

Muonium - Encrypted cloud storage

Paul Feuvraux

December 19, 2017

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.

- **Content Encryption Key (CEK):** Alphanumeric UTF-8 randomly generated string. It is used in the Key derivation function.
- **Passphrase (P):** User's defined alphanumeric UTF-8 passphrase during registration on the client side.
- **File Encryption Key (FEK):** 256-bit symmetric key derived from *CEK*. We use *FEK* to encrypt/decrypt files. Each file has its own FEK.
- **Login Password LP:** Used for authentication procedures
- **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 :**
- **Passphrase P :** Used in the cryptographic system.

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).

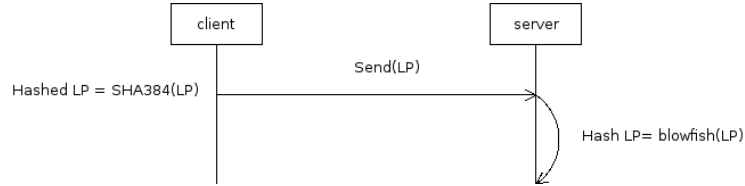


Figure 1: LP is hashed and stored on the server

While the user submits his credentials, Muonium client generates a CEK , encrypts it with P and sends along with the user's credentials to the server as follow. 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, Muonium server return $Enc(P, CEK)$. It is then decrypted as $Dec(P, CEK)$. Client stores CEK locally on ...

Diagram for both subsubsection

3.3 Upload

3.3.1 Key derivation

To proceed to a key derivation, a Salt 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 $F EK = KDF(CEK, Salt, 7000, 256)$ where 7000 (strengthening by factor) and 256 (symmetric key output size) are constants.

3.3.2 Splitting

A file is split in 1Mo chunks. For each chunk are generated an initialization vector (IV) and authentication data (AD), both are generated on 128 bits.

3.3.3 Encryption

For every chunk an encryption is performed with the IV and AD of the chunk itself and the $F EK$ of the file such as $Enc(F EK, Chunk, IV, AD, 128)$ 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 $chunk_{packet} = (AD || IV || encrypted_chunk)$.

3.3.5 Uploading process

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

3.4 Decryption

3.4.1 Extraction

The first chunk of the remote file is downloaded. We extract the IV , AD , and Salt from it.

3.4.2 Key derivation

The Salt is used to proceed to a key derivation, such as $KDF(CEK, Salt, 7000, 256)$, the output will be the $F EK$ of the file.

3.4.3 Decryption

The content of the first chunk (*DATA*) is decrypted with the extracted cryptographic parameters such as $Dec(FEK, DATA, IV, AD, 128)$. The other chunks are decrypted with their own extracted cryptographic parameters and the same *FEK* as the first chunk.

3.4.4 Reassembling

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

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.