Muonium - Encrypted cloud storage

Paul Feuvraux

April 14, 2018

DRAFT

1 Introduction

This paper aims to explain the technical side of *Muonium*, a fully open source encrypted cloud storage.

Muonium aims to protect users' online privacy through *End-to-End Encryption* against malicious parties. Muonium automatically encrypts user's data, uploads and stores them on the cloud. Users can have access to their encrypted files anytime, anywhere.

2 Terms

In this section, we define all technical terms used throughout the paper.

- **File Encryption Key (FEK)**: 256-bit symmetric key derived from CEK. We use FEK to encrypt/decrypt files. Each file has its own FEK.
- **Hash function**: We use SHA-384 at the client-side and Blowfish with 192-bit of Salt at the server-side.
- **Key derivation function:** Every key derivation is performed with PBKDF2-HMAC-256. We denote this process as $KDF(P, salt, iter, len)$ where P is the user's passphrase, $salt$ is the 128-bit salt randomly generated, $iter$ is the iteration number, and len is the key length output.
- **Symmetric encryption**: We use AES-256 over GCM. We denote this process as $Enc(k, pl, iv, AD)$ where k is a symmetric key and pl is the plain-text to be encrypted, iv is the 128-bit randomly generated initialization vector, and AD is a 128-bit randomly generated word used as authentication data.
- **Symmetric decryption**: We use AES-256 over GCM; We denote this process as $Dec(k, x)$ where k is a symmetric key and x is the encrypted text to be decrypted.
- **Asymmetric encryption**: We use a 4096-bit public key, and is denoted as $AsymEnc(pubkey, content)$.
- **Asymmetric decryption**: We use a 4096-bit private key, and is denoted as $AsymDec(privkey, content)$.

3 Protocol

3.1 Registration

Users must register with a Muonium server in order to use its service. During this process, the user has to provide:

- **Email address**: Used for authentication procedures.
- **User name**: Used as an email address alternative for authentication procedures.
- **Login Password** LP: used for authentication system.
- **Passphrase** P: User's defined alphanumeric UTF-8 passphrase during registration on the client side.

The *user name* and the user's *email address* are stored in plain text in the database while the Login Password LP is double-hashed a first time at the client-side under SHA-384 and a second time at the server-side under Blowfish (described in figure 1).

While the user submits his credentials, Muonium client generates a Content Encryption Key (CEK), which is an alphanumeric UTF-8 randomly generated string. Then, an initialization vector (IV), authentication data (AD), and salt ($salt_{kek}$) are both generated on 128 bits. Once these cryptographic parameters generated, we derive the Key Encryption Key from P, such as: $KEK = KDF(P, salt_{kek}, 7000, 256)$. Afterwards, CEK is encrypted such as: $CEK_c = Enc(KEK, CEK, IVAD)$, and is sent to the server with the Json format, $packet_k$ containing CEK_c, IV, AD, and $salt_{kek}$ along with it as well.

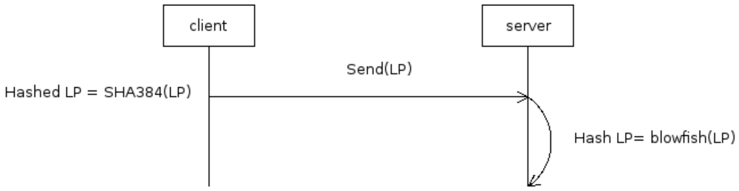

Figure 1: LP is hashed and stored on the server

The connection between *Client* and *Server* is encrypted over AES-256-GCM with TLS 1.2.

3.2 Connection

To use the service, a user must establish a connection with a Muonium server. There are two steps involved: *Authentication* and *CEK decryption.*

3.2.1 Authentication

The user types in his credentials (user name/email address and LP) plus his passphrase P. Muonium server then verifies these credentials.

3.2.2 CEK decryption

Once the user is authenticated, the server returns $packet_k$. From CEK_c (Json format) is taken $salt_{kek}$ in order to decrypt CEK_c. The client proceeds to a key derivation to recompute KEK, such as $KEK = KDF(P, salt_{kek}, 7000, 256)$.

The client then decrypts it such as $CEK = Dec(KEK, CEK_c, IV, AD)$ and stores CEK and KEK locally.

3.3 File Upload

3.3.1 Key derivation

In order to proceed to a key derivation, a salt ($salt_{fek}$) is generated on 128 bits. We use a strengthening by a factor of 7000. The output will be a 256-bit symmetric key. Such as $FEK = KDF(CEK, salt_{fek}, 7000, 256)$ where 7000 (strengthening by factor) and 256 (symmetric key output size) are constants.

3.3.2 Splitting

A file is split in 512MiB chunks. For each chunk are generated an IV and AD, both are generated on 128 bits.

3.3.3 Encryption

For every chunk (chk_x) an encryption is performed with the IV_x and AD_x of the chunk itself and the FEK of the file such as $encchk_x = Enc(FEK, chk_x, IV_x, AD_x, 128)$ where 128 is a constant which represent the tag length.

3.3.4 Encapsulation

Every chunk is encapsulated with the AD and IV such as $pck = (encchk_x || salt_{fek} || IV_x || AD_x)$.

3.3.5 Uploading process

Once encapsulated, the chunk is sent to the server. All chunks are sent one by one.

3.4 File Download

3.4.1 Extraction

The first chunk of the remote file is downloaded. We extract $salt_{fek}$ from it.

3.4.2 Key derivation

The $salt_{fek}$ is used to proceed to a key derivation, such as $FEK = KDF(CEK, salt_{fek}, 7000, 256)$.

3.4.3 Decryption

For every received chunk, the client decrypts it such as: $chk_x = Dec(FEK, encchk_x, IV_x, AD_x, 128)$.

3.4.4 Reassembling

For each chunk belonging to the same file, it is written in a remote file.

Such as $file = (chk_1, ..., chk_y)$, where y is the total number of chunks composing the remote file.

3.5 File sharing

Users can share files, internally (between Muonium users) and externally (via an external link).

3.5.1 External file sharing

Encryption.

The user chose to share his/her file with another Muonium user.

When the file is shared, the first chunk of the file is downloaded in order to extract the salt, which will be used to rebuild the FEK of the file such as $KDF(CEK, salt, 7000, 256)$. Once the FEK rebuilt, a new $salt_{dk}$ is randomly generated in order to derive the FEK into a derivation key DK. Then, we proceed to a key derivation such as $DK = KDF(p, salt_{dk}, 7000, 256)$, where p is the passphrase that the user had to type to share it with external users (people who don't use Muonium).

In order to encrypt FEK, an IV and AT are randomly generated on 128 bits.

Then, the encryption is performed such as $FEK_c = Enc(DK, FEK, IV, AD, 128)$, where FEK_c is the encrypted FEK. Once the FEK got encrypted, the cryptographic parameters are encapsulated with FEK_c such as $packet = (FEK_c||salt_{dk}||IV||AD)$.

Decryption.

An external user goes to the public link, type the passphrase, and download the file.

$packet$ is obtained from the database and sent to the external user. The user types the passphrase p. We extract $salt_{dk}$ from $packet$ and proceed to a key derivation such as $DK = (p, salt_{dk}, 7000, 256)$.

Once DK obtained from the key derivation process, we extract IV and AD from $packet$ and proceed to the decryption of FEK_c to get FEK such as $FEK = Dec(DK, FEK_c, IV, AD)$.

Once FEK obtained, every chunk of the file is downloaded, decrypted, and reassembled to recreate the decrypted file.

3.5.2 Internal file sharing

Internal file sharing uses asymmetric encryption, thus, the user needs to generate their pair of keys if it has not been done before. Asymmetric keys aren't generated at the registration and are generated on 4096 bits.

Encryption. The user types the username of the user they want to share the file with. If a match is found, then the shared-with user's public key $pubkey_u$ is downloaded to the client.

First of all, the first chunk of the file is downloaded, and we extract its *salt* in order to proceed to a key derivation for rebuilding FEK such as $FEK = KDF(CEK, salt, 7000, 256)$.

Once the FEK obtained, we proceed to the encryption of it such as $FEK_c = AsymEnc(pubkey_u, FEK)$, then FEK_c is sent to the server and is stored in the database.

Decryption. The user click on "shared with me", downloads the file, and gets their private key $priv_u$.

The client gets FEK_c from the database, and decrypts it such as $FEK = AsymDec(priv_u, FEK_c)$.

Once FEK obtained, every chunk of the file is downloaded, decrypted, and reassembled to recreate the decrypted file.

3.5.3 Organizations Feature

Keys management. Every sub-user that will be promoted as administrator (admin) later, will have to generate their pair of asymmetric keys. Hence, if the sub-user doesn't have their pair of asymmetric keys, the master won't be able to promote them as administrator.

Asymmetric keys are generated such as $(PK_a, PK_b) = GenAsym(RSA, l)$, where PK_a is the private key, PK_b the public key, and l is the length.

The master of the organization is a normal user (not sub-user), and gets the key of the organization encrypted thanks to its PKB_o.

We denote:

- CEK_u: sub-user's CEK
- KEK_u: sub-user's KEK
- PKA_o: asymmetric private key of the organization
- PKB_o: asymmetric public key of the organization
- A: $A = AsymEnc(PKB_o, CEK_u)$
- B: $B = Enc(KEK_u, CEK_u)$

Creation of the sub-user. The master creates the user with the following parameters:

- **username**
- **email** (optional)
- **temporary password**

When the user logs in for the first time, they will have to generate a CEK, and define a passphrase P in order to derivate it following the standard derivation key function as defined in the section **Terms** (as a normal user).

Promoting a sub-user as administrator. Every master may be able to either create or delete a user. A non-private sub-user cannot be promoted as admin. A sub-user without a pair of asymmetric keys won't be able to be promoted as admin.

The admin gets the sub-user's PKB_u and encrypts PKA_o with it such as $PKA_{oe} = AsymEnc(PKB_u, PKA_o)$. Afterwards, PKA_{oe} is stored on the server.

Non-private sub-ser creation. To begin with, the administrator randomly generates CEK_u, which is the CEK of the non-private sub-user, and pre-defines a passphrase P.

We compute KEK_u such as $KEK_u = KDF(P, S, 7000, 256)$, where S is the salt, a randomly generated bit array on 128 bits. Afterwards, CEK_u is encrypted such as $B = Enc(KEK_u, CEK_u, IV, AD)$. We send store it as a packet: $packet = (B||S||IV||AD)$. We encrypt it with PKB_o such as $A = AsymEnc(PKB_o, CEK_u)$ and store it as well.

Administrator logs in as a non-private user. Admin gets PKA_org, such as $PKA_org = AsymDec(PKA_u, PKA_orgenc)$. Once obtained, admin decrypts the non-private sub-user's CEK (CEK_u) such as $CEK_u = AsymDec(PKA_org, A$. Then, admin will be able to decrypt files from the non-private sub-user's storage space.

4 Conclusion

Muonium is based on the end-to-end encryption model, consequently nobody can access a user's data except the user himself.

5 Notes

- **Key derivation algorithm**: it exists several improved key derivation algorithms, but we've chosen to use PBKDF2-HMAC-SHA256 for its performances and safety.
- **FEK leaks**: when an external or internal user gets to be shared a file with, they get to know FEK, which means that they would be able to leak it publicly. Hence, it is recommended to re-upload the file once the user gets done to share it.
- **SRP instead of double-hashing**: using the SRP protocol would be more secure for the user authentication. We plan to implement it.

6 Thanks to

Hoang Long Nguyen, for your help at writing this whitepaper.