EncryptFS:

An Encrypted File System

By:

Jorge Ornelas (joor2992) | Ulziibayar Otgonbaatar (ulziibay) | Otitochi Mbagwu (otitochi)

Abstract

EncryptFS is an encrypted file system that stores files on an untrusted server. Users can add, modify, rename and delete files on the server and the server cannot obtain any information about any of the files or their names. This is done using RSA signing along with AES encryption of files. File sharing is done via capabilities that allow other users to either read or write to a file. The interface for EncryptFS is one of many other linux programs. The server and client are launched from command line and commands and arguments are also done via the command line. In order to make this easier we developed a shell to run these commands. Overall, we were successful in implementing the file system, but deviated from our original design of peer-to-peer encrypted file sharing.

Intro

As one of our guest presenters once stated, “ In security, at some point there are has to be some trust”. We believe that with regards to remote file systems, the trust should lie with the clients. In this paper, we will elaborate on the motivation, design, and implementation of EncryptFS, an encrypted file system.

Motivation

Recently, we have had our trust in the government and in big tech companies, shaken by the breaches in privacy that have been brought forth. This has not only increased the need for anonymity guaranteeing systems such as Tor, it has also increased a desire to encrypt data in places where we do not have full control. One of these places is remote file storage.

Using ‘cloud’ storage services has become very popular in the last decade. There are several advantages to using these services. It grants additional storage space, a location to backup data, and enables easy file sharing.

Many cloud storage services are based on a centralized file system, i.e. SkyDrive, Drive, and Dropbox. However, given the recent NSA findings, it is obvious that the entity controlling the central server still holds the ability to decrypt the files users are storing.

This is why we have designed a file system that through the use of encryption and clever permission settings, does not require the users to trust the central server.

Threat Model

Malicious file server

We are going to assume that the server can modify, read and delete files that are stored on its hard drive. We are also assuming that attackers can send arbitrary messages to the server.

Malicious user

A user can attempt to read and write files to which they do not have permission to or once had permission to.

Eavesdropping attacker

We assume that there are attackers eavesdropping on the network that users are connected to. Such attackers are able to carry out replay attacks and can intercept any network package that users exchange. Under such conditions, it is important for our system not to reveal any private information over the network.

Design

Our design consists of the use of a linux style user interface, capabilities for permission granting, RSA signing and AES encryption for verifiable file transfer and storage, and a transitively read-only directory structure.

Interface

Our interface is similar to other linux programs. The first thing one has to do is run the server, “server.py”. Once the server is taking requests, running “client.py” and giving it arguments will allow a user to create and modify files among other things. A shell can be launched running “client.py shell” that makes running commands easier.

The following are possible commands:

Create a file
- put [-h] [-n NAME] [-d DATA] [-c WRITECAP] [--t]

Get a file from server
- get [-h] [-n NAME] [-c CAP] [--t]

Show files in directory
- ls [-h] [--v] [-p PATH]

Rename a file
- rn [-h] filename newfilename

Delete a file
- rm [-h] file

Make a directory
- mkdir [-h] [-p PATH] -n NAME [-r]

File creation

When a user request to make a file or directory on the server, a new randomly generated pair of RSA private and public keys are generated. Then, by using the SHA hashing the public and private key respectively, we get a fingerprint and write-key for the file/directory. Essentially, write-key serves as a write permission and the fingerprint is used to validate the public keys.We call the “write-key:fingerprint” the write capability for the file/system. By hashing the write-key, we get the read-key; the read-key concatted with the fingerprint is said to be the read capability.

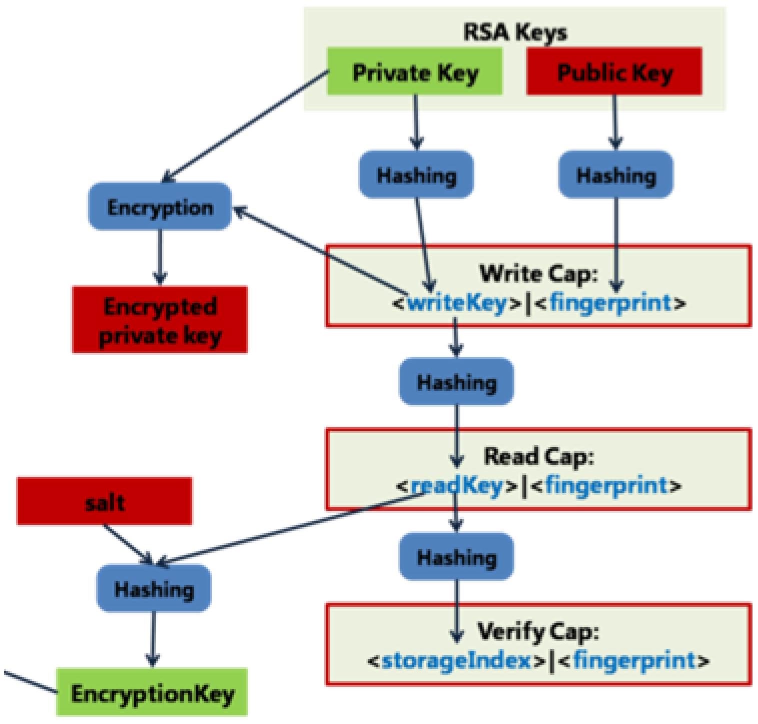
After the capabilities are created on the client side, a text editor spawns up and the user enters the data. Upon closing the file, the data is encrypted and signed with the private key so that the integrity could be checked on the server side and the data is not visible to eavesdroppers. The security section of our paper talks about that in detail.

Capabilities

The claim of our design is that whoever holds the write-capability can identify, access, and modify the file. Whoever holds the read-capability can identify, and read the file. However, the read-capability holder cannot modify the file, when the server is being honest. The security section of our paper talks more about the specific scenarios and how our system is resilient to those.

File Sharing

Users can share files and directories by exchanging file and directory capabilities. Files have read and write capabilities that are generated when a file is created. A read capability for a file can be derived from its write capability.



Encryption and Integrity check

Our method of encryption and key generation borrows a lot from the Tahoe-lafs[1] server design. Essentially when a file is created, we use the write-key to encrypt the private-key associated with that file. Also, we use the read-key hashed with a salt to get an AES encryption key that is used to encrypt the raw data. Then, we concat these two pieces--encrypted private key, encrypted data--and sign it with the private key. Finally, send the encrypted data, the signature along with a public key and the fingerprint to get the data that we send over the network.

When the data is received, an honest server uses the public key attached in the data to see if the data was signed with the private key that is associated with that key. It also makes sure the public key is the correct key by matching it against the Having validated the data attached in the modified without proper permissions.

Directory Structure

Directories are files that is a table of file and subdirectory names to the capabilities associated with those files. The directory capabilities are created in the exact same manner as the file capabilities.

Security
Our system is designed to be resilient against the following attacks.

1. violate confidentiality: the attacker gets to view data to which you have not granted them access
2. violate integrity: the attacker convinces you that the wrong data is actually the data you were intending to retrieve
3. violate unforgeability: the attacker gets to modify a mutable file or directory (either the pathnames or the file contents) to which you have not given them write permission without being detected.

Design Decisions

At the beginning of this project, we wanted to have a peer to peer system that allowed users to store and share encrypted files on each others machines. This turned out to be much more difficult than expected. We then decided to turn to the original system of having one untrusted server that handled all the files.

Another choice we had was to make the user interface online or within the linux environment. We decided to go with the linux environment because of simplicity and stronger security. Using a web interface would introduce many more security threats that would not be worth the arguably better user experience.

Testing

We tested rigorously to make sure every feature of our file system worked properly. This helped in the development process by allowing us to find bugs early on and avoiding having to deal with multiple bugs later on. Testing became difficult when dealing with multiple users since one had to switch to other users and linux does not make this easy, so some tests we just did manually.

Conclusion

In conclusion, EncryptFS allows users to store and share files on an untrusted server that cannot modify or delete files without being detected. In this day and age where privacy is hard to find, we cannot be too careful and EncryptFS and systems similar to it will allow users to have some peace of mind with regards to their files.

References

1. https://code.google.com/p/nilestore/wiki/TahoeLAFSBasics
2. https://tahoe-lafs.org/trac/tahoe-lafs/browser/trunk/docs/architecture.rst
3. https://tahoe-lafs.org/~warner/pycon-tahoe.html