AN END-TO-END ENCRYPTED PROTOCOL FOR CLOUD STORAGE SOLUTIONS

Paul Feuvraux
pfeuvraux@gmail.com

August 15, 2020

TO BE REVIEWED

Current cloud storage solutions don't provide enough details about their cryptographic protocols, which leads to wrong assumptions from the users. In this paper, we describe how to create a scalable protocol for anyone that would want to implement the end-to-end encryption within their cloud storage software.

This paper only describes the cryptographic protocol, and does not explain the security measures that are taken for the clients, the API, or even the infrastucture. Thus, passwords managements, or such issues unrelated to the protocol are not described within this paper.

1. Introduction

Privacy has always been put at risk, and the Internet was never designed with security in mind. The lack of security within the Internet design has made it easier for private companies, goverments, or other entities to spy on the people, and threathen the universal right to privacy. Even though platforms such as Dropbox take security measures to prevent unauthorized access to their users' data, the lack of end-to-end encryption leaves room for data leaks and data reselling from hackers and the data holders themselves. By having end-to-end encryption built-in within the platform, such security vulnerabilities are significantly reduced.

Current solutions tend to use too many primitives which are often inefficient and insecure. There appears to be an absence of end-to-end encrypted cloud storage platforms that are also open source, and thus, transparent. Our proposed cryptographic protocol is a mix of heavy security practices, and high-speed methods. We recommend this protocol to whoever wants to create an end-to-end cloud storage solution, or implement end-to-end encryption within their current software architecture.

In order to avoid breaking most of the common workflows, we decided to make this protocol compatible with the most common use cases. Hence, this protocol allows users to share files through a public link, or with their contacts in a secure way. Moreover, this protocol also considers *private realms*, so that even companies or any entity will be able to have control over access.

2. Network Model

Any communication protocol could be used as long as the code running on the client-side is hardly amendable such as compiled programs. Yet, as HTTPS is very common, secure, and a lot of libraries in many languages exist, we highly recommend it.

The client and server communicate between each other through HTTPS. Our HTTPS implementation involves the following components:

- **Protocol**: TLSv1.2 only
- **Encryption algorithm**: Chacha20-Poly1305 or AES-256-GCM-SHA384
- **RSA key size**: 4096 bits

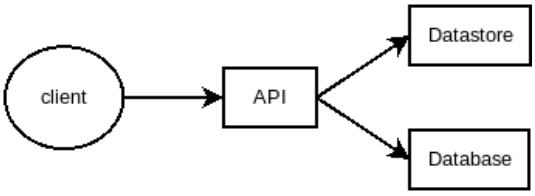

Figure 1: Network Model Overview

As decribed in figure 1, the API (the server) is connected to a datastore (hard drive, GlusterFS, Ceph, etc.), and a database. All communications are encrypted and an internal Certificate Authority is set up in order to limit the MITM attack within the internal network.

Only the files are stored within the datastore. The files and directories hierarchy is stored within the database along with the metadata. For the files and directories hierarchy, JanusGraph is used, for all the other data, we use ScyllaDB.

3. Protocol Functioning

3.1 Definitions

- Terms:
  - **Resource**: defines a file or directory.
  - **Realm**: defines a space such as an organization, a user's space, etc.
  - **Encryption Key**: generic name referring to a key used to encrypt a resource.
  - $chunk_x$: refers to any chunk composing a file.
- Primitives:
  - **Library**: Libsodium
  - **Key Derivation Function**: Argon2-3id
  - **Encryption Algorithms**: xChacha20-Poly1305 or AES-256-GCM (default)
  - **Asymmetric Algorithms**: Curve25519 (default) or RSA-4096
- Functions:
  - KDF(P, s): Argon2(P, salt, 7, 65536, 256), where:
    - P: Passphrase, can be a UTF-8 string or a binary-formatted key
    - salt: Salt, 128-bit binary array, randomly generated
    - 7: Iterations.
    - 65536: Bytes to be used by the Argon2 function
    - 256: Key size output
  - Dec/Enc(k, c, n, ad): Encryption / Decryption function
    - k: 256-bit array (binary formatted)
    - c: un/encrypted content
    - n: Initialization Vector, 128-bit binary array, randomly generated
    - ad: Additional Data: 128-bit binary array, randomly generated
  - rand(size, format): randomly generate data (binary by default). $format$ can be binary (0b), hexadecimal (0x), or UTF-8 encoded (utf8).
  - AsymEnc/AsymDec(k, c), where:
    - k: Public or Private Key
    - c: plaintext / ciphertext

3.2 Key Hierarchy

Users need to share their files and directories in different ways. In order to accomplish this, we define a key hierarchy as the following shows:

- **Content Encryption Key - CEK** : can be considered as a *Root Key*.
- **Directory Encryption Key - DEK**: a directory (folder) encryption key, used to derive the FEK and other DEK of a directory's children. Every directory has its own DEK.
- **File Encryption Key - FEK**: a file encryption key, used to encrypt the data and title of files. Each file has got one or multiple FEK depending on its size.
- **Parent Encryption Key - PEK**: as defined below, the *Parent Encryption Key* can either be the CEK or a DEK, depending on the depth. From depth 0, the PEK is the CEK. For depths $0 + n$ (where $n \geq 1$), the PEK will be a DEK.

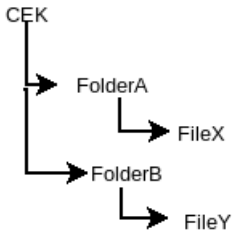

Figure 2: Key Hierarchy Overview

As described in figure 2, keys are set under a cascaded structure. As a result, in order to decrypt fileX, the client would need to decrypt the key of FolderA, and CEK. This structure allows users to share a whole folder with very few computations. The Content Encryption Key can be considered as a *Root Key*. Then, FolderA and Folder B have their own keys, derived from the CEK. Hence, FileX and FileY have their own *File Encryption Keys* respectively derived from FolderA and FolderB keys.

And so, we define three core keys and one relative key:

3.2.1 Example

Let's assume that C is a directory within B, and B is a directory within A. Now, we want to move C up to within A, above B. We define this procedure such as:

Demonstration:

$$\delta_e = Enc(C, \delta) \tag{1a}$$
$$C_e = Enc(B, C) \tag{1b}$$
$$C = Dec(B, C_e) \tag{1c}$$
$$C_e = Enc(A, C) \tag{1d}$$
$$C = Dec(A, C_e) \tag{1e}$$
$$\delta = Dec(C, \delta_e) \tag{1f}$$

Where C is the encryption key of the C directory containing files and directories, and δ is the *File Encryption Key* of a file δ within C directory.

3.3 Initialization

In this section, we describe how the protocol gets initialized during the registration procedure.

During registration, the client generates a CEK. Meanwhile, the user defines a passphrase P. We define two registration procedures:

- Two-Password: user enters a login password and passphrase
- One-Password: user only enters a login password

3.3.1 Two-Password Method

During a Two-Password registration, the user defines a *passphrase* named P being a string. In the meantime, the CEK is randomly generated over 256 bits. We describe the procedure such as:

1. $CEK = rand(256)$
2. $P = input(keyboard)$
3. $salt = rand(128)$
4. $KEK = KDF(P, salt)$
5. $IV = rand(128), AD = rand(128)$
6. $CEK_e = Enc(KEK, CEK, IV, AD)$

Once CEK is encrypted as in CEK_e, the client encapsulates the data such as: $packet = (CEK_e||salt||IV||AD)$.

3.3.2 One-Password Method

The following method allows the user to type only one password named PW. We describe the procedure such as:

1. $LoginPassword = input(keyboard)$
2. $salt_l = rand(128)$
3. $LoginPassword = KDF(PW, salt_l)$
4. Client: $SRP(LoginPassword, salt_l)$
5. $CEK = rand(256)$
6. $salt_p = rand(128)$
7. $KEK = KDF(PW, salt_p)$
8. $IV = rand(128), AD = rand(128)$
9. $CEK_e = Enc(KEK, CEK, IV, AD)$
10. $packet = (CEK_e||salt_p||IV||AD)$

Afterwards, *packet* is sent to the server in order to store it so that the user can access their resources from anywhere.

3.4 Key Decryption

When a user aims to log in, they're asked their password or passphrase depending on which password method has been chosen by the user. In the case the user has chosen the Two-Password method, this procedure is to be followed:

1. User enters their passphrase P
2. Client gets $packet = (CEK_e||salt||IV||AD)$
3. $KEK = KDF(P, salt)$
4. $CEK = Dec(KEK, CEK_e, IV, AD)$

The One-Password method requires more steps, and is described as follows:

1. User enters their password PW
2. Client gets $salt_l$
3. $LoginPassword = KDF(PW, salt_l)$
4. Client: $SRP(LoginPassword, salt_l)$ for authentication purposes
5. Once authenticated, Client gets $packet = (CEK_e||salt_p||IV||AD)$ from the server
6. $KEK = KDF(PW, salt_p)$
7. $CEK = Dec(KEK, CEK_e)$

Now, the user can encrypt and decrypt their files.

4. File Handling

In this section, we describe in details how the files are encrypted and decrypted, and how the titles of files and directories are encrypted and decrypted as well.

4.1 File Encryption

To begin with, the title of the file is encrypted as follows:

1. $IV_t = rand(128), AD_t = rand(128)$
2. $salt_1 = rand(128)$
3. $FEK_1 = KDF(PEK, salt_1)$
4. $title_e = Enc(FEK_1, title, IV_t, AD_t)$

Then, we send the encrypted title for file upload initialization as:

$$packet_t = (title_e, salt_1, IV_t, AD_T)$$

Next, the file is split into chunks in order to parallelize the encryption and upload. For instance, a file of 600KiB, we split the file such as:

id	chunk	size (KiB)
0	$chunk0$	512
1	$chunk1$	88

The first chunk will use FEK_1 which is used for title encryption right above. By using the same FEK for title encryption and first chunk encryption, we save time during the process. For every next chunk are randomly generated a 128-bit array IV and AD.

1. $salt_x = rand(128)$
2. $FEK = KDF(PEK, salt)$, where PEK is the parent key.
3. $IV = rand(128), AD = rand(128)$
4. $encChunk_x = Enc(FEK_x, chunk_x, IV_x, AD_x)$
5. $title_e = Enc(FEK, title, IV_t, AD_t)$, where IV_t and AD_t are randomly generated.

PEK meaning *Parent Encryption Key* can either be a DEK or can be the CEK in this case of file encryption, depending on the depth of the file, as explained above: if the file is at depth-0 (*root*), then the CEK will be used for the key derivation function. Otherwise, the DEK of the current directory will be used.

For every encrypted chunk, the client encapsulates the encrypted content and the cryptographic parameters such as:

$$packet_x = (encChunk_x || salt_x || IV_x || AD_x || IV_t || AD_t).$$

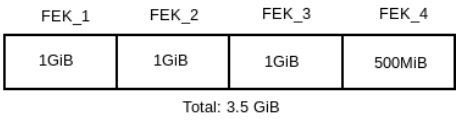

Figure 3: FEK Rotation Overview

As figure 3 describes, once a FEK has encrypted over 1GiB, we proceed to a key rotation by doing a key derivation of the parent key of the FEK, as: $FEK_x = KDF(PEK, salt)$, where $salt = rand(128)$ and PEK is the parent key of the parent directory. Consequently, a file's size between 1GiB and 2GiB will have two File Encryption Keys: (FEK_1, FEK_2), where $FEK_1 = FEK$ in this section, and so on.

4.2 File Decryption

We describe the file decryption procedure as follows:

1. $salt_1$ is obtained from the server for the first File Encryption Key (FEK_1) of the file.
2. $FEK_1 = KDF(PEK, salt_1)$
3. $chunk_x = Dec(FEK_1, encChunk_x, IV_x, AD_x)$, where IV_x and AD_x are both obtained from the server for every chunk.

If the file is larger than 1GiB, then the File Encryption Key will be FEK_x, where x depends on the count of GiB that has been decrypted.

4.3 Directory Encryption

Each directory has its title getting encrypted when being created. We describe this procedure as follows:

1. $DEK = KDF(PEK, salt)$, where *salt* is randomly generated over 128 bits, and DEK stands for *Directory Encryption Key*.
2. $IV = rand(128), AD = rand(128)$
3. $dirTitle_e = Enc(DEK, dirTitle, IV, AD)$

Once the title of the directory is encrypted, the client sends a packet to the server, defined as:

$$packet = (dirTitle_e, salt, IV, AD)$$

4.4 Directory Decryption

We define the decryption procedure such as:

1. $DEK = KDF(PEK, salt)$, where *salt* is obtained from the server along with IV and AD.
2. $dirTitle = Dec(DEK, dirTitle_e, IV, AD)$

Once decrypted, the title is displayed on the client interface.

4.5 Sharing

We define two types of sharing procedures:

- **External Sharing**: share a file or directory through a public link and secret password.
- **Internal Sharing**: share a file between *contacts*

External sharing allows a user to share either a file or directory through a public link and a defined password. We describe this procedure as:

1. $EK = KDF(PEK, salt_e)$, where EK means *Encryption Key* and can either be a FEK or a DEK depending on the type of resource to be shared (file / directory), and $salt_e$ is obtained from the server.
2. $DK = KDF(SP, salt_d)$, where SP is the entered secret passphrase for the resource to be shared, and $salt_d$ is randomly generated over 128 bits.
3. $EK_e = Enc(DK, EK, IV, AD)$, where IV and AD are randomly generated over 128 bits.
4. Client sends $packet = (EK_e||salt_d||IV||AD)$.

Now, anyone with the generated public link and password can access the resource, such as:

1. Client gets $packet = (EK_e||salt_d||IV||AD)$ from Server.
2. User enters SP
3. $DK = KDF(SP, salt_d)$

4. $EK = Dec(DK, EK_e, IV, AD)$
5. $chunk_x = Dec(EK, encChunk_x, IV_x, AD_x)$

Users should be able to share resources between each other in a secure and automated way. We introduce the notion of Contacts later. As the Contacts notion is yet not defined, we assume that the Public Key of a user's contact is safely encrypted and stored by the server. In order to use it, the client needs to get it from the server and decrypt it.

And so, we define the Internal Sharing procedure as follows:

1. Client gets user's contact's Encrypted Public Key (PKB_e).
2. $PKB = Dec(CEK, PKB_e, IV, AD)$, where IV and AD are obtained from the server.
3. $EK_e = AsymEnc(PKB, EK)$

From that point on, the user's contact is now able to decrypt the Encryption Key of the resource (file or directory), and access it, such as:

1. Client gets EK_e, and we assume that the user's private key PKA has already been decrypted.
2. $EK = Dec(PKA, EK_e)$
3. $chunk_x = Dec(EK, encChunk_x, IV_x, AD_x)$

From that point on, the user's contact will be able to download and decrypt every chunk of the file.

4.6 Comments on Files & Directories

Users are able to comment on files and directories, and react to each other's comments the way people react on Slack, Discord, or Facebook. When sharing a resource, the user's client derives the first FEK of the file into a *Social Encryption Key - SEK*, which will be used to encrypt comments and reactions. We describe this procedure as follows:

1. $salt = rand(128)$
2. $SEK = KDF(FEK_1, salt)$
3. $IV_x = rand(128), AD_x = rand(128)$
4. $encComm_x = Enc(SEK, comm_x, IV_x, AD_x)$
5. $packet = (encComm||salt||IV_x||AD_x)$
6. $send(packet)$

As described right above, an IV and AD are generated over 128 bits for every comment. Now, anyone having access to the resource will be able to decrypt FEK_1, and hence will be able to derive it into the SEK with the salt. We describe the procedure such as:

1. Client decrypts FEK_1 as explained in the above sections.
2. $SEK = KDF(FEK_1, salt)$
3. $comm_x = Dec(SEK, encComm_x, IV_x, AD_x)$

Multiple comments can be retrieved from the server and decrypted in the meantime in order to display them faster on the user interface.

4.7 Device Syncing

All user's clients should be able to sync each other without interaction, and using the end-to-end encryption in order for users to access their files natively from any device through the native file explorer of the operating system.

In order to save bandwidth, devices do not sync between each other by exchanging the entirety of the modified files. However, differential deltas will be encrypted and then broadcasted to all the devices. Differential deltas will be stored temporary, so if a device is out of network for a time range longer than the differential delta is supposed to be kept on the servers, the device will need

to download the whole file once it gets connected back. **VCDIFF** is the format that we will use in order to manage the differential deltas and apply them on the other devices.

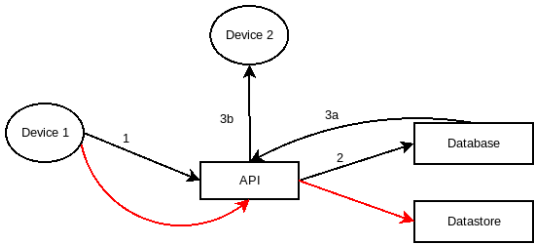

Figure 4: Syncing Model Overview

In red is the whole file being encrypted, uploaded, and then stored within the datastore. This ensures that the user will still have access to the file even through the web. Black arrows describe the steps of the data differencing system. Device 1 computes the differential delta, encrypts it, and sends it to the API server, which will store it within the database, and device 2 will be able to retrieve the delta later on. However, if device 2 is connected to the Internet and is ready to sync, then the API server will send the differential delta right away in order for device 2 to sync it without delay. This system allows our infrastructure to be less used, as no READ request is required from the API server in the case device 2 is ready to sync.

5. Organization Features

5.1 Groups and Communities

5.1.1 Group - Type 1

Group T-1 is just an alias to group the number of actions that a user needs to complete in order to share a resource with their contacts. In other terms, group T-1 is a way for the user to only tick contacts they want to share resources with, without having to manually enter each contacts. Consequently, a user creating a group T-1 only needs to enter their contacts once. In the future, the user will just need to enter the group they want to share resources with.

5.1.2 Group - Type 2

Group T-2 can also be called as *Communities*. A community is a group of users allocating their storage to a group as in a community.

The *Creator* of a community, while creating it, has their client generate a *Group Encryption Key - GEK*. Once the GEK generated, it is shared with contacts that are to be part of the community. We describe the procedure as follows:

1. $GEK = rand(256)$
2. *Creator's* client encrypts the GEK:

$$(GEK_e = AsymEnc(PKB_x, GEK)) \vee (GEK_e = Enc(KDF(P, salt_p), GEK)))$$

,where PKB_x is a key belonging to one of the Creator's contacts, and is decrypted as explained in the sections above. P represents the passphrase entered by the Creator in the case the Community is to be public.

As GEK is to be encrypted for every user entering the community, either under the users' public keys, or under a passphrase, a community remains a safe place for users to share resources without compromising their privacy. GEK can be considered as the *Root Key* of the community. When a user from within the community wants to upload a file, this procedure is to be applied:

1. $salt = rand(128)$
2. $FEK = KDF(GEK, salt)$
3. $IV = rand(128), AD = rand(128)$

4. $encChunk = Enc(FEK, chunk, IV, AD)$
5. $packet = (encChunk, salt, IV, AD)$

As described right above, once the file is encrypted, it is encapsulated in a packet containing the cryptographic parameters allowing the other users to recompute the FEK and decrypt the file.

5.2 Contacts

Users are able to add contacts in a secure way. As described above, users can share resources between each other allowing them to securely share resources without being victim of a *Man-In-The-Middle* attack. In order to use the contacts feature, users' clients need to generate a pair of asymmetric keys. We define the following procedure as:

1. $(PKA, PKB) = AsymGen(128, 25519)$
2. $DK_a = KDF(CEK, salt_a)$, where $salt$ is randomly generated over 128 bits.
3. Client generates IV_a and AD_a, both over 128 bits.
4. $PKA_e = Enc(CEK, PKA, IV_a, AD_a)$
5. $DK_b = KDF(CEK, salt_b)$, where $salt_b$ is randomly generated over 128 bits.
6. Client generates IV_b and AD_b randomly, over 128 bits.
7. $PKB_e = Enc(CEK, PKB, IV_b, AD_b)$

Afterwards, the client encapsulates the cryptographic parameters along with the keys such as:

$$packet = (PKB||PKB_e||PKA_e||IV_a||AD_a||IV_b||AD_b||salt_a||salt_b)$$

The client sends *packet* to the server in order to store it.
When users have their pair of asymmetric keys generated and stored encrypted, they can make a contact request with other users. Let's assume Alice adding Bob as a contact, we describe the following procedure such as:

1. Alice's client gets PKB, being Bob's public key.
2. Alice's client generates IV and AD, both randomly over 128 bits.
3. $PKB_e = Enc(CEK, PKB, IV, AD)$, Alice's client encrypts Bob's Public Key.

Alice now has Bob's Public Key. Consequently, Bob can now accept Alice's request and execute the same procedure. Once Bob is done, they both get able to compare the fingerprint of each other's Public Key in order to verify whether or not they have been victim of a *MITM* attack. Every time Alice and Bob want to share a resource between each other, they need to get each other's encrypted Public Key and share the Encryption Key of the resource, such as:

1. Client gets $packet = (PKB_e, salt, IV, AD)$
2. $DK = KDF(CEK, salt)$
3. $PKB = Dec(DK, PKB_e, IV, AD)$
4. $EK_e = AsymEnc(PKB, EK)$

In order for Alice to verify Bob's fingerprint, Bob shows her a QRCode containing the fingerprint. Alice scans the QRCode with her phone, while her phone will fetch Bob's PKB_e, decrypt it, compute its fingerprint, and compare it with the fingerprint from the QRCode. Furthermore, the contact request can be realized through NFC, which guarantees that the Public Keys have not been compromised.

5.3 Private Realms

Some users would want to create a *private realm* (an organization), and control users. In order to provide such a feature, we define roles, permissions, a key hierarchy, and the cryptography behind this feature. We define several types of roles:

- **Owner**: has full rights on the private realm
- **Master**: has full rights but cannot act upon the Owner
- **Admin**: can manage members
- **Member**: two types of members:
  - Private Member: admins do not have access to the member's keys.
  - Non-Private Member: admins have access to the member's keys

During the creation of a private realm, the *Owner* generates a pair of asymmetric keys, (PKA_o, PKB_o). Those will be used for the admins, master, and the owner to decrypt the files of non-private members. Their pair of asymmetric keys of the private realm are encrypted under the Owner's CEK.

5.3.1 Admins Creation

To begin with, any admin has to be a private member and must have access to the private realm's keys. Consequently, we define the following steps in order for the admin to obtain the keys:

1. Master: creates an admin, an invitation link or QRCode is generated.
2. Admin: clicks on the link or scan the QRCode and generates the keys.
3. Admin: $(PKA_a, PKB_a) = AsymGen(128, 25519)$
4. Admin: encrypts (PKA_a, PKB_a) under their CEK.
5. $PK_o = (PKA_o, PKB_o)$
6. Master: $PK_e = AsymEnc(PKB_a, PK_o)$
7. Master: $send(PK_e)$ to the server.

5.3.2 Non-Private Member Creation

When an admin creates a non-private user, the keys for this user are generated by the admin. We describe the procedure as follows:

1. Client gets encrypted PKB_o of the private realm.
2. Client gets admin's PKA and PKB (PKA_a, PKB_a) and decrypt them.
3. $PKB_o = AsymDec(PKA_a, encPKB_o)$
4. $CEK_u = Gen(256)$
5. $(PKA_u, PKB_u) = AsymGen(128, 25519)$
6. $CEK_e = AsymEnc(PKB_o, CEK_u)$
7. $PK_u = (PKA_u, PKB_u)$
8. $PK_e = AsymEnc(PKB_o, PK_u)$

Once CEK_u and PK_u being the non-private member's keys are encrypted, and then sent to the server in order to be stored along with their cryptographic parameters. In the meantime, CEK_u and PK_u are encrypted under a passphrase with the same procedure as described in section *External Sharing*, where the passphrase is to be given to the user through NFC, QR code, e-mail, etc.

5.3.3 Decrypting a non-private member's resource

Admins are able to decrypt non-private member's resources, we define the procedure such as:

1. Client gets private realm's PKA_e and decrypts it.
2. $DK = KDF(CEK, salt)$, where $salt$ is obtained along with PKA_e from the server.
3. $PKA = Dec(DK, PKA_e, IV, AD)$, where IV and AD are obtained from the server as well.
4. Client gets $encCEK_u$, which is the non-private member's CEK
5. $CEK_u = AsymDec(PKA, encCEK_u)$
6. $chunk_x = Dec(CEK_u, encChunk_x, IV_x, AD_x)$

As described above, Admins are able to decrypt non-private members' resources.

5.4 Guests

Guests can be created by any normal user. Although, this feature may be blocked by admins within an private realm (organization). When a user creates a, several options are proposed for the creator to transmit the Guest Account creation page:

- **QRCode**: Guest User scan the QRCode which will redirect them to the Guest Account creation page.
- **Link**: Guest User just needs to click on the link in order to be redirected.

Now, when the user accesses the Guest Account Creation page, they can choose several options, as described below:

- Passphrase: Guest User enters a passphrase. We define the procedure as:
  1. $(PKA, PKB) = AsymGen(128, 25519)$
  2. $IV_a = rand(128), AD_a = rand(128)$
  3. $DK_a = KDF(passphrase, salt_a)$, where $salt_a$ is randomly generated.
  4. $PKA_e = Enc(DK_a, PKA, IV_a, AD_a)$
  5. $IV_b = rand(128), AD = rand(128)$
  6. $DK_b = KDF(passphrase, salt_b)$, where $salt_b$ is randomly generated.
  7. $PKB_e = Enc(DK_b, PKB, IV_b, AD_b)$
- QRCode: Keys will be stored within the QRCode along with a link for the Guest User to access the resources. We define the procedure as:
  1. $(PKA, PKB) = AsymGen(128, 25519)$
  2. $DK = rand(256)$
  3. As describe above, PKA, PKB are encrypted under DK.
  4. Once encrypted and stored on the server, DK is converted into a QRCode along with the link of the Guest User realm.
- File: A file is to be saved by the Guest User, we define the following procedure as:
  1. $DK = rand(256)$
  2. As describe above, PKA, PKB are generated and encrypted under DK.
  3. DK is saved into a file.
  4. The file is saved by the Guest User, every time they'll want to access their realm, they'll just have to select this file in order to decrypt the content they're accessing.

Once PKB and PKA encrypted, Client sends the following:

$$packet = (PKB||PKB_e||PKA_e||salt_a||salt_b||IV_a||AD_a||IV_b||AD_b)$$

The User encrypts PKB, which is the Public Key of the Guest User. We define the procedure as:

1. $salt = rand(128)$
2. $IV = rand(128), AD = rand(128)$
3. $DK = KDF(CEK, salt)$
4. $PKB_e = Enc(DK, PKB, IV, AD)$

Public Key could be spoofed, it is then recommended that both Guest User and User verify each other's fingerprints.

Users can share resources with the Guests they created. We define the sharing procedure as:

1. $DK = KDF(CEK, salt)$, where $salt$ is obtained from the server.
2. $PKB = Dec(DK, PKB_e, IV, AD)$
3. $FEK_e = AsymEnc(PKB, FEK)$, where FEK is to be decrypted as described in the above sections.

6. Evaluation

We evaluate the security and usability of a perfect implementation of this protocol.

6.1 Architecture Security Evaluation

The primitives that are used in this protocol are known to be the strongest and safest around. Furthermore, the CEK is needed for the user in order for them to access their files. An adversary cannot access the CEK as it is encrypted before even reaching the server. Files, directories, and their titles are encrypted, which means that any adversary having successfully penetrated a storage server could at most read the *Access Time, Creation Time* and *Modification Time* of a file. This could eventually be improved over the future releases.

6.2 Usability Evaluation

Since the protocol aims to achieve an end-to-end encryption model, some features might not be available such as searching the files by names as titles are encrypted. Consequently, some workflows may need some modification from the users' part for them to exploit the cloud storage software at its fullest.

6.3 Scalability

This protocol is easily scalable, from one user to thousands. However, the contact feature makes it hard to scale. Since the owner of a resource has to encrypt the *Encryption Key* of the resource for every contact, having many contacts to share a resource with will increase the number of computations ($O(n)$, where n is the number of contacts to share the document with). However, as Curve25519 is the default curve for asymmetric keys and operations, this statement is not likely to be true unless a huge number of contacts are involved.

In regards to the implementations, web being a very portable platform, any JavaScript implementation of this protocol will be slower than a native implementation due to the single-threading architecture of JavaScript. Therefore, this protocol will be slow on non-multithreaded languages. In order to counter this issue, we recommend developers to parallelize some parts of the protocol, such as the encryption of the chunks that can be performed simultaneously.

6.4 Attack Vectors

In this section, we discuss some of the most potential attack vectors that could break the protocol.

6.4.1 Web implementations

As for every protocol, the implementation of it plays a big role in the security degree that can be achieved. Web can be weak, especially when it comes to a very bad *HTTPS* implementation. Hence, a bad HTTPS implementation would most likely lead to an undesired change in the JavaScript files during the client-server exchanges, resulting to a security breach breaking the entirety of the protocol.

6.4.2 Fingerprints

As described in the Contacts section, it is necessary for users to check each other's Public Key fingerprint in order to avoid being victim of a MITM attack. If not checked properly, the attacker could decrypt every single file that both parts shared with each other.

6.4.3 Memory-based Attacks

In order to limit attacks based on the memory, we recommend developers to shift all the bits to zero from the beginning to the end of the chain composing a private parameter, such as a chunk, or a key once the client does not need to use it anymore.

6.4.4 Brute-force

In this section, we describe two types of brute-force vectors:

- Brute-force on entered private strings such as passwords or passphrases.
- Brute-force on the encrypted content.

As required since we want users to access the resources from anywhere, the *salt* is not private so that the keys can be recomputed. Consequently, an attacker proceeding to a brute-force solution against a passphrase is likely to be a success if users do not define strong passphrases. Although, as this protocol uses Argon2 with strong parameters, brute-forcing a passphrase remains difficult. Furthermore, as AES-256-GCM and xChacha20-Poly1305 are both considered as the strongest primitives for encryption, we consider that any resource as practically undecipherable.

6.4.5 OOB Communication Channel

External Sharing involves an Out-of-Band communication channel such as e-mails, instant messaging, etc. In order for the users to communicate the secret password of an externally shared resource, users should use an end-to-end compliant channel. Otherwise, in the case of sending plaintext e-mails or plaintext instant messges, the secret password of the externally shared resoure could be intercepted by the provider of this OOB communication channel.

6.5 Zero-Knowledge Proof

In order to prove that our system complies with the end-to-end encryption principle, we define a procedure to follow in order to run tests against the protocol.

6.5.1 Key Derivation Test

We define a passphrase P, and a salt S, and a derived key K, where $K = KDF(P, S)$.

In order to test the safety of the key derivation process, we generate P and S randomly, and iterate 10,000 times (this number can be increased). For every iteration, the generated derived key is compared with K. At the end of the 10,000 iterations, we enter P and S into the key derivation function. In normal circumstances, K should equal the generated derived key of the 10,001 iteration.

6.5.2 Encryption Test

We proceed to the same method, but now the parameters are:

- K, the encryption key.
- IV, an initialization vector (128 bits).
- AD, some additional data (128 bits).

For 10,000 iterations, a key K_g is generated over 256 bits and is used to try to decrypt the encrypted resource. At the 10,001 iteration, we use K instead of K_g, which should result to a successful decryption.

6.5.3 Safe Transit Test

We monitor the exchange between Client and Server over HTTP without SSL/TLS. Client encrypts a file and send the chunks in parallel to Server. Our monitoring system has the right key K and the right cryptographic parameters of every chunk of every file being IV and AD since both of those parameters are not private. Client encrypts and uploads 10,000 files. At the 10,001 file, our monitor uses K to decrypt a chunk of the last file, which should result to a successful decryption. Consequently, this shows that as long as the key K is not leaked, the protocol remains safe.

6.6 Security Improvements

It is not necessary for contacts to sign off each other's Public Key's fingerprint. And so, in the case of a MITM attack, the protocol could be broken. We could indeed wait for a manual action from the user in order for them to sign off a fingerprint. Although, in order for our protocol not to be too strict when it comes down to the user experience, we only decided to ask the users to make sure they manually verify each other's fingerprint.

7. Conclusion

Our protocol has been proven to be secure and compliant with the principle of *End-to-End Encryption*. Even though end-to-end encryption is to be respected, the protocol does not break most of the common workflows. This protocol uses a lot of heavy computations, especially for the key derivation function that is based on Argon2. Consequently, systems with very low memory or a full-javacript implementation may end up running out of resources, or slowed down during the process of deriving keys. We recommend to tweak the Argon2 parameters in order to adapt the protocol for very resource-limited environments, and find a good trade-off between security and speed.

8. Acknowledgements

Thanks to Hoang Long Nguyen for editorial feedback.