Secure Channel Chat: Report

By Emile Sonneveld and Bert Van Mieghem

Introduction

The goal of this project is to develop a messaging service that allows users to communicate one-to-one or in channels using text messages or files. The transfer of these chat messages and files should be encrypted end-to-end and thus completely safe. There is a central server, but an attacker with root access on this server should not be able to know what users are in which channels or be able to read the content of any message.

The protocol

Register as a new user

The messaging protocol goes over different layers. First, all users need to register. They do this by logging in with Facebook OAuth. OAuth is very similar to OpenID, it delegates authorisation to a third party where the user probably already had an account. OAth is actively used by Facebook, and that is why we used it.
An extra advantage of Facebook-OAuth is that we can ask for the Facebook-friend list of the logged in user. This way we can gain more easily trust of the user that he won't be chatting with impersonators.
One could think that a downside is that Facebook will know who is using this chat service. We don't take that as an issue as we consider this public information.

Facebook's implementation of OAuth expects a callback URL to redirect the user to when he is authenticated. As our application is not a website, so we had to implement a small loophole: Behind the screen, we launch a client side HttpServer that is only accessible on localhost. When the user clicks the "Login" button, the default browser is opened with a "Login with Facebook" button. When the user presses this button, the Facebook authentication popup will show. After this, the login page will be closed, and the java application will receive the Facebook-login-token. This token only keeps valid for a few seconds.
The client side HttpServer has the default CORS settings and is not accessible to other web pages opened on the same computer.

Now that the facebook login token has been obtained, we can contact the main server. We generate an asymmetric keypair and send the public key together with the facebook token to the main server. The server will use the token to get the facebook username and id and store all information in its database. The client then asks for a list with all registered users and verifies it's facebook_id with the one the server found out.
Note that we use slow hashes of facebook_ids to protect make it more difficult to find what Facebook user corresponds to a certain hash. In the other direction, it is easy to search for a facebook friend between the users.

Handshake with other users

Clients only communicate with other clients 1 to 1. A handshake is an exchange of vital information to allow secure communication.
Every client has a list of all the other clients on the system. This list contains public keys, hashed Facebook IDs and names.
Client A makes a *symmetricKey* that will be used for sending messages to client B
Client A also makes an *ephemeralId*, that will be used as a kind of nickname for targeting B
It constructs a message and sends it to the handshake buffer on the server.

$$\textbf{\{symmetricKey\}publicKey}_{\textbf{B}}\textbf{,\{\{facebookID}}_{\textbf{A}}\textbf{, ephemeralID\}Sig}_{\textbf{A}}\textbf{\}symmetricKey}$$

B can decrypt this handshake, the server only stores this handshake for transit, it does not know who sent it, to who it is destined, or what it contains.
Because it is not possible to know to what user this handshake message is targeted, all users need to download all handshakes from the server and try to decrypt them. Once client B has decrypted this handshake, he knows that future messages to him will be labeled with the *ephemeralID*.
This handshake is unidirectional. If B wants to send back messages to A, he has to send a handshake first.

Sending messages

Messages are the general term for information that is exchanged between clients. For example, a chat message is a type of message.
When A wants to send a message to B, he must send the following to the server:

$$\{\{\{payload, facebookID_A\}Sig_A\}symmetricKey, ephemeralID$$

Where *symmetricKey* and *ephemeralID* are defined by the previous handshake are agreed up.
The server will store this message for future user.
Also here, the server only delegates the messages, it does not know who sent it, to who it is destined, or what it contains.
When B want to check if he received a message, he will pull all the messages targeted to the *ephemeralIDs* that he received with handshakes. He knows that he will be able to decrypt those messages.
As an convention, we only send JSON through the messages

Making channels and chatting

Through the messaging system instructions and data are sent between the clients. To create the notion of channels, each client keeps a local representation of a channel. It contains an owner, a name, an unique GUIDand a list of all members. The state of a channel is updated with event sourcing.
For example, when a client sends a chat message, it has to send a copy to each member of that channel, separately encrypted. This is called technique client fanout.
A chat message could look like this:

```
{
   "content": {
      "channel_uuid": "2dcf4551-77fb-4738-a4d9-264bafc04720",
      "chat_message": "test"
   },
   "message_type": "chat_message_to_channel",
   "sent_time": "2019-06-04T13:02:35.936961200Z"
}
```

Creating a channel has a tricky part. Assume that A wants to make a channel together with B. She will first make a local representation of a channel where she is owner. To keep consistent with event-sourcing, she will send herself an invitation message.
Then she can invite B to the channel. Here she will send the whole representation of the channel, expect the messages, and send it an invitation message to B.

```
{
```

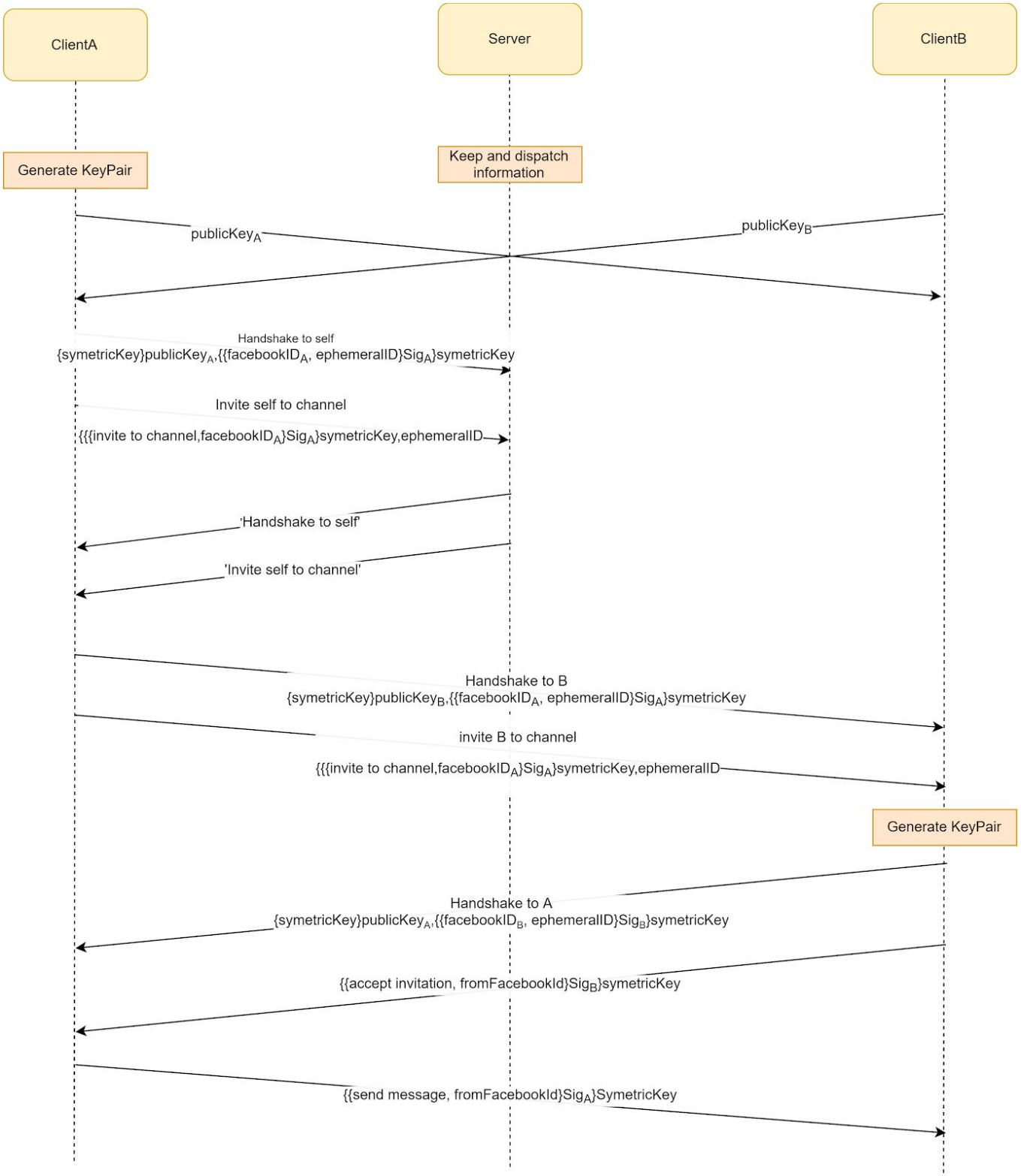
```
  "content": {
    "channel_content": {
      "chatMessages": [],
      "members": [
        {
          "facebook_id": "3ioNdDVMU7lgO2tt9jwxglyVzAUBqcMyagqxZM+ukQc=",
          "status": "OWNER"
        },
        {
          "facebook_id": "1cirmiPBx5bSE6nRgYAqS4nemF49cJ422CeErEX2o1k=",
          "status": "INVITE_PENDING"
        }
      ],
      "name": "TheChannel",
      "uuid": "2dcf4551-77fb-4738-a4d9-264bafc04720"
    },
    "invited_facebook_id": "1cirmiPBx5bSE6nRgYAqS4nemF49cJ422CeErEX2o1k="
  },
  "message_type": "invite_to_channel",
  "sent_time": "2019-06-04T13:02:28.520343700Z"
}
```


The *chatMessages* are not included, as they are sent between users in the channel without them expecting that someone in the future could read them. Also, it would prove difficult to cryptographically verify their authenticity.

Sending a file is implemented as a special case of a chat message that contains a base64 encoded file.
Other actions on channels are also possible, like renaming, accepting an invitation, removing a person. All according to the requirements.

We intentionally didn't implement requirement 4 completely (*"Any user can send invitations for other users to join channels they created. Any user, provided he/she knows the name of a channel, can post a request to join the channel."*). Our reasoning follows: If one would be able to find a channel by name, people could start to guess what conversations are taking place on the server by searching for example for names like 'Resistance against Israeli Government'. This could be countered by not confirming that the channel you want to join exists or not. The channel owner would receive a message that someone would like to join his channel. And only when he accepts, the joining person will get notified.
But this would also cause a privacy issue. As we can have multiple channels with the same name a request to join a channel would be forced to be sent to all channels sharing the same name. So a spy could make a huge net of channels with different names to see invitations coming in.

Activity Diagram of communication between clients



The code

The application is written in java and available as open source. A demo video is also included: https://github.com/BertVanMieghem/Messaging-with-End-to-End-Encryption
The client uses swing to render it's UI.

Injection

Swing is less vulnerable to injection-attacks compared to JavaScript.
We explicitly had to disable HTML-rendering for labels. HTML-rendering would make it possible to inject <img> elements on some places of the GUI that could reveal the IP-address of the person seeing this image.
We didn't use an ORM like Hibernate and have double checked that no of the SQL statements allowed SQL injection.

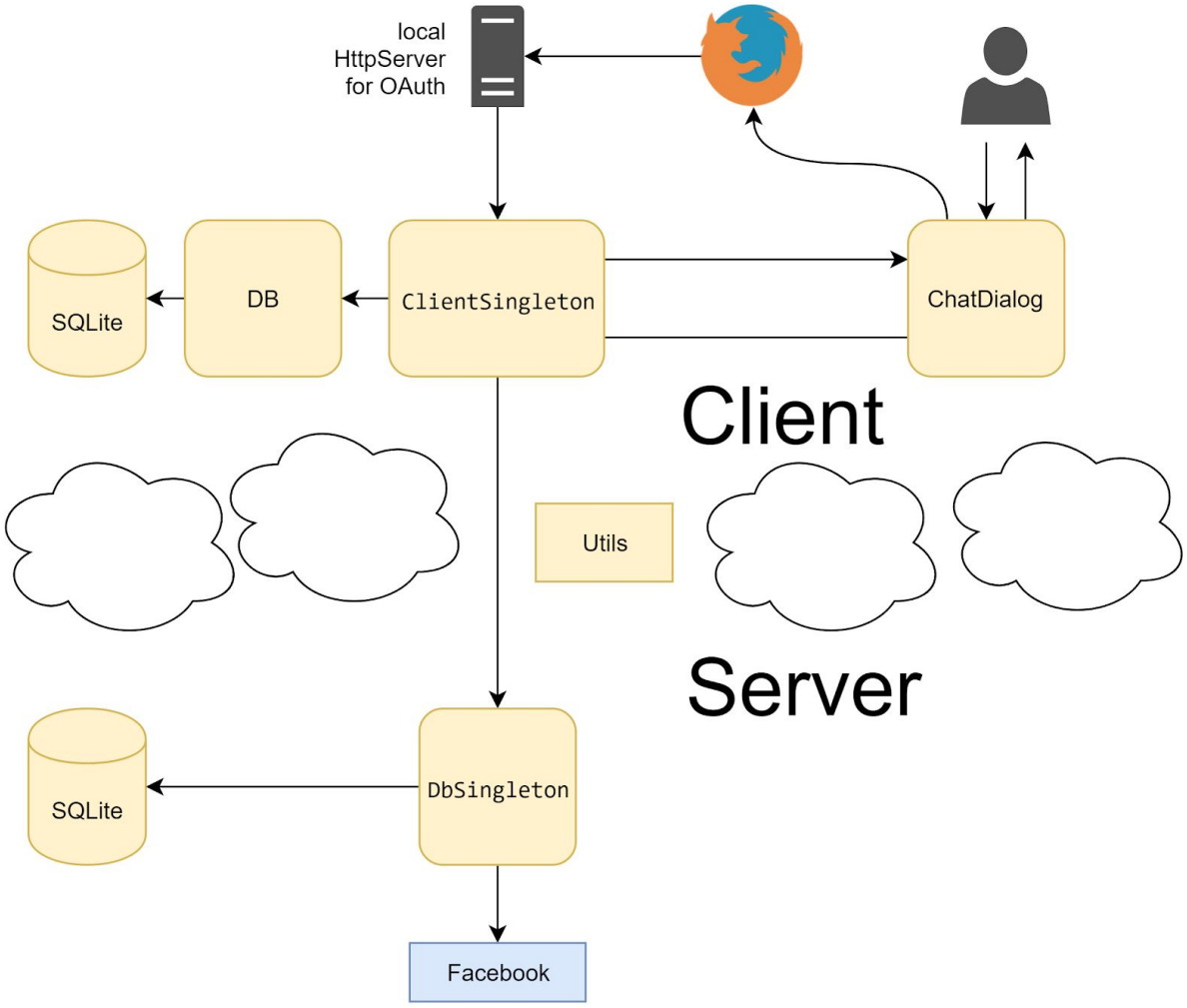
Linting

Codacy was used to analyze the application to find security issues. For example, Codacy was able to find a switch statement with an incorrect use of a break-statement which could result in hard to detect bugs and possible security issues.

Single device only

This application is restricted to one device per user. If a user wants to check his messages on another device, he should copy the local database from his original device to another. We only support one private/public keypair per user.

Diagram of the application



Further improvements

- Let the client-server communication go over a Tor-network.
- Pass the code trough FindBugs / hire a pentester to test the application
- Server-fanout could be used instead of client-fanout, to increase performance. As described here: https://jameshfisher.com/2017/10/25/end-to-end-encryption-with-server-side-fanout/
- Encrypt client-database with windows session, just like chrome: https://www.howtogeek.com/70146/how-secure-are-your-saved-chrome-browser-passwords/
- Allow clients to compare their public keys via a side channel. To detect a possible man in the middles. This could be done by making an unique ‘safety number’ which is a mix of both facebook_ids and both public keys of the users, but hashed. This hash should be the same for both parties in a conversation.
- For consumer support, make a tag on a third-party service like StackOverflow to discuss issues with the application openly. We’d use a third-party, so the user can trust we don’t censor embarrassing issues.
- Put jitter on requests against timing attacks. Also add jitter to size of requests.
- The protocol could be formally verified: http://proverif16.paris.inria.fr/index.php

- DDOSing is fairly easy. If a malicious user would send many valid or invalid handshake requests, the whole system could slow down. This could be mitigated by rate limiting rogue users.