Secure File Store - Design Document

Submitted By - Vinayak Trivedi (160790) AND Ayush Bansal (160177)

Design for storing User-data

Let the client side gives *username* and *password* to server (might be hashed or not, depending on client code), then we will store it in our data-store with key (call it Userkey) generated by hashing the username and password provided with string "user", i.e. -

Userkey = sha256 (*username* + *password* + "user")

The value stored at this key will be a User Struct, described as follows - (Note that we have stored all hashed Usernames at some pre-defined key(can be chosen at coding-time) in datastore, so that we may check if an Username already exists at time of InitUser).

```
User Struct {
        Username
        Salt
        IV
        Password
        Private_RSA_key
        Files
        HMAC_check
}
```

At time of InitUser, we generate a RSA key-pair for that user. We register his Public Key in Key-store and store his Private key in User Struct.

Explanation of User Struct -

Username -> sha256(*username*)
Salt -> Generated Randomly and stored in plain text
IV -> Generated Randomly and stored in plain text
Password -> sha256 (*password* + Salt)
Private_RSA_key -> CFB Encrypted private RSA key of user. It is Encrypted with initialisation vector as IV, and key being generated from Argon2 by giving *password* and Salt as argument.
Files -> CFB encrypted An array of key-value pairs where key is the Filekey (the key used to get the File struct from datastore) and value being the token. Token is used encrypt Data and other things in corresponding File Struct. Key and IV used for CFB encrypting this array is same as used above.
HMAC_check -> This is for Integrity check. All the above information of User struct (Username, Salt, IV,Password, Private_RSA_Key,Files) is treated as a message and key being generated from Argon2 By giving *password* and Salt as argument. This Key and Message is used to build a HMAC hash which will be used for checking if anything has been changed in this struct.

Design For File-Store and Append

The Files will be stored in Struct File when created first time/over-written, while on appending in a File, new data is stored in a new Append Struct. Given any *filename* ,from client with *Userkey* (When User Logs in, we generate his session in server and store his Userkey), the key to index datastore to get File Struct will be-

FileKey -> sha256(*filename* + *UserKey* + "file")

File Struct {

Symlink - Nil if it is a original file, else equal to FileKey which was shared with this name.
IV - Initialisation Vector for CFB Encrypting data
Data - Data of file, encrypted with CFB, using key as Token (which was stored in User Struct's Files Array) and IV as above.
Number_of_appends - Plain-text number of appends.
HMAC_check - The whole above data of this struct is used as message, and key being used as the Token, to generate the HMAC hash which is used for Integrity
}

When we append to any file, an Append-Key is generated and corresponding data stored in a new Append Struct, and number of appends incremented in main File Struct. Files which are symlinks, we get the original *FileKey* and increment appends in that file Struct. Each time we append, we update the HMAC check of File Struct using the Token we have for that file. Append-Key for any file is generated as follows -

Append-Key = sha256(*FileKey* + "append" + str(*Append_number*))

Append Struct is as follows -

Append Struct {
Data - CFB encrypted data with IV taken from File Struct and key as File Token.
HMAC_check - Message as above data and key as File Token to generate a hash, to be used for Integrity.
}

The Token used for Encryption and HMAC in file Struct is generated at time of file Store by a User (with corresponding *Userkey)* as follows (FileSalt is generated randomly and not stored anywhere)-

FileToken = sha256(*Userkey* + *FileKey* + *fileSalt*)

Design for Sharing/Revoking

When we wish to share any file, we generate a token as follows (FileToken is explained above)-

SharingToken = *FileToken* + *FileKey*

We encrypt this token with Public Key of receiver, and sign the Whole message with our (sender's) Private Key. Receiver verifies sign from Private key of sender, and decrypts using this private key. He will use the FileToken to read/manipulate things of File Struct, and FileKey to be value of symlink in his own fileStruct. He stores this FileToken in his User struct to use in future. Since FileToken is of constant length(256), we can easily split the FileKey and FileToken from SharingToken.

At time of Revoke, the User just changes the *FileToken* by changing the *fileSalt* (*UserKey* of user who is revoking and *FileKey* are known), so a new *FileToken* is generated. He then Decrypts the whole File Data using his previous fileToken (if he is not user who created the file or whom file is shared with, then this previous Token will be wrong, and he can't decrypt the file). He then encrypts the whole file Data using this new Token, updates the HMAC checks, and stores this new Token in his own User Struct. This way, other Users will no longer be able to get the File Data. (In order to traverse the file, we will use number of appends information).

Test-Cases

By choosing HMAC as check for Integrity, we have countered **Oracle Padding Attack**. By encrypting all of Userdata with his Private key, we have maintained confidentiality of User. Attacks to try searching for any specific Username, his files, his shared files, anything is not possible in this design setting. All shared Users are unaware of each other, so no attacks from that side is possible. All sharing/revocation is done directly through a single Token. Who owns a file, what content is in a file, everything is abstracted out and kept secret. Man-in-the-middle attack is not possible since everything is encrypted, and checked for Integrity by Signatures. Only Brute-force attacks are possible in this setting.