Thalos: a secure approach to file storage in cloud

Luca Maria Castiglione
luc.castiglione@studenti.unina.it

July 2017

1 Introduction

Nowadays it is rapidly increasing the need to store our files in remote to keep them easily available on multiple devices. Normally people do not have their own storage servers so they need to rely on someone who keep their files for them, that's why services like Google Drive or Dropbox rapidly grew up in the technology market.

At this point, we need to ask ourselves how much secure are this kind of services and what would happen to our files if someone seized storage servers or hacked into them. And, at least, how much it is really possible to trust in those companies?

This project would propose a solution to this kind of problems: Thalos is storage service completely robust and secure by design.

The chosen cryptographic algorithms and the way they are applied offer to the final user the opportunity to securely store his files remotely, denying any attempt of access them without the proper authorization. Thalos design, indeed, makes it impossible for anyone who has physical or virtual access to the servers to decrypt files without the right key and neither to establish an exact match between one specific file and its owner.

Thalos relies on local elaborations to perform encryption: everything outside user laptop is hard encrypted with asymmetric algorithms (AES 4096 bit key) according to OpenPGP standards [1]. Due to the most known critical issues that belong to read and write operations, in fact, cryptography is executed locally to the user machine. About that, Thalos absolutely do NOT memorize keys or passphrases in cookies or, more general, anywhere in the browser.

2 Concept

Thalos shows up as a service that can be easily used, in theory, by any device connected to the internet. People could easily register an account using their email address and choosing an username and a password; sessions keep track of the users across the application. Once a user is registered, a first key pair can be generated; calculated keys are:

- **Master Key** : The private key of the pair, it belongs to the user that can unlock it trough a passphrase chosen during the creation process. It's highly recommended to choose a passphrase different from the account passoword.
- **Public key** : As can be guessed by its name, this is the public key of the pair, it is stored on a remote database. It could be also used for secure file sharing in future improvements.

Once a key pair is generated it is possible to add a **basket** to your own basket list. **Baskets** are virtual file containers (they can be thought as very simple virtual filesystems), each basket is described by a **basket description file** which basically stores information about contained files including name, type, size and *a pointer to the encrypted file on disk* (attribute id) as it can be seen in Figure 2. Among with the basket, two new keys are generated, we are talking about:

- **Basket Private Key**: Used to decode the basket description and each file which belongs to the basket itself.
- **Basket Public Key**: Used to encode the basket description and each file which belongs to the basket itself.

Basket description files are stored remotely encrypted with the basket private key.

Furthermore, a **basefile** is associated to each user, it is remotely stored encrypted with the Master Key of the user to whom it belongs. A basefile contains the **basket private keys** of the baskets owned by the user it is associated with, as pointed out in Figure 1

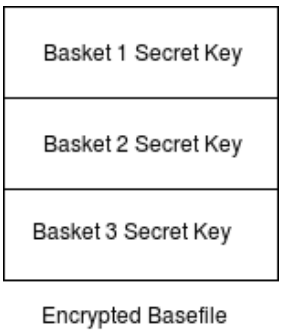


Figure 1: Basefile

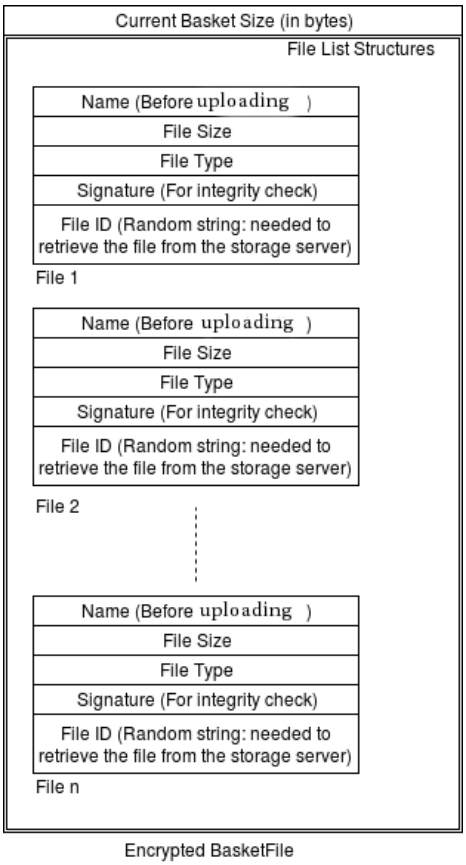


Figure 2: Basket Description

File upload In Figure 3 and 4 it is showed how Alice and Bob can securely store their files using Thalos.

In Figure 3

1. Alice retrieves from remote her basefile which is encrypted with her Master Key Pair.
2. Alice decrypts locally to her laptop the basefile and gets the keys needed to unlock her baskets.
3. Alice retrieves from remote the encrypted description of the "UAV" basket.
4. Alice decrypts the description and gets the entire file list which included pointers to real file on disk.
5. Now Alice can securely download any files she wants to.

Furthermore in Figure 4

1. Bob retrieves from remote his basefile which is encrypted with his Master Key Pair.
2. Bob decrypts locally to his laptop the basefile and gets the keys needed to unlock her baskets.
3. Bob retrieves from remote the encrypted description of the the basket that contains his deppest secrets.
4. Bob decrypts the description locally to his laptop and gets the entire file list which included pointers to real file on disk.
5. Now Bob can securely add a fresh secret to its list, no one will ever heard it from Thalos.

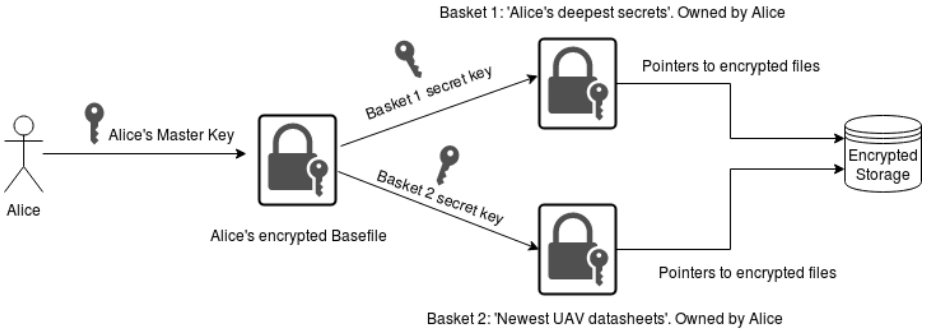


Figure 3: Alice's path

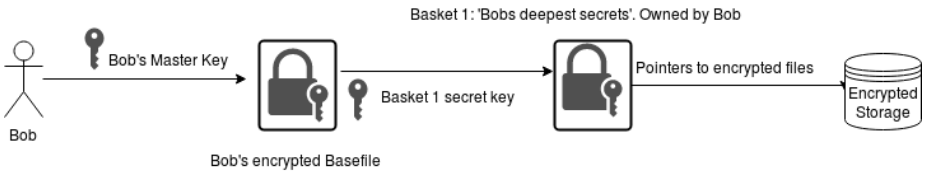


Figure 4: Bob's Path

Alice's and Bob's files are stored (encrypted) on the same hard drive among with other users files. It is impossible to find a reverse path which leads from a file to its owner.

3 Beyond the theory: Thalos software description

In practice, Thalos shows up as a Web Application that can be reached through any modern Internet browser (it has been successfully tested on Firefox and Google Chrome) and allows users to create an account, generate a master key pair and, eventually, securely manage their files. Those functions are described in the *Use Case Diagram* in Figure 5.

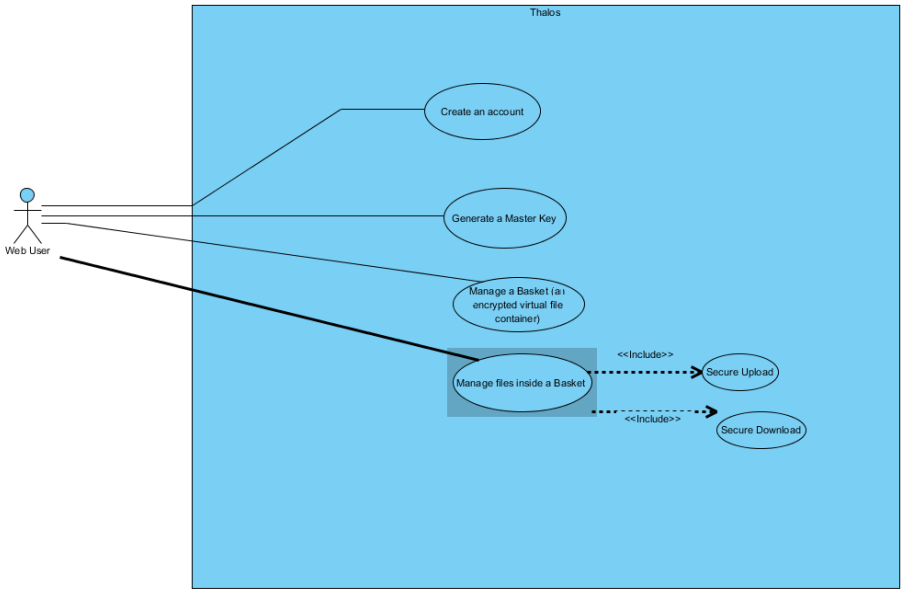


Figure 5: Thalos Use Case Diagram

More in details Thalos has been developed following a *Client-Server* pattern where the client role is played by a Web Browser.

3.1 Server side architecture

Since the project runs on NodeJS [3], server routines are programmed in server-side Javascript. Furthermore, the server has been designed following a *Model View Controller* paradigm as showed by the architectural view in Figure 6
The application uses **PUG** [4] as view engine to dynamically render HTML pages and, eventually, **SEQUELIZE** [6] as ORM tool to dynamically map views inside a relational database and managing migrations.

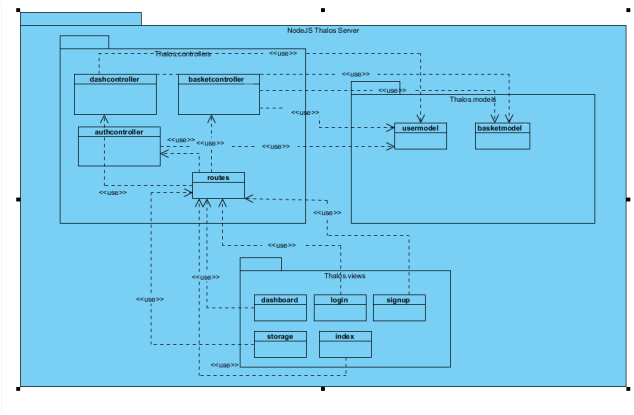


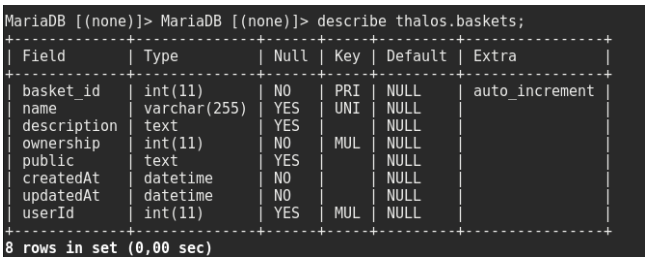
Figure 6: Thalos Atchitecture

Models decription Two models are used in this application, they are 'user' and 'basket'. As it can be easily guessed by their name, the first one is needed to manage users and the other to manage baskets. Tables content is described in Figures 7 and 8. Particular fields are

- **users.public_key** : stores the user public key (from master key pair).
- **users.base** : stores the encrypted basefile associated to the user.
- **baskets.description** : stores the encrypted basket description.

```
MariaDB [(none)]> describe thalos.users;
+------------------+--------------------------+------+-----+----------+----------------+
| Field            | Type                     | Null | Key | Default  | Extra          |
+------------------+--------------------------+------+-----+----------+----------------+
| id               | int(11)                  | NO   | PRI | NULL     | auto_increment |
| username         | text                     | YES  |     | NULL     |                |
| public_key       | text                     | YES  |     | NULL     |                |
| email            | varchar(255)             | YES  | UNI | NULL     |                |
| password         | varchar(255)             | NO   |     | NULL     |                |
| last_login       | datetime                 | YES  |     | NULL     |                |
| basketcount      | int(11)                  | YES  |     | 0        |                |
| status           | enum('active','inactive') | YES  |     | inactive |                |
| activation_token | varchar(255)             | YES  |     | NULL     |                |
| base             | text                     | YES  |     | NULL     |                |
| createdAt        | datetime                 | NO   |     | NULL     |                |
| updatedAt        | datetime                 | NO   |     | NULL     |                |
+------------------+--------------------------+------+-----+----------+----------------+
12 rows in set (0,00 sec)
```

Figure 7: User Table



```
MariaDB [(none)]> MariaDB [(none)]> describe thalos.baskets;
+-------------+-------------+------+-----+---------+----------------+
| Field       | Type        | Null | Key | Default | Extra          |
+-------------+-------------+------+-----+---------+----------------+
| basket_id   | int(11)     | NO   | PRI | NULL    | auto_increment |
| name        | varchar(255)| YES  | UNI | NULL    |                |
| description | text        | YES  |     | NULL    |                |
| ownership   | int(11)     | NO   | MUL | NULL    |                |
| public      | text        | YES  |     | NULL    |                |
| createdAt   | datetime    | NO   |     | NULL    |                |
| updatedAt   | datetime    | NO   |     | NULL    |                |
| userId      | int(11)     | YES  | MUL | NULL    |                |
+-------------+-------------+------+-----+---------+----------------+
8 rows in set (0,00 sec)
```

Figure 8: Basket Table

Views description Views are written according to **PUG** syntax [4]. **PUG** engines dynamically renders HTML pages. Informations from controllers are sent to the view as messages through **flash** [7]

Controller description The **express** [5] framework has been used with NodeJS in order to manage HTTP REQUEST. Express manages incoming connections trough the use of *routes*, when an HTTP REQUEST is incoming routes calls the associated callback, if it exists. Following controllers work on Thalos:

- **passportController** : Defines strategies for user login and registration.
- **dashController** : Manages operations on user dashboards, it allows users to upload and download keys, basefiles and to create baskets. It exposes the following interfaces
  - *addBasket*: reponds on POST requests. Retrieve user data from the current session and update the basket tables with POST parameters received among with the request. Returns a JSON with the result.
  - *addPublicKey*: reponds on POST requests. Retrieves user data from the current session and update the users table with the new public key and the new basefile both received from POST parameters. Returns a JSON with the result.
  - *getBasefile*: reponds on POST requests. Retrieves user data from the current session. Returns a JSON with the result.
- **basketController** : Manages operations on basket as description download and upload, and file download and upload. This controller exposes the following interfaces
  - *getFile*: reponds on POST requests. Returns file selected by file id Returns a JSON with the result.
  - *updateBasket*: Retrieve user data from the current session and update the basket tables with POST parameters received among with the request. Returns a JSON with the result.

  - *deleteBasket*: reponds on POST requests. Delete a basket. Returns a JSON with the result.
  - *getBasket*: reponds on POST requests. Retrieve user data from the current session. Returns a JSON with the result.

- **authController** : Manages user authorizations and exposes the following interfaces
  - *login*: reponds on GET requests and commands the pug engine to show the login page.
  - *signup*: reponds on GET requests and commands the pug engine to show the signup page.
  - *validateUser*: change the user status from 'inactive' to 'active', this allows the user to login. It's a kind of antispam.

Controllers do not implement or call any kind of encryption algorithm since the data they work with are already encoded.

3.2 Client side architecture

In order to execute all encryption operations locally to the user machine, particularly in the user browser, the client side part of the project has been written entirely in *Javascript*. About this, the client side routines requires OpenPGP.js library [2] to perferm their duty. The code is divided in three main categories, according to the functions that are carried out.

- **Operations on dashboard**:
  - *genkey*: given a passphrase generates a keypair. The public key is sent to the server trough AJAX as user public_key. The private one is the user Master Key, a downlodable file is generated on the fly and a link is displayed.
  - *addBasket*: given the Master Key, the Master Key passphrase and a basket passphrase, it generates a new basket for the user who requested it. Eventually the function updates the basefile and sent it among with the new basket data to the server through AJAX.

- **Operations on baskets**:
  - *bloadlist* : Given a user, it makes an *XMLHttpRequest* to the server asking for the basefile. Eventually decrypts the basefile and displays the user basket list
  - *openbasket*: Given a user and a basket name, this functions make an *XMLHttpRequest* to the server asking for the description file of the basket. Once got it, it decrypts it and displays it to the user.

- **Operations on files**:

- *Upload*: Chosen a file and a basket. This function locally loads the file from a HTML form, save its info in a JSON object, encrypts the file, push the new JSON into the basket description array and encrypts the description as well. The file and the updated description are sent to the server trough the remote interface *UpdateBasket*.
- *Download*: Chosen a basket and a file to download. This method retrieves file info from the basket description and then asks for the file to the server through the file id. The server responds, the client decrypt it and generate a downloadable file on the fly.
- *Delete*: Chosen a basket and a file to delete. This method updates the current basket description deleting the selected file from its fid. The updated basket description is sent to the server among with the query needed to remove the file from the store server as well.

Furthermore, the following functions encoded in ***libthalos.js*** shows how informations are received trough an asyncrhonous call from the server and THEN locally decoded.

Listing 1: getBasketList

```
function getBasketList(keyfile, passphrase, callback) {


  var xhr2 = new XMLHttpRequest();

  xhr2.open("POST", '/user/getbasefile', true);
  xhr2.setRequestHeader("Content-type", "application/x-www-form-urlencoded");

  xhr2.onreadystatechange = function() {
    if (xhr2.readyState==4 && xhr2.status==200) {


      var encryptedbasefile = atob(JSON.parse(xhr2.responseText).basefile);
      var user_public = atob(JSON.parse(xhr2.responseText).public);
      var bcount = JSON.parse(xhr2.responseText).bcount;
      var userid = JSON.parse(xhr2.responseText).userid;



      if(keyfile) {
        var reader = new FileReader();
        reader.readAsText(keyfile, "UTF-8");

        reader.onload = function(e) {
          var privatekey = atob(e.target.result);

          var privKeyObj = openpgp.key.readArmored(privatekey).keys[0];
```

```
            privKeyObj.decrypt(passphrase);

            opts = {
              message: openpgp.message.readArmored(encryptedbasefile),
              publicKeys: openpgp.key.readArmored(user_public).keys,
              privateKey: privKeyObj
            };

            openpgp.decrypt(opts).then(function(plaintext) {


               callback(plaintext.data, privKeyObj);
             });
          }
        }
      } else {

      }
    }
    xhr2.send();


}
```

Listing 2: getFileList

```
function getFileList(keyfile, passphrase, basket_name, bpassphrase, callback) {
  var name = basket_name;
  getBasketList(keyfile, passphrase, function(data, privKeyObj) {
    var basketlist = JSON.parse(data);
    var index = -1;
    var mybasket = basketlist.baskets.find(function(item, i){
                if(item.name === name){
                  index = i;
                  return i;
                }
              });

    var basket_private = atob(basketlist.baskets[index].key);

    var xxhttp = new XMLHttpRequest();

    xxhttp.open("POST", '/user/getbasket', true);
    xxhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded")

    xxhttp.onreadystatechange = function() {
      if (xxhttp.readyState==4 && xxhttp.status==200) {
```

```
            var encryptedcontent = atob((JSON.parse(xxhttp.responseText)).descriptio
            var basket_id = JSON.parse(xxhttp.responseText).basket_id;
            var basket_public = atob(JSON.parse(xxhttp.responseText).public);

            var privKeyObj4 = openpgp.key.readArmored(basket_private).keys[0];
            privKeyObj4.decrypt(bpassphrase);

            bopts = {
              message : openpgp.message.readArmored(encryptedcontent),
              publicKeys: openpgp.key.readArmored(basket_public).keys,
              privateKey: privKeyObj4
            }

            var tmp = openpgp.decrypt(bopts).then(function(plaintext) {

            callback(plaintext.data, privKeyObj, privKeyObj4, basket_public);


            });

          }
        }
        xxhttp.send('bid='+name);


    });
}
```

4 Dynamics Views

In this section some of the actions described in the previous section are showed trough sequence diagrams. This section aims to give the reader a more clearer vision of the whole projects pointing out how client and server work together.

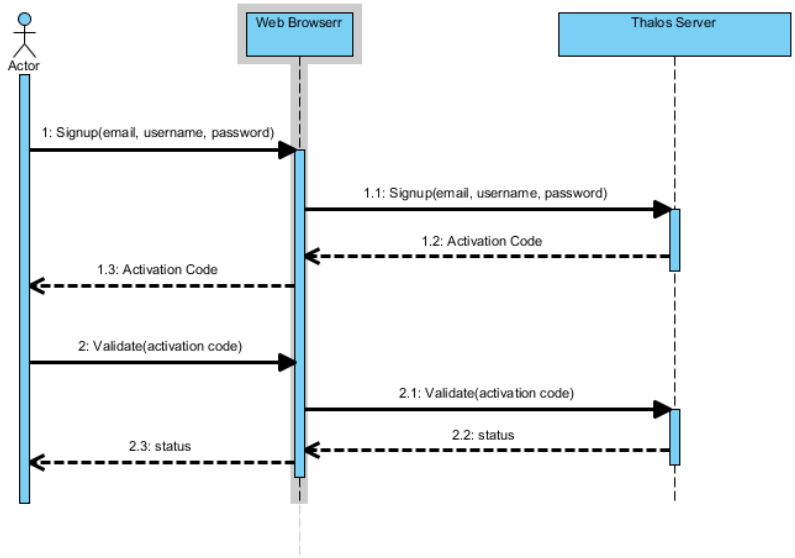


Figure 9: Sequence: User registration

From the following diagrams is clear that all private keys and passphrase are kept safe in the client.

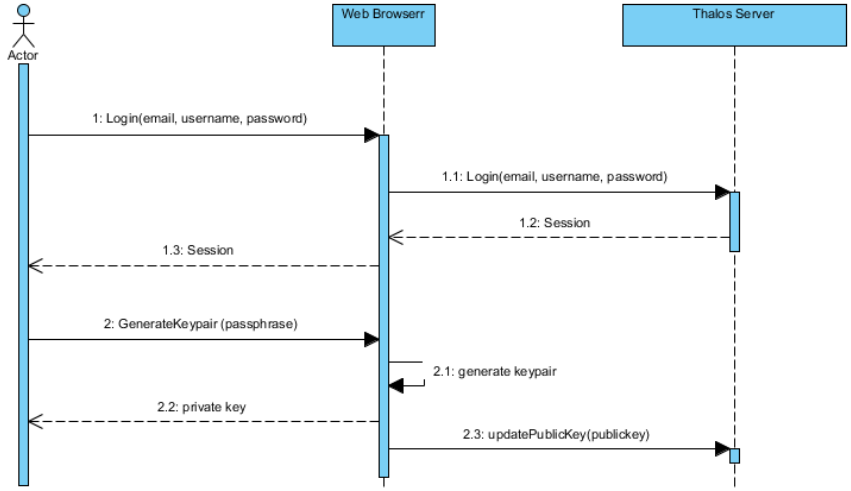


Figure 10: Sequence: Generate a Master Key Pair

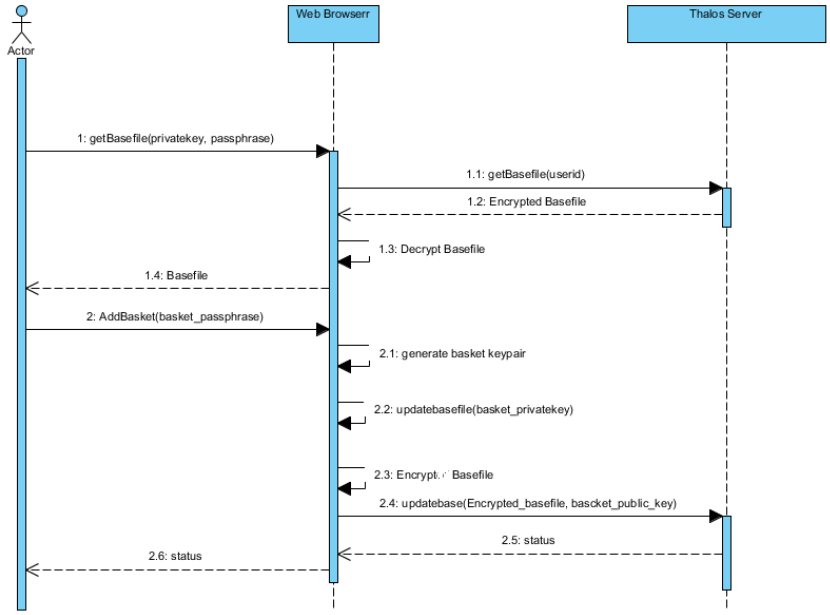


Figure 11: Sequence: Basket Creation

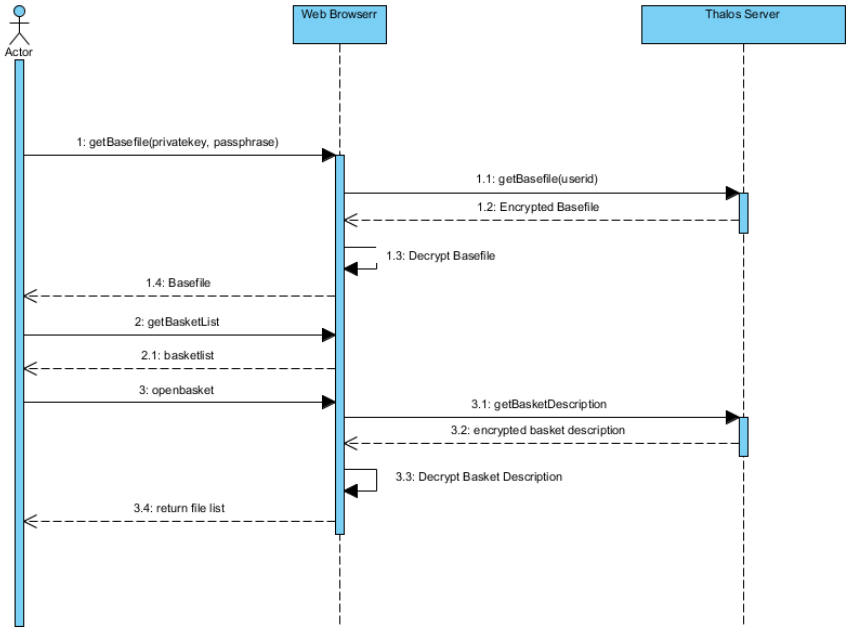


Figure 12: Sequence: Retrieve Basket list

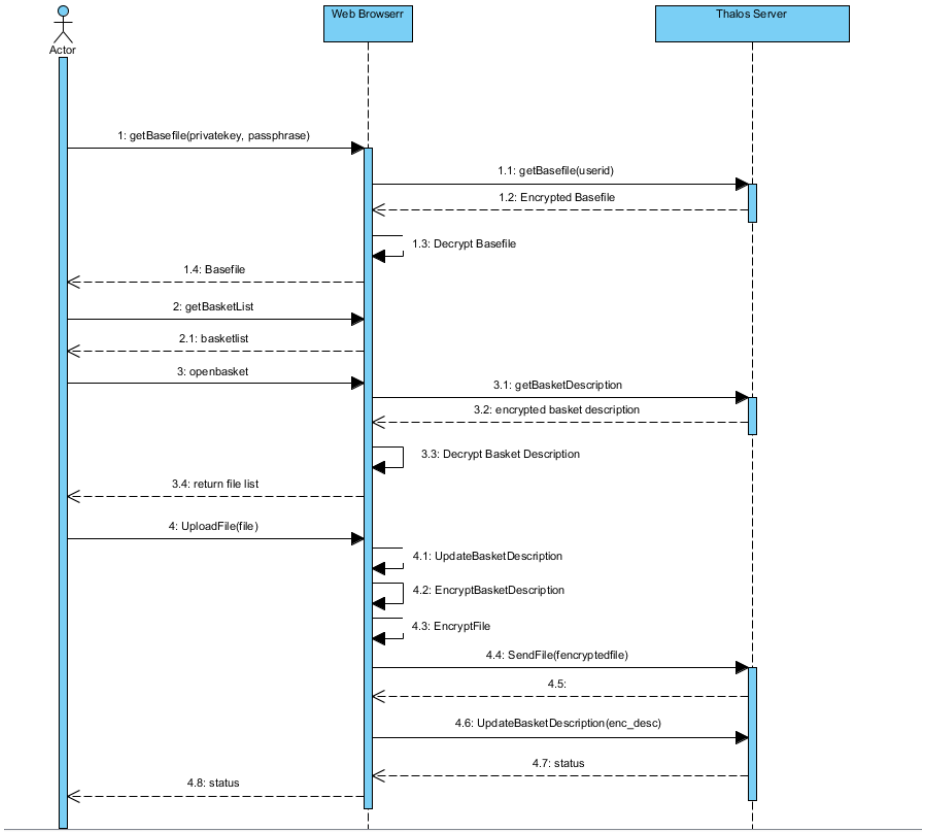


Figure 13: Sequence: File Upload

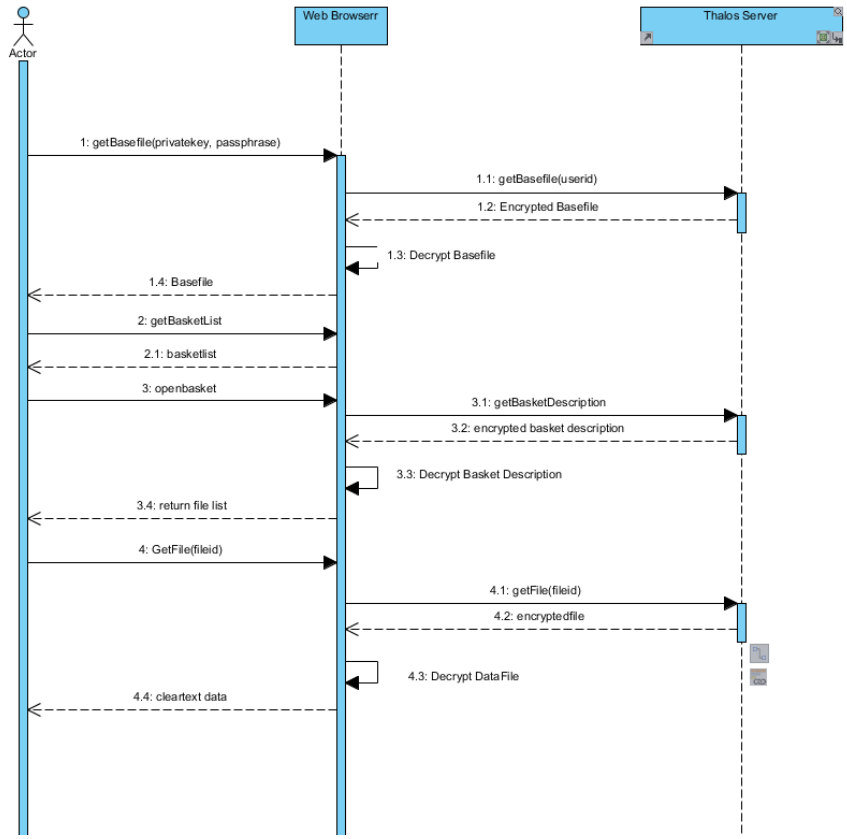


Figure 14: Sequence: File Download

5 The external perspective

In this last section we try to follow the breadcrumbs left by a user who use Thalos to securely store his files to show to the reader how the system works in terms of files and data stored on the server. As it can be seen in Figure 15 the actor creates an account in the webapp and validates it.

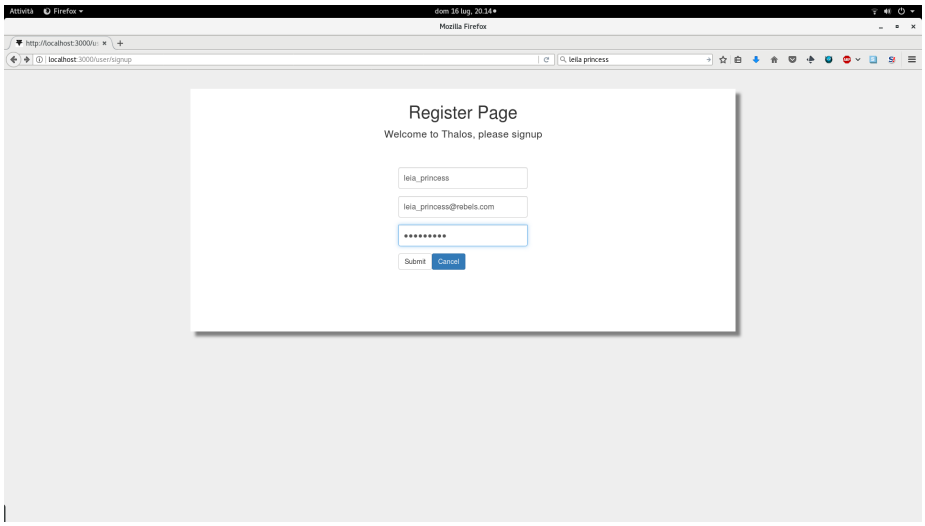


Figure 15: Account Registration

Our special guest signs into the dashboard and generates hers master keypair by clicking on *Generate Master Key* button (Figure 16).

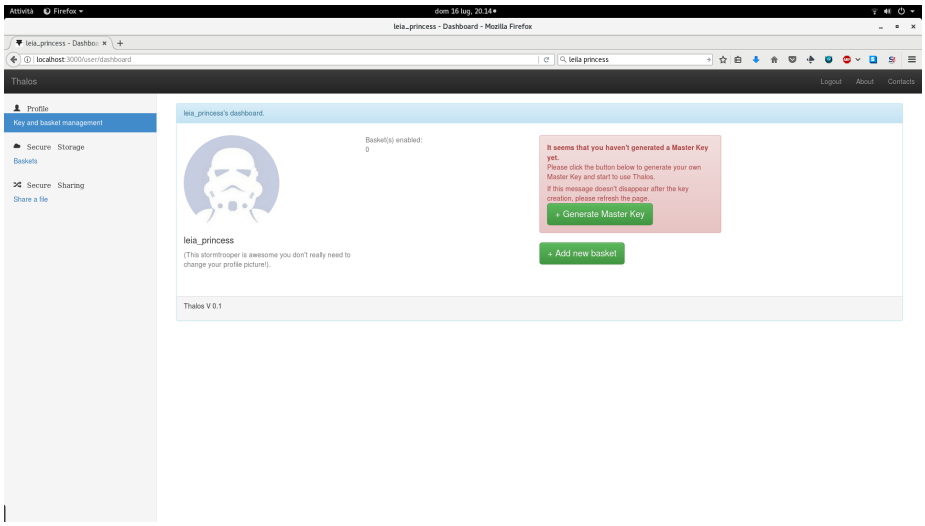


Figure 16: Dashboard

Using the newly calculated keypair she added a basket to hers basket list following the displayed instructions and, eventually, uploads the confidential informations (Figure 17).

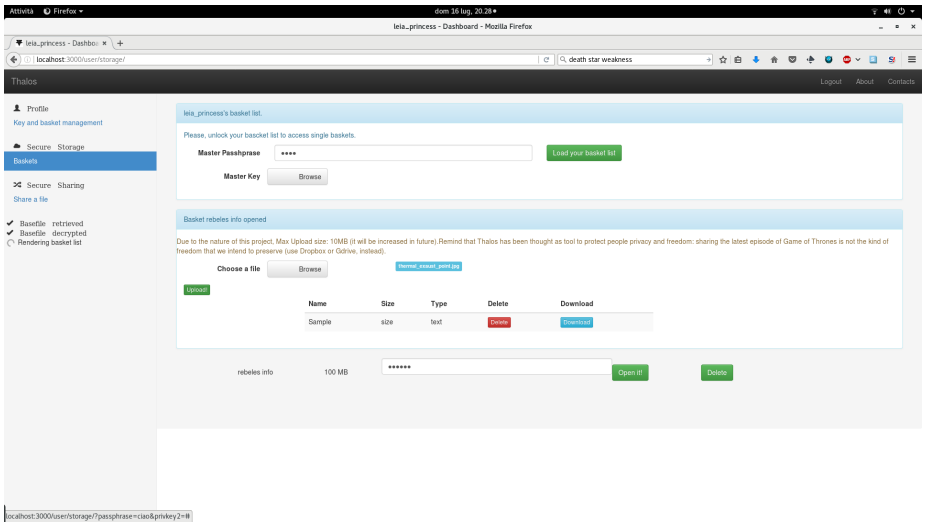


Figure 17: Informations Uploading

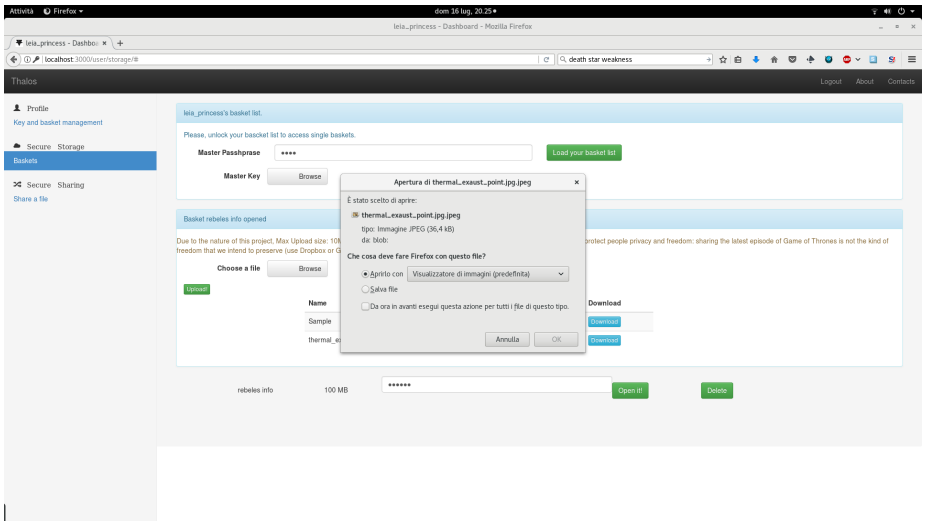
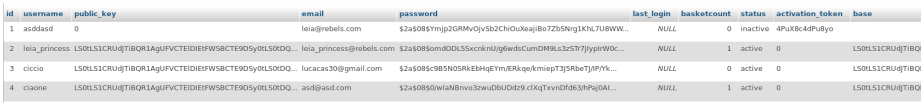


Figure 18: Data can easily be downloaded by clicking the download button

At this time the file is on the storage and it is ready to be downloaded by its owner whenever it comes the need 18.

Now assume that a bad guy, in some way, got access to the database, what can he actually learn about our user?

In Figure 19 it is shown a view of the database.



id	username	public_key	email	password	last_login	basketcount	status	activation_token	base
1	asddasd	0	leia@rebels.com	$2a$08$Ymjp2GRMvOjvSb2ChiOuXeajiBo7ZbSNrg1KhL7U8WW...	NULL	0	inactive	4PuX8c4dPu8yo	
2	leia_princess	LS0tLS1CRUdjTiBQR1AgUFVCTElDIEtFWSBCTE9DSy0tLS0tDQ...	leia_princess@rebels.com	$2a$08$omdODLS5xcnknU/g6wdsCumDM9Ls3zSTr7JlypIrW0c...	NULL	1	active	0	LS0tLS1CRUdjTiBQ
3	ciccio	LS0tLS1CRUdjTiBQR1AgUFVCTElDIEtFWSBCTE9DSy0tLS0tDQ...	lucacas30@gmail.com	$2a$08$c9B5N0SRkEbHqEYm/ERkqe/kmiepT3J5RbeTJ/IP/Yk...	NULL	0	active	0	LS0tLS1CRUdjTiBQ
4	ciaone	LS0tLS1CRUdjTiBQR1AgUFVCTElDIEtFWSBCTE9DSy0tLS0tDQ...	asd@asd.com	$2a$08$0/wlaNBnvo3zwuDbUDdz9.clXqTxvnDfd63/hPaj0AI...	NULL	1	active	0	LS0tLS1CRUdjTiBQ

Figure 19: Database view: users

basket_id	name	description	ownership	public	createdAt	updatedAt
1	rebeles info_2	LS0tLS1CRUdjTiBQR1AgTUVTU0FHRS0tLS0tDQpWZXJzaW9uOi...	2	LS0tLS1CRUdjTiBQR1AgUFVCTElDIEtFWSBCTE9DSy0tLS0tDQ...	2017-07-16 18:16:22	2017-07-16 18:30:11
2	asd_4	LS0tLS1CRUdjTiBQR1AgTUVTU0FHRS0tLS0tDQpWZXJzaW9uOi...	4	LS0tLS1CRUdjTiBQR1AgUFVCTElDIEtFWSBCTE9DSy0tLS0tDQ...	2017-07-16 18:50:37	2017-07-16 18:50:37

Figure 20: Database view: baskets

The attacker is now sure that *princess Leia* uses Thalos (but if he wasn't, why should he have attacked us?) and that she owns one basket named *rebels_info* (Figure 20). Thus the attacker tries to decrypt the basket reading the *princess* basefile where keys are stored and he gets what is shown in Figure 21. The same thing happens if he tries to read the any of the basket descriptions: they knows Leia created one containers but they still don't know how many files had been uploaded.

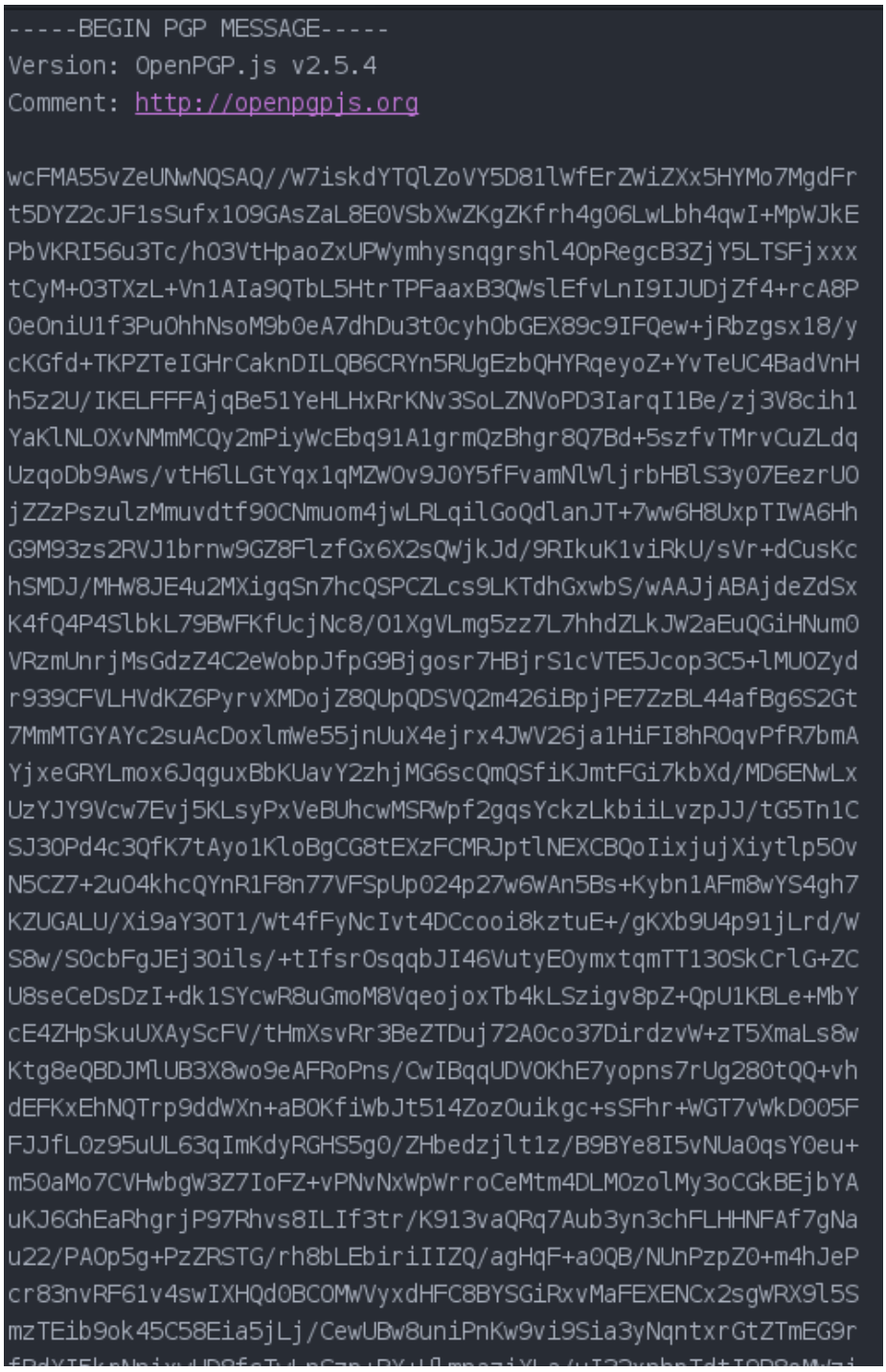


Figure 21: Encrypted Basefile: the attacker perspective

Assume, at this point, the attacker accesses the hard drive as well as the database. Here is what is displayed when he tries to list the storage directory: a lot of files are saved with a random generated 199 characters length name and encrypted with who some key (Listato 5). Again, the match between each file and its owner is recorded into the basket description that is encrypted with the basket key which, itself needs to be decoded with the master key. In conclusion the attacker will never know user's plans as long as she keeps hers Master Key in a safe place.

```
[ecelipteon@localhost Thalos]$ ls -a public/maze/
.
..
639y5fdpadh60i84hhz8aj2x9jj77ck4s5n4efjyh1hld1e4ejflhru9rw7ee3osnvvgeklvaw2psl2wtrg
dp94tk012a8k5vokoocj2oc15u13fxijlyxztowhneuo8u4wc5eg8qthf3v4hczn4si774nyk4gitu61f
fjktlobt1i7oh8njxbz4g8hx752vpgobwpvu7amymo39unbpgwroilg4tf41juzghtqn0fwhvqybgdm
nq53nvtz2lkwkw1zg0krr2bj0mfych05f4qf7ekg15oqrflykp46429diuwszsrss6yb13jm34sntdaw
```

6 Limits and possible improvements

Thalos has been developed as course project for the thelematic applications class at University of Naples Federico II. Due to the lack of time and experience many things miss and others could be done in a more efficient way. Here follows a short list of possible improvements that could be made

- *File transfer*: File transfer is made abusing the HTTP POST requests and it is hard limited to 10 MB.
- *File sharing*: Thalos stores enough informations to allow a secure sharing across the platform (with public keys).
- *Errors control*: Sometimes not enough feedbacks are provided to the user in case of error.
- *Collision control*: Although the chance of name collision between uploaded files is extremely unlikely (we are talking about random generated alphanumeric string 199 characters length), it could be wise to implement a sort of collision control algorithm that renames a file if a collision is detected.

References

[1] Network Working Group(IETF). *RFC 4880.* https://tools.ietf.org/html/rfc4880

[2] OpenPGPjs org. *OpenPGP.js* https://openpgpjs.org/ https://github.com/openpgpjs/openpgpjs

[3] NodeJS Foundation *NodeJS* https://nodejs.org/en/

[4] PUGjs https://pugjs.org

[5] Expressjs https://expressjs.com/

[6] Sequelizejs http://docs.sequelizejs.com/

[7] FlashJS http://flashjs.org/