are encrypted with each shared user's public key, such that only the appropriate user can decrypt them.

Suppose we have two users, user_1 and user_2. User_1 uploads a file, file_1, to the server and wants to give user_2 permission to read it. After user_1 adds this permission to the file, the DarkFS client will do the following. It will encrypt the file_read_key with user_2's public key. Call this encrypted_file_read_key. DarkFS will add an entry [user_2 → (encrypted_file_read_key,null)] to the permissions map.

Now, suppose user_1 wants to give user_2 permission to write a file. Similar to before, DarkFS will encrypt the file_write_key using user_2's public key. This encrypted key will be added to the map.

We reiterate that the permissions are enforced by the client. The DarkFS client will check the permissions. The server has no role in this. We believe this is the correct design, because we cannot trust the server to enforce the permissions.

NOTE: in the actual implementation, RSA keys are too long to encrypt again with another RSA key, so we had to implement an intermediate form of encryption. We referenced CamelFS from 6.858 2013 regarding this particular issue.

5.4 Protecting the files against the server

We guarantee that all file names as well as file contents are encrypted and therefore confidential and protected from the file server. The encryption of filenames is non-deterministic, because we randomly generate AES keys to encrypt every file. The encrypted filenames are further protected by the use of a random initialization vector to seed the AES encryption process.

We can detect whenever the server maliciously modifies the data. This is done in the same way we check if a user has write permission. When the server returns a file, we use RSA to verify the signature is valid. As before, we check if the returned contents were written with the correct file_write_key. If not, we know the server has tampered with the data.

We can also detect if the server replaces one file's contents with those of another file. Since the contents and signature of the original file are replaced, the new signature is technically correct. However, since each file is encrypted using a unique AES encryption key, if the server replaces file_1’s contents with file_2’s contents, then the DarkFS client will not be able to decrypt the data. Thus, we can detect that the server has acted maliciously.

Finally, file metadata (permissions) is protected in a different way than file contents. A user stores a cryptographic hash of the metadata for each file that he owns (in a client-side database). Upon downloading a file from the server, the owner’s client checks the hash of the received metadata against the saved hash. This means that if a file is shared to some user, that user cannot check if the metadata is valid. Only the owner can. We accept this quirk because only the owner can manage the metadata. The worst that could happen to a shared user is that they lose their access to the file. A shared user can still detect their loss of access and ask the owner to fix the issue.

6. Future work

Future work would involve adding support for directories. Our file system now only supports a flat structure. We only had two group members and wanted to focus on getting the encryption and security right. We felt making our file system secure was more important than adding support for directories.

In a different light, we later want to support freshness guarantees in our system.