VonaDrive - Encrypted cloud storage

Paul Feuvraux

March 14, 2019

DRAFT

1 Introduction

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

VonaDrive aims to protect users' online privacy through *End-to-End Encryption* against malicious parties. VonaDrive 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 $AsymDec(privkey, content)$.
- **Encapsulation**: Often is this document, "packet" is mentioned, a packet is a UTF-8 string which may be represented this way:

  $packet = "\$var0" + " : " + "\$var1"$, where + represents the concatenation.

3 Protocol

3.1 Registration

Users must register with a VonaDrive 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.

While the user submits his credentials, VonaDrive 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: $CEK_c = Enc(KEK, CEK, IV AD)$, 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.

$packet_k$ is a JSON document, such as:

```
{
        "iv": "IV"
        "v" : 1,
        "iter" : 7000,
        "ks" : 256,
        "ts" : 128,
        "mode" : "gcm",
        "adata" : "AD",
        "cipher" : "aes",
        "salt" : "saltFEK",
        "ct" : "CEKc"
}
```

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 VonaDrive 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. VonaDrive 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.

For instance, if we take a 600MiB file, we have got:

id	chunk	size
0	$chunk0$	512
1	$chunk1$	88

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 $pck = (encchk_x||salt_{fek}||IV_x||AD_x)$.

pck is a UTF-8 string, we represent it such as:

$pck = "\$encchk_x" + " : " + "\$salt_{fek}" + " : " + "\$IV_x" + " : " + "\$AD_x"$,
where $\$x$ is a variable which represents a base64-encoded bit array.

3.3.5 Title encryption

The title of the file is encrypted under the FEK. The client generates an IV and AD, two different 128-bit arrays randomly generated.

Once the IV and AD generated, the client encrypts the file such as: $title_e = Enc(FEK, title, IV, AD, 128)$. $title_e$ is then sent to the server and stored into the database as a packet: $pckTitle = (title_e||IV||AD)$.

3.3.6 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 the cryptographic parameters into a table, such as:

id	content
0	$encchk_x$
1	$salt_{fek}$
2	IV_x
3	AD_x

3.4.2 Key derivation

The $salt_{fek}$ is used to proceed to a key derivation, such as $FEK = KDF(CEK, salt_{fek}, 7000, 256)$, where $salt_{fek}$ is base64-decoded before getting used for the computation.

3.4.3 Decryption

For every received chunk, the client decrypts it such as: $chk_x = Dec(FEK, encchk_x, IV_x, AD_x, 128)$, where $encchk_x$, IV_x, and AD_x are base64-decoded before getting used for the computation.

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.

3.4.5 Title Decryption

The client gets $pckTitle$ and decrypts it such as: $title = Dec(FEK, title_e, IV, AD, 128)$.

3.5 File sharing

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

3.5.1 External file sharing

Encryption. The user chose to share their file with another VonaDrive 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 VonaDrive).

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 pick the contacts they want to share the file with. The client gets the encrypted PKB stored as a packet and extract all the element of it, such as: s, IV, AD, PKB_e. The client then proceeds to a key derivation such as: $DK = KDF(CEK, s, 7000, 256)$. Once DK computed, the client decrypts PKB_e such as: $PKB = Dec(DK, PKB_e, IV, AD)$.

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 $FEK_c = AsymEnc(PKB, FEK)$, 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.

3.6 Folder Sharing

Due to the current structure of this crypto-system, a non-empty folder cannot be shared. Hence, the user will only be able to share a folder if and only if the folder is empty.

3.6.1 External Folder Sharing

When a folder is shared, a REK (Directory Encryption Key) is generated over 256 bits. Once generated, the client encrypts it under the passphrase defined by the user while externally sharing the folder. In order to achieve the encryption, the client proceeds to a key derivation of the passphrase P, such as: $DK = KDF(P, s, 7000, 256)$, where s is a 128-bit array randomly generated. Then, the client encrypts REK such as: $REK_e = Enc(DK, REK, IV, AD, 128)$, where IV and AD were randomly generated over 128 bits. Once REK is encrypted, the client sends it to the server as a packet, such as: $packet = (REK_e||s||IV||AD)$.

For the owner, to be able to upload the file from his account directly, without using the public link, the REK is encrypted under his derived CEK, such as: $DK = KDF(CEK, s, 7000, 256)$, and $REK_e = Enc(DK, REK, IV, AD, 128)$. Once encrypted under the owner's derived CEK, REK_e is sent to the servers as a packet, such as: $packet = (REK_e||s||IV||AD)$.

Uploads by the owner When the owner, through their account, upload a file within this shared folder, it gets REK_e which was encrypted their derived CEK. The client firstly proceeds to a key derivation, such as: $DK = KDF(CEK, s, 7000, 256)$, and then decrypts REK_e such as: $REK = Dec(DK, REK_e, IV, AD, 128)$.

Once REK obtained, the client will encrypt every file with a derived REK.

Upload by the public link When the users browse the publick link, they are asked to submit the passphrase for decrypting REK_e. The user enters the passphrase P, then the client derives it such as: $DK = KDF(P, s, 7000, 256)$, once the passphrase is derived, the client decrypts REK_e such as: $REK = Dec(P, REK_e, IV, AD, 128)$.

Once REK is obtained, for every file, the client derives REK and encrypts the chunks with the derived REK.

3.6.2 Internal Folder Sharing

When a folder is shared, a REK (Directory Encryption Key) is generated over 256 bits. Then, the client encrypts it under the users' PKB such as: $REK_e = AsymEnc(PKB, REK)$.

For the owner to be able to upload files within this internally shared folder through their own account, the client encrypts REK under the owner's derived CEK. The client derives the CEK, such as: $DK = KDF(CEK, s, 7000, 256)$, where s is a 128-bit randomly generated array. Once DK obtained, the client encrypts it such as: $REK_e = Enc(DK, REK, IV, AD, 128)$, where IV and AD are two randomly generated bit arrays. Then, the client sends REK_e as a packet such as: $packet = (REK_e||s||IV||AD)$.

Uploads by the owner When the owner wants to upload a file within this shared folder, the client derives the CEK such as: $DK = KDF(CEK, s, 7000, 256)$. Then, the client decrypts REK_e such as: $REK = Dec(DK, REK_e, IV, AD, 128)$.

Then, for every file the user wants to upload within the shared folder, the client will derive REK and encrypt the file under the derived REK.

Upload by the shared-with contacts The client gets REK_e and decrypts it such as: $REK = AsymDec(PKA, REK_e)$.

For every file that the shared-with contacts want to upload within this shared folder the client derives REK and encrypts it with the derived REK.

3.7 Contacts

In order to be able to add a contact or be added as a contact, the user needs to generate a pair of asymmetric keys.

3.7.1 Asymmetric generation

First of all, the user generate at the client-side a pair of asymmetric keys, such as: $(PKA, PKB) = AsymGen(RSA, l)$, where PKA is the private asymmetric key, PKB the public asymmetric key, RSA is the asymmetric algorithm used, and l is the final length of the keys, which will results either 2048 or 4096 bits.

The paire of keys is encrypted with a derived CEK. The client randomly generates a salt s, IV, and AD. The CEK is derived such as $DK = KDf(CEK, s, 7000, 256)$. Then, the client encrypts PKA such as: $PKA_e = Enc(DK, PKA, IV, AD)$, and makes it a packet before sending it to the server, such as: $packet = (PKA_e||s||IV||AD)$. The client will encrypt and make a packet of PKB in the same way.

The unencrypted PKB is kept and sent to the server in order to store it unencrypted so that the user may be added as a contact by other users.

3.7.2 Adding a contact

In order to add a contact, the client gets the public key PKB of the user they want to add.

Once the client gets PKB, it computes the fingerprint of it and displays it so that the being-added user and the user can make a comparison of the two fingerprints and confirm that PKB is not compromised.

As preamble, a salt s, IV and AD are generated over 128 bits. The client proceeds to a key derivation of the CEK such as: $DK = KDF(CEK, s, 7000, 256)$. Afterwards, it encrypts PKB such as: $PKB_e = Enc(DK, PKB, IV, AD)$. Finally, the client sends PKB_e to the server such as: $packet = (PKB_e||s||IV||AD)$.

3.7.3 Fingerprint comparison

Users may want to make a comparison of the fingerprints of their public keys in order to confirm that the keys are the right ones, and are not compromised.

Before the client displays the fingerprint, it gets the encrypted PKB of the user and compute its fingerprint.

3.8 Comments on Files

Users will be able to comment on files. Either one internally shared files, or externally shared files.

3.8.1 Allowing comments - Internally Shared File

First of all, when the user allows comments on an internally shared file, the client randomly generates a symmetric key LEK (Comments Encryption Key) over 256 bits. Once generated, the client generates an IV and AD over 128 bits and encrypts it such as: $LEK_e = Enc(CEK, LEK, IV, AD, 128)$, and send it to the server as a packet, such as: $pck = (LEK_e||IV||AD)$.

In the case the file is already externally shared and the comments are enabled, the client gets LEK_e and decrypts it under the user's CEK in order to proceed to the next step.

The LEK is encrypted under the users' PKB, such as: $LEK_e = AsymEnc(PKB_x, LEK)$, where PKB_x is the Public Key of the user(s) that the user wants to internally share the file with, and LEK is the master key of every comment made for this file.

The client sends LEK_e for every encryption made by PK_x to the server along with the AD and IV.

3.8.2 Adding a comment - Internally Shared File

When the users want to add a comment to the file, their client gets LEK_e and decrypts it such as: $LEK = AsymDec(PKA_x, LEK_e)$, where PKA_x represents the PKA of the user that wants to add a comment.

Once LEK decrypted, an IV and AD are randomly generated over 128 bits. Then, the client encrypts the comment ct such as: $ct_e = Enc(LEK, ct, IV, AD, 128)$. Finally, the client makes it a packet before sending it to the server, $packet = (ct_e||IV||AD)$.

3.8.3 Reading Comments - Internally Shared File

The client gets LEK_e and decrypts it such as: $LEK = AsymDec(PKA, LEK_e)$. Once decrypted the client gets every packet, and for each of them, the client extracts ct_e, IV, and AD.

Once the data extracted, for each packet, the client decrypts the comments such as: $ct_x = Dec(LEK, ct_e, IV, AD)$, where ct_x is one of the comments to be decrypted. Then, the client does that for every comment.

3.8.4 Allowing Comments - Externally Shared File

The client checks if the file is already internally shared and if the comments are enabled. If so, then the client gets LEK_e, and decrypts it such as: $LEK = Dec(CEK, LEK, IV, AD, 128)$.

In the case the file isn't shared already, the client randomly generates a LEK over 256 bits and encrypts it under the user's CEK. An IV and AD are generated over 128 bits, then the client encrypts it such as: $LEK_e = Enc(CEK, LEK, IV, AD, 128)$.

Once LEK obtained, the client encrypts it under the passphrase that the user will define for externally sharing it.

Adding and reading comments will be done the same way as described in the "Internally Shared" part above.

3.9 Organizations Feature

3.9.1 Keys management.

When an organization is created, a CEK is generated, we call it CEK_o.

Then, CEK_o is encrypted under the master's PKB, where PKB is encrypted and defined as PKB_e.

The client needs to decrypt the user's PKB_e. In order to proceed, the client derives the master's CEK such as: $DK = KDF(CEK, s, 7000, 256)$. Once derived, the client decrypts the master's PKB_e under DK, such as: $PKB = Dec(DK, PKB_e, IV, AD, 128)$.

Then, CEK_o is encrypted under PKB, such as: $CEK_oe = AsymEnc(PKB, CEK_o)$ and then is sent to the server.

3.9.2 Creation of a private 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, their client will have to generate a CEK, and the user will 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), and their client will generate a pair of asymmetric keys (PKA,PKB) as defined in the **Contacts** section.

3.9.3 Creation of a non-private sub-user.

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

Then, the client generates a salt s, an IV and AD over 128 bits. Afterwards, the client derives CEK_o such as: $DK = KDF(CEK_o, s, 7000, 128)$ in order to encrypt it under the CEK of the organization.

Afterwards, CEK_u is encrypted under DK, such as: $CEK_ue = Emc(DK, CEK_ue, IV, AD, 128)$, and then sent to the server.

The client generates a salt s, an IV and AD over 128 bits. And derives the passphrase P such as: $KEK = KDF(P, s, 7000, 128)$.

Once KEK is computed from P, the client encrypts the sub-user's CEK under the KEK such as: $CEK_ue = Enc(KEK, CEK_u, IV, AD, 128)$, and sends it to the server.

The client generates a pair of asymmetric keys (PKA, PKB). Once generated, the client generates a salt s, an IV and AD over 128 bits.

Afterwards, the client derives CEK_o such as: $DK = KDF(CEK_o, s, 7000, 128)$. Once DK computed, the client encrypts CEK_u under DK such as: $CEK_ue = Enc(DK, CEK_u, IV, AD, 128)$ and sends it to the server and do exactly the same for PKB as defined in the **Contacts** section.

Next, the client encrypts (PKA, PKB) under CEK_u. The client generates a salt s, an IV and AD over 128 bits. Once those parameters generated, the client derives CEK_u such as: $DK = KDF(CEK_u, s, 7000, 128)$. Then, the client encrypts PKA under DK such as: $PKA_e = Enc(DK, PKA, IV, AD, 128)$ and sends it to the servers. The client does the same for PKB.

3.9.4 Making a non-private user private

The keys of the non-private sub-user encrypted under CEK_o are deleted from the server and the status of the sub-user is changed to private.

3.9.5 Promoting a sub-user as administrator.

Every master or administrator is be able to either create or delete a user.v A non-private sub-user cannot be promoted as admin.

The client gets PKB_e of the user that will be admin. Then, the client derives the user's CEK such as: $DK = KDF(CEK, s, 7000, 256)$. Afterwards, the client decrypts PKB_e such as: $PKB = Dec(DK, PKB_e, IV, AD, 128)$.

Once PKB decrypted, the client encrypts CEK_o such as $CEK_oe = AsymEnc(PKB, CEK_o)$ and sends it to the server.

3.9.6 Administrator logs in as a non-private user.

The administrator's client gets PKA_e and derives CEK such as: $DK = KDF(CEK, s, 7000, 256)$. Once DK obtained, the client decrypts PKB_e such as: $PKB = Dec(DK, PKA_e, IV, AD, 128)$.

Once PKA_e decrypted, the client gets CEK_oe and decrypts it such as: $CEK_o = AsymDec(PKA, CEK_oe)$. Then, the client gets CEK_ue and derives CEK_o such as: $DK = KDF(CEK_o, s, 7000, 256)$. Once DK obtained, CEK_ue is decrypted such as: $CEK_u = Dec(DK, CEK_ue, IV, AD, 128)$.

Finally, the admin is able to decrypts the files of the non-private user with CEK_u.

3.10 Groups

3.10.1 Group - Type 1

When a user wants to share a file with several contacts, they need to tick all the corresponding boxes to their contacts. In order to counter that, a user can create groups that will tick all the checkboxes for them.

3.10.2 Groupe - Type 2

Groups Type 2 allow the users to share folders, files, and storage space.

Creating a Group In order to create a group, the master's client (creator/owner of the group) generates a Group Encryption Key GEK over 256 bits and encrypt it under the master's Public Key PKB such as: $GEK_e = AsymEnc(PKB, GEK)$.

Introducing users into the Group When the master or administrators of the group want to add members, their client gets the member's Public Key and encrypts GEK under their Public Key such as: $GEK_e = AsymEnc(PKB, GEK)$.

Structure

Keys Management

- **GEK**: Group Encryption Key (256 bits). This Key is used for Key Derivation to create DEKs.
- **DEK**: Directory Encryption Key (256 bits). This Key is used for Key derivation to create FEKs (256 bits).
- **FEK**: File Encryption Key.

File Encryption Key In the case the user is at the root of the group, their client will derive GEK such as: $FEK = KDF(GEK, Gen(s, 128), 7000, 256)$, where $Gen(s, 128)$ is a salt s as a bit-array generated over 128 bits.

In the case the user is inside a folder, the same process happens, but DEK is used instead of GEK.

Directory Encryption Key In the case the user is at the root of the group, their client will derive GEK such as: $DEK = KDF(GEK, Gen(s, 128), 7000, 256)$, where $Gen(s, 128)$ is a salt s as a bit-array generated over 128 bits.

In the case the user is inside a folder, the same process happens, but the DEK of the parent folder is used instead of GEK.

Externally Sharing a File The process is exactly the same as described within the **External File Sharing** section.

Externally Sharing a Folder In the case the user wants to share a folder located at the root of the group, then they are asked to enter a passphrase P. Then, the client derives P such as: $DK = KDF(P, Gen(s, 128), 7000, 256)$, where $Gen(s, 128)$ is a salt s as a bit-array generated over 128 bits.

Then, the client encrypts GEK under DK such as: $GEK_e = Enc(DK, GEK, Gen(IV, 128), Gen(AD, 128)$ and sends the pck to the server such as: $pck = (GEK_e||s||IV||AD)$.

4 Conclusion

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

5 Weaknesses

Every system is vulnerable to some attacks, in this section we explain how weak can our system be.

5.1 Key Management

Storing the Keys at the Client-Side. Keys absolutely must not be stored in a permanent store (hard drive, USB key, etc.). By doing so, the possibility of having the user's keys extracted by an attacker increases.

We recommend the developers to store the keys on the memory only, and delete them from it during the program activity if they are not needed anymore for the future actions.

Storing the Keys Remotely. Users need to access their files wherever they are, whatever their device is. Hence, the server needs to store the keys for future use.

The master key which is defined as KEK here, is never sent to the server. Instead, we send the crypto-parameters that will help the client to recompute it later: salt, initialization vector, etc. THe passphrase used for recomputing the master key is also never sent to the server.

In regards to the other keys, such as the encryption key, it is encrypted under its parents, which can be the master key, or another key, and is then sent to the server. For instance in our system, the File Encryption Key is encrypted under the Content Encryption Key and then sent to the server along with some relevant crypto-parameters. The Content Encryption Key is encrypted under the master key (KEK) and then sent to the server.

We recommend the developers to be very careful in sending the keys to the server. For every change in their code related to our system, they always should track their network activity and change whether the sent keys are encrypted. Moreover, developers should always think twice before sending a key despite being encrypted: maybe sending it is no use for the future.

Contact's Keys Users can add contacts and their public keys will be exchanged. In such situation, both ends should take care of verifying the fingerprints of each other's public key in order to avoid any MITM attack over time.

5.2 Server-Side Security

Web Applications. Our biggest issue is the web applications. JavaScript security can be very weak sometimes, and MITM attacks are very easy on poorly configured TLS implementations at both side, client and server.

In the case the web server is compromised, the JavaScript code might be compromised as well, and the users' keys may have been stolen by the attacker over time.

Our system encrypts the keys, hence, the attacker needs to modify the JavaScript, and wait for the users to reload the web application. Hence, in the case the web server is compromised, developers should make sure to stop it right away, and spin up another instance of that web server, from scratch.

6 Notes // @TODO

- **Key derivation algorithm**: it exists several improved key derivation algorithms, but we've chosen to use PBKDF2-HMAC-SHA256 for its performances and safety. We are going to migrate to Argon2.
- **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.
- **Englishify**: some sentences are way too much turned in a French way, a user can't be inside a folder.

7 Thanks to

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