Thalos: Secure File Storage in Untrusted Clouds

Luca Maria Castiglione and Simon Pietro Romano(✉)

DIETI - Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione, Università degli Studi di Napoli "Federico II", Naples, Italy
luc.castiglione@studenti.unina.it, spromano@unina.it

Abstract. In this paper we present Thalos, an architecture for the secure storage of files in the presence of untrusted third parties. Namely, Thalos has been conceived at the outset as a system for protecting both the confidentiality and the privacy of users who rely on an untrusted remote server for storing their files. The system ha been designed as a browser-enabled client-server application and its implementation has been conducted by leveraging the Model-View-Controller pattern. The paper discusses the rationale behind our work, as well as briefly presents the design and implementation phases by focusing on the main use cases that Thalos is capable to support.

1 Introduction

Nowadays, there is a growing interest towards the possibility of remotely storing our files while making them readily available across multiple devices. People do not normally manage their own storage servers; thus, they need to rely on third-party, cloud-based storage services like Google Drive or Dropbox, which have rapidly gained momentum in the technology market. We might ask ourselves how much secure are these kinds of services [1] and what would happen to our files if someone seized storage servers or hacked into them. More in general, we might wonder whether to trust those companies at all.

This paper presents Thalos as a solution to the above-mentioned issues.

Thalos is an extremely robust storage service that is made secure by design. The chosen cryptographic algorithms and the way they are applied offer to the final users the opportunity to securely store their files remotely, while denying any attempt to 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. It also prevents any possibility of establishing an exact match between one specific file and its owner. Thalos relies on local elaborations to perform encryption: everything outside the owner's computer is hard encrypted with asymmetric algorithms (AES 4096 bit key), according to OpenPGP standards. Due to the most known critical issues that belong to read and write operations, in fact, cryptography is executed locally on the machine of the user who owns the original contents. About that, in no way does Thalos memorize keys or pass-phrases in browser cookies or anywhere else.

R. Doss et al. (Eds.): FNSS 2018, CCIS 878, pp. 178–192, 2018.
https://doi.org/10.1007/978-3-319-94421-0_14

Thalos will be provided as a service that can be easily used, in theory, by any device connected to the Internet. Prospective users can easily register an account by using their email address and choosing a 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. The following keys are created: (i) **Master Key:** the private key of the cryptographic key pair. It belongs to the user that can unlock it through a pass-phrase chosen during the creation process; (ii) **Public key:** this is the public key of the pair and is stored on a remote database. It will also be used for secure file sharing in future improvements.

Once a key pair is generated it is possible to add a basket to one's own basket list. Baskets are virtual file containers (they can be thought of as very simple virtual file systems). 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 Fig. 1.

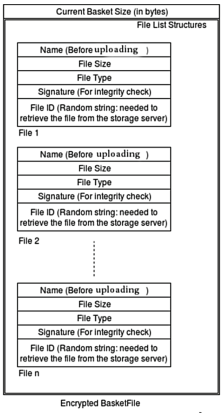

Fig. 1. Thalos basket description

Together with the basket, two new keys are generated: (i) *Basket Private Key*: used to decode the basket description and each file which belongs to the basket itself; (ii) *Basket Public Key*: used to encode the basket description and each file which belongs to the basket itself.

Basket description files are stored remotely and are encrypted with the basket private key. Furthermore, a base file is associated with each user. It is remotely stored and is 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.

2 Using Thalos

In the following we will briefly illustrate how our usual friends Alice and Bob can securely store their files using Thalos. Figure 2 illustrates Alice's example usage path with Thalos. The involved sequence of steps is reported below:

1. Alice retrieves from remote her base file which is encrypted with her Master Key Pair;
2. Alice decrypts locally in her laptop the base file and gets the keys needed to unlock her own 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 includes pointers to the actual files on disk;
5. Now Alice can securely download any files she wants to access.

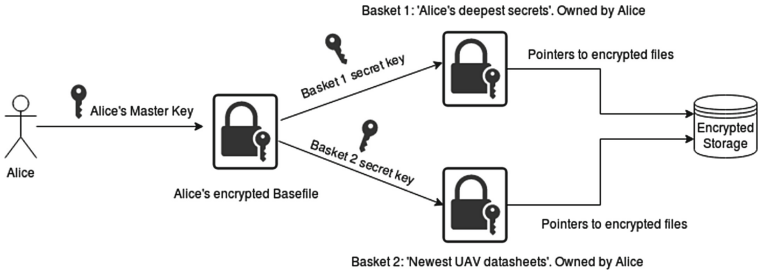

Fig. 2. Alice's Thalos use path

In much the same way, Fig. 3 sketches Bob's interactions with Thalos:

1. Bob retrieves from remote his base file which is encrypted with his Master Key Pair;
2. Bob decrypts locally in his laptop the base file and gets the keys needed to unlock his own baskets;
3. Bob retrieves from remote the encrypted description of the basket containing his deepest secrets;
4. Bob decrypts the description locally in his laptop and gets the entire file list which includes pointers to the actual files on disk;
5. Now Bob can securely add a brand new secret to his list, staying assured that no one will ever steal it from Thalos.

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

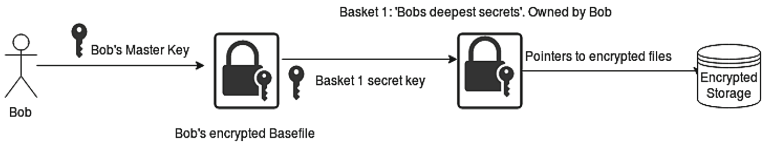

Fig. 3. Bob's Thalos use path

2.1 Protecting Master Keys in Thalos

The *Master Key* appears to be both the bottleneck and the weakness of our storage system. Indeed, the key is strictly related to the device used to read and write remote content and moving it among different laptops or smartphones might constitute a security issue. In addition, in case of a device being lost or stolen, the whole remote content would be exposed to unauthorized users. This problem has been analyzed and a mitigation has been found in what we have called the *Multiple Key Management System*. In the newest version of Thalos, indeed, a user is allowed to generate and manage more than one Master Key (MK). Once a MK has been compromised, the victim can simply disable the access rights deriving from the impaired credentials. The Multiple Key Management System has been designed by taking advantage of the hierarchical structure of Thalos. At the end of a key addition process, on the remote system many basefiles will exist, one for each key actively owned by the end user. To avoid malicious exploitations of this feature, the creation of an additional key is carried out via a feedback from an already existing key, following these steps:

1. Standard user authentication (login and password) from the 'new device'. Up to this moment the 'new device' will be logged-in and it will not be able to decrypt user's content, yet.
2. Key Pair Generation within the context of the 'new device'. New private Master Key will be stored locally while the public key will be sent to the Thalos Server along with a new key association request. The server forwards the request through a push notification towards the set of already associated devices.
3. The user will accept the request from an already associated device. Using the old (locally stored) private key, the basefile is first decrypted and then re-encrypted with the new public key.
4. From now on, the user can access his files from multiple devices.

The process introduces a minimal redundancy in the system since encryption of a single file is carried out using the keys of the baskets. On the other hand, key compromising is a well-known issue of asymmetric encryption and the Multiple Key Management System is able to completely protect the end user provided that the deletion of an impaired key is carried out as soon as possible.

3 From Theory to Practice: 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.

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 Fig. 4. The application uses PUG [4] as view engine to dynamically render HTML pages and SEQUELIZE [6] as ORM (Object to Relational Mapping) tool to dynamically map views inside a relational database and manage migrations.

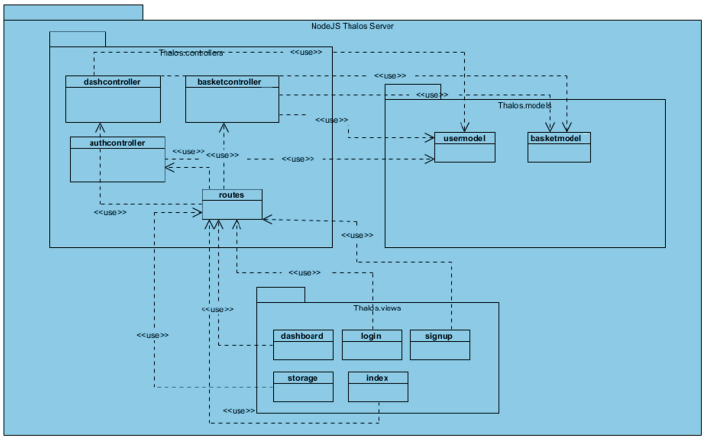

Fig. 4. Thalos architecture

Models Description. Two models are used in this application, namely 'user' and 'basket'. As it can be easily guessed by their name, the former is needed to manage users and the latter to manage baskets. Tables content is described in Figs. 5 and 6, respectively. Particular fields are:

- `users.public_key`: stores the user public key (from master key pair);
- `users.base`: stores the encrypted base file associated with 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)
```

Fig. 5. Users 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)
```

Fig. 6. Baskets table

Views Description. Views are written according to PUG syntax [4]. PUG engine dynamically renders HTML pages. Data coming 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 requests. Express manages incoming connections through the use of routes: when an HTTP request is incoming, it calls the associated callback, if it exists. The following controllers have been defined for Thalos:

- **passportController:** Defines strategies for user login and registration.
- **dashController:** Manages operations on user dashboards. It allows users to upload and download keys and base files, as well as to create baskets. The following interfaces are exposed:
  - `addBasket:` responds to POST requests. Retrieves user data from the current session and updates the basket tables with the POST parameters received along with the request. Returns a JSON object containing the result.
  - `addPublicKey:` responds to POST requests. Retrieves user data from the current session and updates the users table with the new public key and the new base file, both received from POST parameters. Returns a JSON object containing the result.
  - `getBasefile:` responds to POST requests. Retrieves user data from the current session. Returns a JSON object containing the result.

- **basketController:** Manages operations on baskets, like download/upload of a description, download/upload of a file. This controller exposes the following interfaces:
  - `getFile`: responds to POST requests. Returns file selected by file id. Result is returned as a JSON object.
  - `updateBasket`: retrieves user data from the current session and updates the basket tables with POST parameters received along with the request. Returns a JSON object containing the result.
  - `deleteBasket`: responds to POST requests. Deletes a basket. Returns a JSON object containing the result.
  - `getBasket`: responds to POST requests. Retrieves user data from the current session. Returns a JSON object containing the result.
- **authController:** Manages users' authorizations and exposes the following interfaces:
  - `login`: responds to GET requests and commands the PUG engine to show the login page.
  - `signup`: responds to GET requests and commands the PUG engine to show the signup page.
  - `validateUser`: changes user status from 'inactive' to 'active'; this allows the user to login. It's a kind of 'antispam' filter.

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 require the OpenPGP.js library [2] to perform their duty. The code is divided in three main categories, according to the functions that are carried out.

- **Operations on dashboard:**
  - `genkey`: given a pass-phrase generates a keypair. The public key is sent to the server through AJAX as user public key. The private one is the user Master Key. A downloadable file is generated on the fly and a link is displayed.
  - `addBasket`: given the Master Key, the Master Key pass-phrase and a basket pass-phrase, it generates a new basket for the user who requested it. Eventually the function updates the base file and sends it along with the new basket data to the server through AJAX.
- **Operations on baskets:**
  - `bloadlist`: given a user, it sends an XMLHttpRequest to the server asking for the base file. Eventually, it decrypts the base file and displays the user basket list.

  - `openbasket:` given a user and a basket name, this function sends an XMLHttpRequest to the server asking for the basket description file. Once got it, it decrypts it and displays it to the user.
- **Operations on files:**
  - `Upload:` this function assumes that a file (to upload) and a basket have both been selected. It locally loads the file from an HTML form, saves its related information into a JSON object, encrypts the file, pushes 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 through the remote interface UpdateBasket.
  - `Download:` this function assumes that a file (to download) and a basket have both been selected. It retrieves file information from the basket description and then requests the selected file to the server through the file id. Once a response from the server has arrived, the client decrypts it and generates a downloadable file on the fly.
  - `Delete:` this function assumes that a file (to delete) and a basket have both been selected. It updates the current basket description by deleting the selected file (identified through the provided file id). The updated basket description is sent to the server along with the query needed to remove the file from the storage server as well.

4 Dynamic Views

Some of the actions described in the previous section are herein reported through sequence diagrams. This section aims to give the reader a clearer vision of the whole project by pointing out how client and server work together.

Figure 7 shows how user registration is based on the validation of an activation code that is generated at subscription time on the server's side.

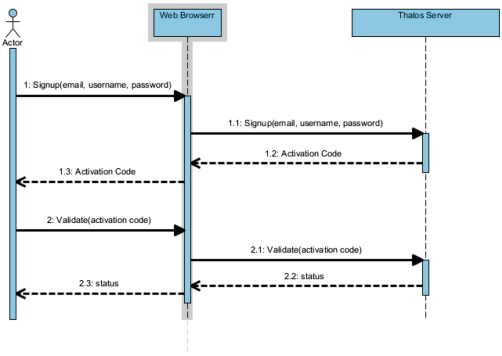

Fig. 7. User registration sequence diagram

Similarly, Fig. 8 illustrates that a request for the creation of a key pair (through a user-provided pass-phrase) is served directly within the browser. Of the pair in question, the private key is provided back to the user, while the public key is delivered to the Thalos server, where it gets stored for all future uses.

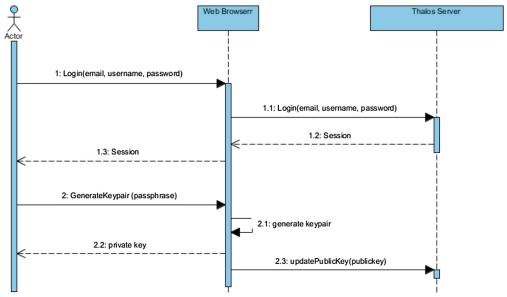

Fig. 8. Master key pair generation sequence diagram

Figure 9 sketches what happens when creating a Thalos basket. Basically, the original base file is first retrieved from Thalos and locally decrypted with the crypto material provided by the user (pass-phrase and private key). Then, the *addBasket* method is triggered, which translates into the creation of a brand new basket key pair, the update of the downloaded base file and eventually the upload of the updated (and encrypted) base file to the Thalos server, together with the newly generated basket public key.

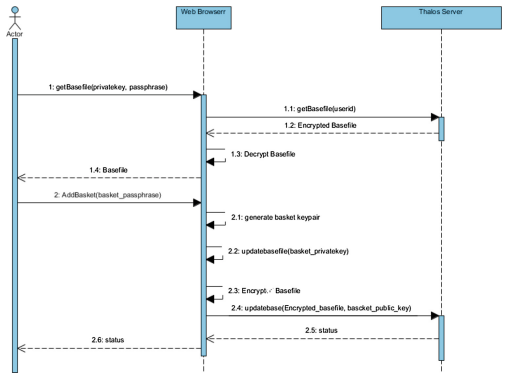

Fig. 9. Basket creation sequence diagram.

Figure 10 describes the process of retrieving the list of files contained inside a basket. As already discussed for the previous case, we first download the encrypted base file, which is locally decrypted with user-provided information.

Then, we call the *getBasketList* method on the client-side Thalos JavaScript library. With the basket list readily available, we can eventually call the client-side *openbasket* method, which in turn downloads from the Thalos server the encrypted basket description file. As usual, the encrypted description is locally decrypted and the resulting file list is provided back to the requesting user.

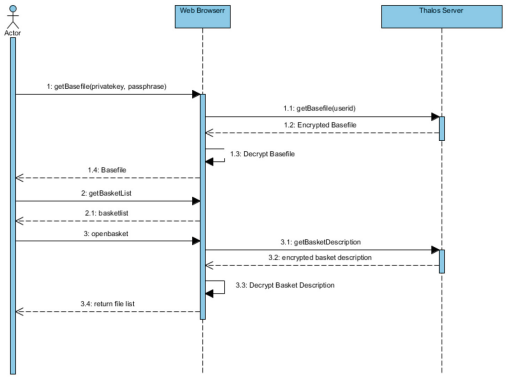

Fig. 10. Basket list retrieval sequence diagram

Figure 11 focuses on file upload. Steps 1 through 3.4 are exactly the same as those described when commenting Fig. 10. Starting from there (i.e., assuming the file list has been made available to the end-user), we can call the client-side *UploadFile* method, which: (i) updates and encrypts the basket description; (ii) encrypts the file to be uploaded; (iii) uploads to the Thalos server both the encrypted file and the encrypted (as well as updated) basket description.

Finally, Fig. 12 focuses on file download. Once again, we start from step 4 in the diagram, which shows how a call to the client-side *GetFile* method gets translated into an analogous *getFile* call to the Thalos server. Such a call allows the browser to download an encrypted copy of the requested file, which is decrypted on-the-fly and provided to the end-user in the clear.

5 The External Perspective

In this last section we try to follow the breadcrumbs left by a user who uses Thalos to securely store his/her files. The goal is 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 Fig. 13, the actor in question first of all creates an account in the webapp and validates it.

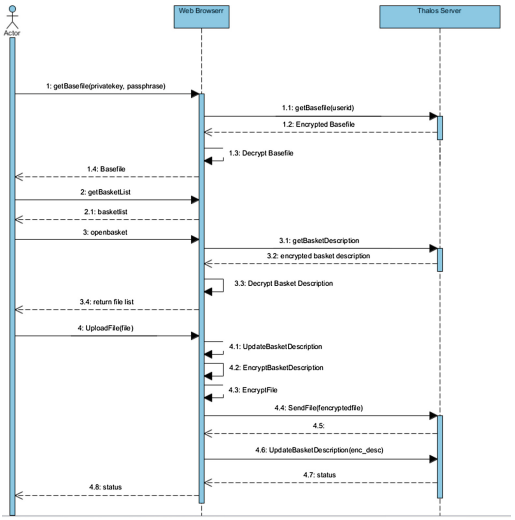

Fig. 11. File upload sequence diagram

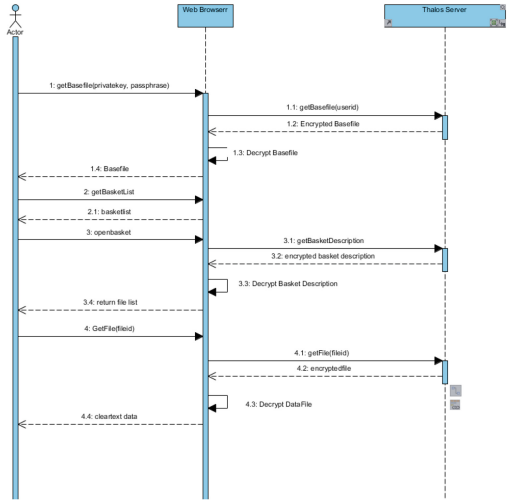

Fig. 12. File download sequence diagram

Once done with the previous step, our special guest signs in through the dashboard and generates his/her master key pair by clicking on the *Generate Master Key* button (Fig. 14).

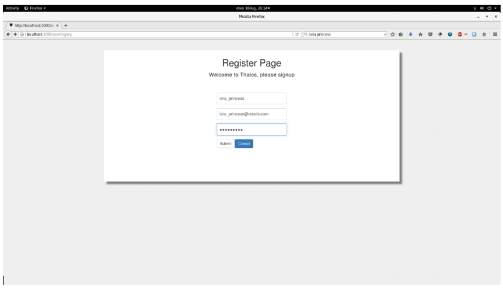

Fig. 13. Account registration

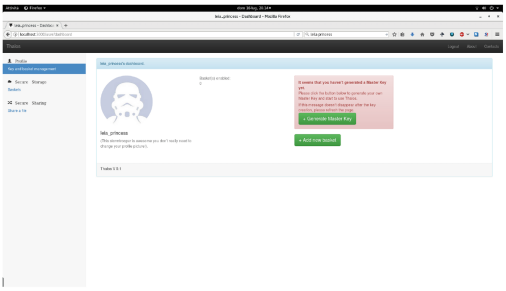

Fig. 14. Dashboard

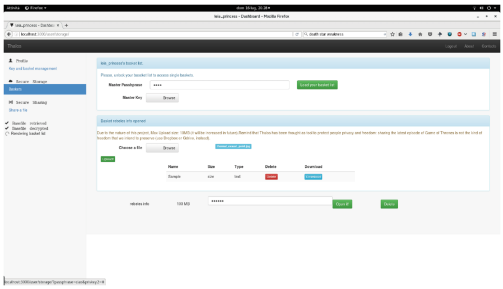

Fig. 15. Information uploading

Using the newly created key pair, the user adds a basket to his/her basket list by following the instructions displayed in the web consolle. Eventually, he/she uploads confidential information (Fig. 15) to the remote Thalos server.

From now on, the file is in the secure storage and it is ready to be downloaded by its owner whenever the need arises, as illustrated in Fig. 16.

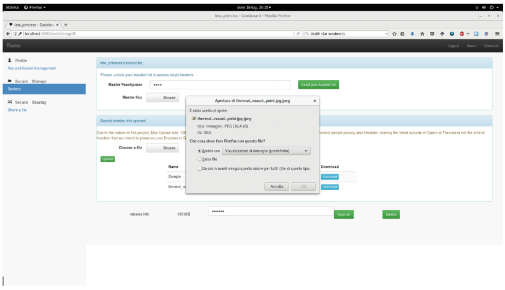

Fig. 16. Information downloading

Let's now assume that a bad guy has gotten, in some way, access to the database. What can he actually learn about our user? In Fig. 17 a view of the database is shown.

Fig. 17. Database view: users

From the attached snapshot, we can derive that the attacker is now sure that *princess Leia* uses Thalos and that she owns one basket named *rebels_info*, as reported in the further snapshot in Fig. 18. Thus, the attacker tries to decrypt the basket in question by reading the *princess* basefile where keys are stored. What he gets is just a meaningless sequence of PGP-encrypted bytes.

The same thing happens if he tries to read any of the basket descriptions. The only information that is 'leaked' is that Leia created one container. Though, nothing is leaked with respect to sensitive data like, e.g., the total number of files that have been uploaded.

Fig. 18. Database view: baskets

Assume that, at this point, the attacker accesses the hard drive as well as the database. When he tries to list the storage directory, the only thing that he realizes is that a lot of files are saved on disk with a random generated 199 characters length and a name that is encrypted with some key.

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 discover

sensitive user's information as long as the user in question keeps his/her Master Key in a safe place.

6 Related Work

In this section we provide an overview of common services that focus on *untrusted computing* and its application to the field of secure storage techniques. We will try and highlight their differences with respect to our solution, for better or worse.

MegaNZ is probably the most famous service on the market offering secure storage for free on the Internet with a file level granularity. Mega developers have written from scratch their own implementation of encryption algorithms using Javascript asm. Files encryption works locally to the user machine and their work has been open-sourced[1]. Moreover, the service is offered through a friendly interface and the asymmetric encryption process is completely transparent to the end user. On the other hand, the infrastructure does not provide any form of anonymization. In fact, as written within the privacy policies, information on files such as metadata, ownership and upload date is clearly stored by remote servers[2]. Also, the asymmetric encryption breaks when the needs arises to share a file. In such a case, the key needed for file decryption has to be explicitly provided along with the file. Finally, the entire server side architecture is kept hidden by the company; in this sense, the service cannot be deployed within a private network.

Storj [8] is an interesting service that uses a pure P2P configuration in order to keep user files secret. Every file is split in hundreds of shards that are stored, encrypted, all over the nodes. The service is completely free and the code, written in C, has been open-sourced. Unlike our solution, this service comes with the strengths and weaknesses of peer-to-peer and neither reliability nor availability of files can be ensured under all circumstances.

Clear storage with an encrypted security layer is another approach that can be considered capable to reach a good privacy level in remote storage. It consists in adding a double key encryption layer to a common storage service such as Dropbox, Google Drive and Microsoft One Drive. Many applications have been developed with this purpose but, unlike Thalos, they cannot provide the user with file anonymization. This approach indeed requires a user to be aware of the common privacy issues, as well as of the existence of countermeasures such as advanced encryption.

[1] https://github.com/meganz/webclient.
[2] https://mega.nz/privacy.

7 Conclusions

In this paper we have presented Thalos, an architecture for the storage of content within third-party storage facilities, with both security and privacy guarantees. We have discussed how Thalos has been designed and implemented as a remotely accessible, web-enabled service. We have also briefly compared Thalos functionality with wide-spread cloud-based storage facilities.

Thalos has been already presented to the international security community as an open-source tool for the secure storage of contents in the presence of untrusted third-party storage providers. Namely, the project has been presented at the recent BlackHat Europe 2017 conference that has taken place in London between the 4^{th} ad the 7^{th} of December 2017. BlackHat is a renowned venue for security researchers and practitioners, providing attendees with the very latest advances in research, development, and trends in Information Security. Thalos has been part of the so-called *BlackHat Arsenal*[3], that is a session entirely devoted to the presentation of cutting-edge tools in all fields of security.

Source code, documentation and installation information for Thalos are all publicly available on gitlab at the following address: http://gitlab.comics.unina.it/NS-Projects/Thalos.

Acknowledgments. This work was partially funded by the European Space Agency, within the framework of project SHINE, ESA Contract No.: 4000118273/16/NL/CLP.

References

1. Rong, C., Nguyen, S.T., Jaatun, M.G.: Beyond lightning: a survey on security challenges in cloud computing. Comput. Electr. Eng. **39**(1), 47–54 (2013). ISSN 0045–7906
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/
8. Storj. https://storj.io/

[3] http://www.blackhat.com/eu-17/arsenal/schedule/index.html.