Design and development of a distributed, secure and resilient vault management system

Mathonet G.

University of Liège, Belgium

June 2017
This thesis aims at solving two practical issues:

- Ease the storage of personal private data
- Ease the sharing and update of such data

When reviewing existing software, it either:

- Makes commercial use of the acquired data without consent
- Does not allow for sharing
Obviously, we want to provide some strong guarantees to the end users:

- An end to end privacy between sender and receiver
- A decentralized system for availability and resilience
- Create a simple, extensible and if possible open-source stack which allows for interconnection
Functionally, we want to provide the following services:

- Allow anyone to store data pieces which the server cannot access
- Share those data pieces to other users, which is called receiving a vault
- Provide an API, and a client side API for obtaining SSO and vaults on the fly
A secure mail exchange model (1)

User 1

2's Public Key

User 2

2's Private Key

Session key encrypted using 2's public key

Data encrypted using session key
Based on this, we want to allow updates.

We can encrypt the session key one more time with 1’s public key, we call this creating *bound vaults*.

If we want the update not to be propagated to 2, we have to duplicate the whole data, creating *unbound vaults*.
Key setup (1)

User

Server

HTTPS

Send public key and identity

Sign public key with own private key, generating certificate

Echo the certificate

Ask for a challenge

Save a local challenge

Encrypt the challenge using own private key

Echo challenge

Send response

Decrypt response, if match, identity is in the certificate

Echo result of authentication
Key setup (2)

User

Server

Send identity and password

Create a keypair, sign the public part, encrypt the private part, keep but a hash of password salted

Say OK

Accept authentication

Check for match when hashing it

Ask for private key

Send private key, encrypted

Decrypt private key
The first model is obviously better, security-wise. However, it is harder to use for humans so both models will co-exist. We still need a PKI infrastructure for users who delete their account, etc.

For the ciphers themselves, their choice is mainly driven by the small-end devices capabilities. We might even allow for temporary key sharing for faster encryption or decryption.
Other security concerns

- We do not support perfect forward secrecy as getting the private key of a user would allow for reading all of his data
- Protection against DDOS, mainly by token bucket
- Tarpitting
- Client side puzzle
API usage

- We expect two types of users: humans and (large) companies.
- When using client side certificate, the SSL connection itself is trusted.
- Otherwise, a token or the credentials must be echoed with each request.
- The API itself cannot be behind a CDN.
- However it can delegate the SSL handshake to a reverse proxy.
- We will use the most used nowadays JSON format.
We do not require SQL: everything is encrypted!
We can only query the database using an ID
HBase is probably the fastest at storing key-value mappings
MongoDB stores documents in JSON formats already
MongoDB replication better suits our model (replication at TCP layer)
To allow for each user to maintain his documents and update them, we will create virtual filesystem

But the document maximum size (16MB) will have us split users over several documents

We are using bigrams or trigrams which provide a recursive level

Trigrams are slower, but should allow for never needing two levels of recursion
Design and development of a distributed, secure and resilient vault management system

Mathonet G. (University of Liège, Belgium)
To allow for each user to maintain his documents and update them, we will create virtual filesystem.

But the document maximum size (16MB) will have us split users over several documents.

We are using bigrams or trigrams which provide a recursive level.

Trigrams are slower, but should allow for never needing two levels of recursion.
Database (5)

Mathonet G. (University of Liège, Belgium) Design and development of a distributed, sect

June 2017 16 / 28
The server obviously needs to be replicated, and so must be the database.

It is already provided between some local nodes by the database engine itself (MongoDB shards).

We want to study the speed of convergence of the knowledge of a new use inside the system.
Replication algorithms (2)

- Resource Location Indexes (RLI) is a peer-to-peer model
- Servers propagate the rows they locally have with partial or full updates at different intervals
- Servers query peers for existence, and the fetching is done recursively

- Captain Cook (CC) is a tree hierarchy
- Servers propagate the rows they locally have to their local root, which themselves propagate what they know
- Servers query roots for existence, and the fetching is done by themselves once existence resolved
Performance

- Messages are sent authenticated as a master user, and serialized using Google Protobuf.
- When investigating the time required to bootstrap a new user, the variables are the length of messages exchanged, the number of active users, and the interval between messages.
- As the processing time of the messages is shorter than the interval between messages, it makes sense to study the results in terms of number of cycles required for the process.
Frontend session (1)

- The servers are RESTFUL, but the clients side will emulate a stateful server, for convenience
- By storing the user’s keys, and a token if using a password
The client must also be made secure, as long as the device is not compromised.

- Against CSRF by using local storage rather than cookies.
- Against XSS by using libraries known for their good protection against it (Angular 2).
The goal of the program is to help people share data between each other.

Defining a known format for this data is a valuable feature.

The whole definition of the fields, their contents and validation methods are defined within JSON and evaluated dynamically to be easily modified.

The client side supports transition schema’s to re-encode data to a new definition if it were to change.
Latest features (1)

- Symbolic links for sharing the same data at several locations towards the same user
- Detachable **vaults** which can be claimed by the granted user
- Merging of accounts, which imply to make the decryption of session keys verifiable to know which private key to use for a user which has several
- Password recovery
Latest features (2)

1. Ask for password recovery

2. Ask for registered other owners of parts, restore email, and for part of Whigi-Restore

3. Return the encrypted vault to be decrypted containing the answer

4. Ask for password part

5. Ask for user's part

6. Return to be decrypted part (normal vault)

7. Return password, unencrypted

8. Send a mail with a once-only usable key, timed limited, that can download encrypted password by mail. The key for the password is sent and forgotten.
Our project, WiSSL, can be used using the API, and there exists an API for:

- Use it as identity provider
- Request for authorization (OAuth)
- Request for a grant on the fly
We have created a platform in which, by its design, users can trust.

Remaining work: test replication at a larger scale and study transition schema’s for ciphers.

We have projects which use the platform: WiSSL-Contacts, WiSSL-Advert or WiSSL-Voting.