# PARANOIA

**(Another Encrypted File System)**

Happy Enchill (henchill) Quan Nguyen (qmn) Aakriti Shroff (aakriti)

## Introduction and Threat Model

Online file storage allows users to store their files in a highly reliable environment and access them from anywhere in the world. However, online backup services are prone to data breaches and surveillance. We would like to offer users the convenience and benefits of remote data store, but we also want to protect users' data. *Paranoia*, an encrypted file system, helps provide these guarantees.

In *Paranoia*, we assume that there are legitimate users and malicious users connecting to a potentially malicious server to store files. Legitimate users can create, delete, read, and write files, as with any other file system. However, malicious users cannot alter the contents of legitimate users' files if they are not explicitly authorized. Any malicious actions taken by the server may not be prevented, but will be detectable by the client.

## Honest Client-Server Interaction

A user, perhaps Alice, obtains the server's public key on which to encrypt all future communication. Alice registers with the server by sending her username along with her public key, and now Alice and the server can communicate securely. In a production-quality deployment of *Paranoia*, we expect that the client and server connect over SSL to protect the initial key exchange and subsequent communications, but the implementation of SSL is beyond the scope of our project.

Alice encrypts files with a symmetric encryption key (the *file key*) that she generates. In addition to the file contents, the ciphertext contains the plaintext filename and the timestamp of its most recent modification. Files are contained within directories, and at registration, each user has their own home directory (i.e., Alice has a "home" at `/Alice`). Directories may be arbitrarily nested in other directories, yielding a hierarchical Unix-like file system.

A separately-maintained access control list (ACL) contains list of users and their associated file permissions, in plain text but signed by the owner. Each file has its own ACL, specifying only file permissions. Each directory has its own ACL, specifying read/write permissions on that directory. A directory's ACL also contains the file key for decrypting all files and their filenames.[1] In this example, the ACL for directory `/Alice` contains the file key encrypted with Alice's public key, in addition to Alice's read/write permission.

Suppose Alice wants to share a file with Bob. The owner, Alice, obtains the file key and Bob's public key. Alice encrypts the file key with Bob's public key and stores it alongside the granted permissions in the ACL. Now, Bob can access the file by obtaining the ACL.

[1] This reduces the potential explosion in the number of symmetric keys.

## Coping with a Malicious Server

Because the encrypted file contains the filename, Alice can verify that the file reflects the filename she requested. The server cannot serve another file in place of the one requested. The server also cannot serve an older file if it has already served a newer file. Alice's client retains the timestamp attached to the most recent version of the encrypted file, and rejects files older than the given timestamp. This policy provides a primitive freshness guarantee.

The server cannot modify the ACL because only the owner may update the ACL. If the server serves Alice's file, but its ACL is not signed by Alice, it could be regarded as a potential attack. Furthermore, the ACL contains the filename, to ensure the server cannot serve the ACL for another file. The ACL also contains a timestamp, preventing a rollback attack in which the server serves an old version of the right file's ACL.

Directories' contents are also signed with the owner's public key, ensuring that the server cannot create or delete files without being detected. The user verifies the integrity of the directory listing before making any changes made to the directory (creating, deleting, or renaming files and directories).

## Coping with Malicious Users

If the server can be trusted to enforce the ACLs, no malicious user, perhaps named Mallory, can be permitted to read Alice's files. Even in the event the server serves Mallory the files, the encryption will thwart any attempts at decryption.

Suppose Alice has shared the file with Mallory and wants to revoke sharing. Alice can remove Mallory from the ACL for that file, and trust that the server will enforce the ACLs. If Alice wants to ensure Mallory has no access, Alice can re-encrypt the file with a new key and distribute it to all sharers except Mallory.

## System Implementation

The server is a continuously running Python program that opens a socket on TCP port 1025 and listens for client connections. Presently, the server stores files as objects in arbitrary nestings of directory objects within the running Python interpreter instance.

Clients connect with a client-side Python program and interact with the server by passing (encrypted) JSON objects. Due to the implementation of sockets in Python, we are limited to sending small buffers of data at a time. Because RSA encryption and base64 encoding grows the representation of encrypted data, we often have to send several packets to be reassembled once they reach the server.

*Paranoia* uses two encryption systems: 2048-bit RSA with OAEP for asymmetric encryption (encrypting communication, encrypting keys, and signing), and AES with 128-bit block sizes for symmetric encryption (encrypting files and their names). Encryption is provided

by the PyCrypto library, wrapped with other logic to standardize communication between the client and server.

## Data Structures

*Paranoia* leverages a few data structures to provide an encrypted file system atop an untrusted server.

*File System*
We abstract each directory and file as *DirEntry* and *FileEntry* objects, and the the file system is essentially an array of *DirEntry* objects. The file system's hierarchy is achieved by nesting *FileEntries* and sub-directories inside *DirEntry objects.* Upon registration, each user is assigned a home folder, and is initially given permission to change contents of his/her own home directory. If the user wants to create or delete entries under a different user's home directory, they must first receive authorization to do so. In our design, this is done by storing a file key encrypted with the recipient's public key in the ACL.

*DirEntry*
Each directory is identified by a *DirEntry* that stores:

- Directory name
- Owner (creator) : Only the owner has permissions to change an ACL associated with the directory.
- Contents: List of *FileEntry* and sub-directory *DirEntry* objects
- ACL: List of sub-directory ACLs. A directory's ACL lists users' permissions and file keys encrypted with the user's public key. A parent directory holds the ACLs for it's subdirectories as directory names need to be decrypted correctly when the client tries to list directory contents via an *ls* command.

*FileEntry*
A File in *Paranoia* is internally represented as a FileEntry and is similar to a DirEntry except that its contents store the encrypted file contents, and it's ACL lists per-user permissions. A file's filename and contents are encrypted using the keys listed in the parent directory's ACL. As discussed in the previous sections, we decided against a different file key per file to reduce the number of symmetric keys that the client needs to decrypt files within a directory. We depend on the server to correctly process ACLs and serve files that the user is authorized to view. Each FileEntry is represented as:

- File Name
- Owner (creater): Only the owner can change the ACL associated with a file.
- Contents: Encrypted file contents
- ACL: A *FileEntry* ACL stores the per-user permissions for the file. The file key required to decrypt the file name and contents is the same for every file in directory, and can be obtained with the directory's ACL.

*Messages:*

Messages are encoded in JSON, which translate effortlessly into a Python dictionary. From this human-readable representation, we can obtain information about a request or response. In addition to the action, the requesting, and the signature, a message can contain the encrypted contents of files, ACLs, and other metadata. Timestamps included with the signed contents help honest servers resist replay attacks based on messages captured from the network.

Messages sent to the server have the same simple structure. Upon reception, the server uses username to obtain the public key of the user and verifies the signature.

msg → {'username': curr_user, 'signature': data_sig, 'data': actual_data}

The 'data' attribute in the message changes based on the action being sent to the server, but an example data object for create would be:

'data' → {'username': curr_user,
'action': 'create',
'filename': pathlist,
'file': contents,
'acl': acl,
'signature_acl': acl_signature}

The 'filename' entry is a list containing the parent directories of the file, and the filename itself all encrypted with their appropriate shared key. This is how all pathnames are passed to the server.

Also, the 'file' entry contains contents of the file encrypted with the filekey for that file. This ensure that the server does not know the name or contents of files.

## Conclusion

*Paranoia* is an encrypted file system where the server cannot tamper with any data it hosts while still providing users the ability to share files and directories in a robust manner. While we provide the freshness guarantee that a client will not see any file that is older than the most recent file it received, *Paranoia* doesn't guarantee fork-consistency as described in SUNDR. Nevertheless, *Paranoia* makes for an interesting starting point to investigate implementing stronger freshness guarantees and incorporate versioning into an encrypted file system.

---

Code Repo: https://github.com/henchill/encrypted_file_system