lt-se.me Documentation

Release 0.9-rc2

Benjamin Kampmann

April 09, 2014

It-se.me is a secure, privacy-focussed contact interchange service. Heck, it is so concerned about privacy, it itself doesn't know your contact details.

It's goal is provide a secure system to allow seamless and automatic user lookup for jabber clients (like ChatSecure) based on the users address book while requiring the least amount of private data as possible. It is an open system, with an open API, which may be used by anyone and anything but is designed and build with this specific case in mind.

It's hope is that by providing a seamless while secure user lookup service, more people may be willing to switch and use more secure chatting solutions based on open protocols and standards.

CHAPTER

ONE

WANT TO LEARN MORE?

1.1 Foreword

We live in a time, where government but also big corporations can't be trusted with our data. They have proven over and over again that they won't only collect more data about us than needed but even further they will use them against each and everyone of us. As a result, we have to protect us from them.

Direct person-to-person communication has always been an essential part of telecommunication services and the security and privacy is vital for freedom of speech and in general. And though the technology is there to offer protected communication using XMPP/Jabber, OTR and ZRTP their usage isn't as widespread as it supposed to be.

1.1.1 Problem of usability of secure services

One of the major inhibitors for wide-spread adoption lies in their cumbersome usability and conceptional flaws when it comes to day to day usage. This projects tries to fix one of these issues: how do you figure out, who of your contacts is already using a secure service (like ChatSecure) without having to expose your own address book and privacy relevant data.

1.1.2 Our tiny piece in solving this

Using well-known cryptography, we've developed **a system, where not even the storing server has to keep the contact details but still provide the feature-rich experience of "people autodiscovery" of Whatsapp and alike**. But don't trust our word, here is the source code and a full explanation of how the system works.

Unfortunately, we live in a time, where the fundamentals of the internet are broken. This is a corner stone in taking back our privacy, security and eventually our all personal freedom.

1.2 The Concept

So, then how can you share your contact details without even the server storing them for longer than needed to authenticate them? And more-over, how can another client request permission to contact a third party through this services without leaking privacy details about the other person?

1.2.1 Scenario setting

In order to create secure and privacy concerned system we've set out to create a concept for contact details interchange through a trusted instance under the following conditions:

- the service acts a trusted middle man
  - as of that all contact details need to be authenticated against that service
  - **but as also a trusted middle mans security can be breached** the service shall know as little privacy concerning data as possible, specifically it shall not know which email belongs to which account
- a third party shall be able to authenticate itself against that service with one or multiple contact details and through that authenticating be able to deposit a target-address others shall be forwarded to in case of a match
- when a third party wants to connect with other people, it shall be able to do so without leaking their own address-book of contacts
- if a match is found, the deposited account shall be informed about said request by forwarding their own authenticated details
- that account can then match those account details against their own database and decide to act accordingly

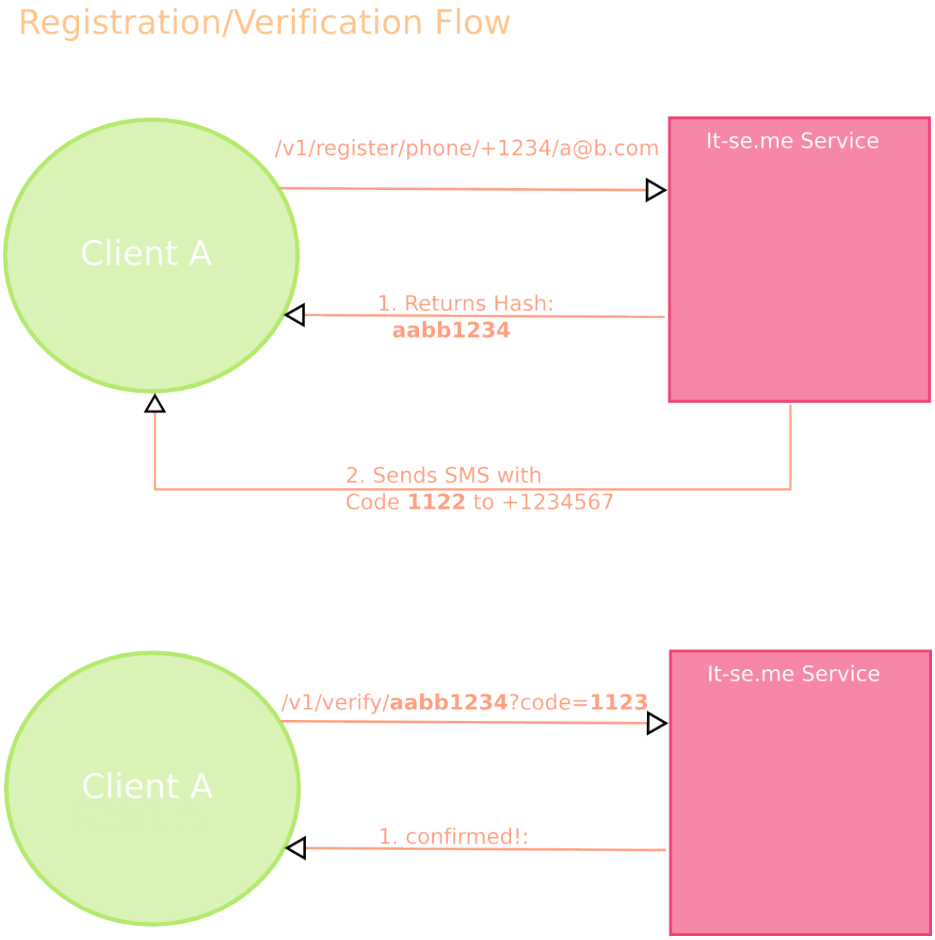
1.2.2 Set up

The way we've implemented this is by using extensive hashing (SHA512) for account information storage. Meaning, that the contact request details will be hashed and the deposited account will be linked to it once authentication of that channel is done.

1.2.3 Registration and Authentication

Let's look how this works. Let's say Amy is starting a new instance of her ChatSecure and want to connect with her friends already stored on her phone. The first thing the App will do is prompt Amy to authenticate various contact channels of her own. For example her phone number. For that the App sends a registration request with the medium *phone* and the specific id (in this case a phone number) to the service for authentication together with the target (a jabber-id) this details should be connected to. The service then sends back the corresponding hash and a SMS to the phone number given.

Once the phone receives that code, it sends this together with the hash to the service. The service then checks it against the code and if properly authenticated discards the medium and specific id. The only thing the server now knows about is a one-way-hash that belongs to a given target it shall contact.



Amys service does this with a few contact details, for example the phone number (currently only SMS is supported), her Facebook and Twitter Account, Github and Email-Address. For each access-type the service will only store the hash and the given new target address to talk to later. Through which a reverse look up is not possible.

1.2.4 Contact Lookup and matching

The server knows that behind a given hash of medium+specificId is a given jabber-account to talk to. This one belongs to Amy but the service doesn't know and also doesn't care. Her friend Michael wants to connect with her through his newly created Jabber-Account but doesn't know Amies account. He does, however have her phone number. So after he also authenticated his jabber account against his details, he sends a request to the server to connect with Amy. But instead of sending the contact details, Michaels app generates the hash itself and only sends that, as well as Michaels contact details to the service.

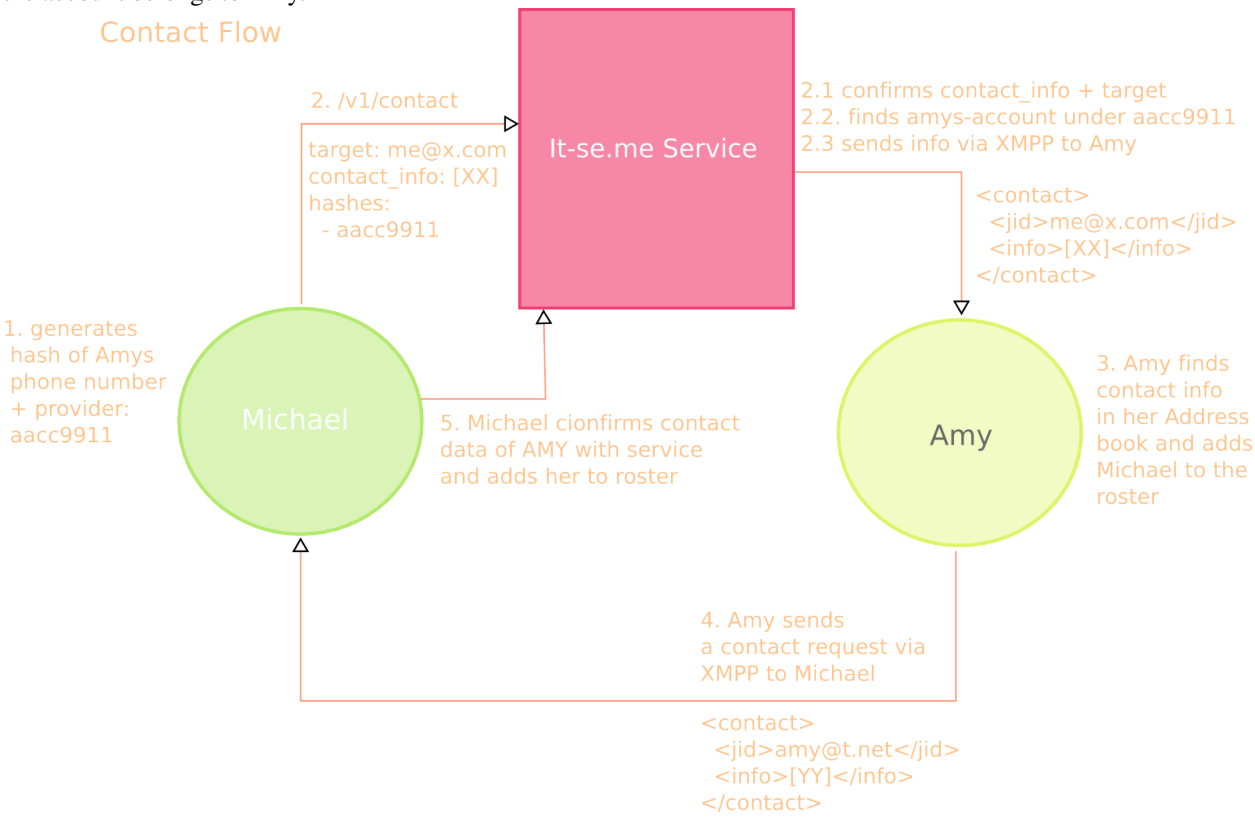
The Service then authenticates that the given details coming from Michael indeed to belong to that given Jabber account (by probing against the hash) and then looks up the hashes the App provided. For one hash the service finds the jabber account of Amy. Now the service sends a message to that jabber account, providing said confirmed account details. Up to this point, the two aren't connected yet. But as soon as Amys app receives that request, it can look up whether it has any of the provided information in the addressbook itself.

As the service, as the trusted instance, had previously verified the owner of those information to be the account that asking, the App can be sure that the contacting jabber account given indeed belongs to Michael it found when looking up the Phone number in Amys phone book. At which point the App sends a connect request to the provided jabber account.

Up before this point Michael had no knowledge about the other account yet as the trusted service didn't reveal whether

or to whom it has sent any request to protect the privacy of Amy. If Amys App, for example, didn't find the requesting details in the phone book it should prompt Amy about the request and Amy can then decide whether she wants to connect to Michael or not. Michael would never know whether Amy has even gotten any request, if she chooses to not connect with him.

But in this scenario, Amy has Michael in her database and indeed wants to be connected. So her app has sent Michael's account a request with the data she previously confirmed through the trusted instance. When Michaels phone receives such a message, it matches it against the pending hashes it requested previously first. There it finds that the hash for the phone-number, to its knowledge, belongs to Amy. One call with both the hash and the account the message was sent from to the trusted instance, confirms that indeed those are the connected details and Michaels phone can be sure the account belongs to Amy.



The trusted instance doesn't keep records of who connected whom nor where look ups came from. It discarded all details information and isn't even included in the processes of knowing whether those requests it forwarded did, in the end, connect to each other. Nor did it ever have the knowledge, who was really behind those. All it knows about is a bunch of hashes that will be privately redirected to bound accounts. Michael was able to connect with Amy and no one will ever know.

1.3 API Documentation

1.3.1 Basics

The API-Service can be queried and managed using HTTP-Calls to *api.it-se.me* and it will answer with JSON-Formatted output.

This is just the basic API documentation. To understand the underlying conceptional idea, please see :ref:concepts and :ref:faq.

Warning: Though we consider the v1-API kind of stable and want to only add new features, it is still in a Beta-Testing phase and might be subject to change if problematic security concerns are raised. Please keep this a default-off extended feature in your (production) App until further notice.

Limitations

In order to allow wide, anonymous access to the API and foster a wide addoption there is no App authentication necessary to use the service. However, there is a rate limitatin of 10 request we allow per second per IP. There might also be specific limitations per call.

Error Codes and Messages

For general Error reporting, we are using HTTP-Error-Codes. Further those provide some insight of what went wrong as json like in this example:

```
{
    "error": {
      "code": "target_unconfirmed",
      "message": "Please confirm 'X' first."
     }
}
```

1.3.2 Register

URL: /v1/register/<provider>/<provider_id>/<target>

Supported Methods: GET

Optional Parameters: resend=1

With this method you can register a new entry to the service. You have to specify the provider (medium) and the id on that one as well as the target xmpp-id it should be bound to.

If you call it more often for the same *target* it will only send the confirmation code once. If, and only if, the User asks for it to resend please attach the "?resend=1" parameter to the call, which will generate and send a new code.

If everything goes according to plan, you'll receive a json response as follows:

```
{
 "status": "pending",
 "hash": "397ee3ee893ba686b8f228078803ce34911b35c8bf15a7986310de1225589fe13706a3242376da92c144a0e38e4
}
```

The response might contain a *goto*-field containing a URL. If present the APP *must* redirect the user to this URL to continue authentication. Otherwise the confirmation code will be handled in a different way (see providers).

Note: Before you can bound any medium to a jabber-id, you first need to confirm said ID by requesting the *provider* 'xmpp' with the given jabber-id as *provider_id* and *target*. For *xmpp@example.com* this would for example be */v1/register/xmpp/xmpp@example.com/xmpp@example.com* and you won't be able to bind anything to this target until it has been verified.

Providers

Currently the following providers are support and acts as follows:

xmpp: The xmpp provider expects a jabber id as the *provider_id*. It will then connect to that account and send a four digit number to be used to verify against the server. This provider shall also be used to initially register a jabber id by providing the same id for *provider_id* and *target*. The app should either wait for that request or prompt the user to wait for that code.

email: Expects an email-address as *provider_id* and will then send a six digit verification code plus URL to the given Email-Address for confirmation. The App should inform the user about this and ask them to confirm manually.

phone: Expects the *provider_id* to be an international phone number including *+XXX* number codes. An example call would be: */v1/register/phone/+1234456789/xmpp@example.com* . This provider will send a SMS with a four digit code to the given number.

facebook: The facebook provider expects your username as the *provider_id*. As it uses OAuth, the return will contain a *goto*-url the App should redirect the User to in order to sign in and grant the permissions

twitter: Similar to facebook the *provider_id* is expected to be the twitter username and it will also return a *goto*-url to redirect the User, too.

github: Same as for twitter and facebook, the expected *provider_id* is the username and there will be a *goto*-url returned to opened by the App for the User to authenticate against.

Errors

401 - target_unconfirmed: If the given *target* hasn't been successfully confirmed yet. You might want to resend the register-code for your jabber-id

1.3.3 Verify

URL: /v1/verify/<hashkey>

Supported Methods: GET

Optional Parameters: code=XXXX

In order to verify a previously registered entry of *hashkey*. Commonly called with a *code* parameter containing the code send through the providing channel. Also used as the callback endpoint for some OAuth2-based providers like *Github* or *Facebook*.

If the given code matches the code in the system, a successful return confirms the authentication happened and deleted the reference to the previously provided *provider_id*. The returning JSON would like like this:

```
{
 "confirmed": true
}
```

But can also contain some further information given by the provider.

Errors

404 - not_found: There has no pending entry been found for the hashkey. In case you have recently registered the account and reused the hashkey, this most commonly means the account has been verified meanwhile. You might want to use *confirm* to confirm this though.

400 - missing_code: The provider requires you to provide a *code* parameter but none has been given. Please prompt the User to enter the code and try again.

400 - faulty_code: The code provided doesn't match the one last send through the channel. Most commonly because another resend-registration-request has been triggered meanwhile and superseeded the used code. Please prompt the User that the code was wrong and that they might wait for another to arrive to retry or check for typing mistakes.

401 - OAuthException: A problem occured in the OAuth interna of the *twitter*, *facebook* or *github* providers. Please look at the provided *message* for further information. Most likely happens because of faulty redirection, failure in authentication with the service or missing parameters during development.

400 - wrong_user: In case of OAuth, the provided *provider_id* doesn't match with the one presented by the OAuth provider after authentication. Please prompt this to the User and explain them the requested account data didn't match their input. They might simply picked the wrong account when logging in.

500 - internal server error: Please provider a bug report something went terribly wrong on the server side.

1.3.4 Contact

URL: /v1/contact/

Supported Methods: POST

Post data (JSON-formatted String):

```
{
  "target": "xmpp@example.com",
  "contact_info": [
      {"protocol": "phone", "id": "+00112345678"},
      {"protocol": "email", "id": "hunter@jobs.com"},
      {"protocol": "phone", "id": "+4912345"}
      ],
  "contacts": [
      "397ee3ee893ba686b8f228078803ce34911b35c8bf15a7986310de1225589fe13706a3242376da92c144a0e38e4693
      "e5d20f91694fde312aeb9e784178c8bd8a386d8c2789dfed7dc14a35fb8ea88fd0a1583a0a98b80058e8c9e6d7c8a
      "abce880ed2d448abffa8efa8939d8e15625ad16ff2330d97388f32fee480d799b9753e1d2f362c7deb1f7ea83bfbb
      ]
  }
```

A post request asking the service to send a contact request to all provided hashes in *contacts*. A core feature of the service: takes all provided contact_info, makes them into hashes and matches them against their database entry. Only those matching the given target will be forwarded to the clients.

Secondly, looks up the targets of the specified clients and sends every one found the *target* as well as the list of confirmed contact infos for matching against their own address book. See *contact-matching*. The send will be done via XMPP using the itseme-extension in messages, see *itseme-contact-extension*.

If everything goes well and independant of the number of matches found (even if none), the returning response will be:

```
{
  "status": "requests_send"
}
```

Errors

400 - json_decode_error: The provided JSON couldn't be read. Please make sure you've encoded it properly.

401 - target_unconfirmed: If the given *target* hasn't been successfully confirmed yet. You might want to resend the register-code for the jabber-id.

400 - key_error: You forgot to provide at least one of the mandatory keys. Please consult 'message' to learn which one.

400 - value_error: At least one of the provided parameters didn't match the required criteria. Please consult 'message' for further information.

400 - insufficient_contact_info: None of the provided contact info could be confirmed against the service, no requests will be send. At least one contact details needs to be provided and needs to be confirmed on the platform otherwise we won't forward the request. If you see this error, you might want to ask the user again to authenticate against at least one service.

413 - too_many_requested: Per request only not more than 100 hashes shall be provided. Please check before sending and batch the requests accordingly, preferably after the previous succeded to not run into the rate limit.

1.3.5 Confirm

URL: /v1/confirm/<hashkey>/<target>

Supported Methods: GET

Confirm that the given *hashkey* links to the provided *target*. Returns a json if and only if the hashkey was found and the stored target matches the provided one:

```
{"confirmed": true}
```

In any other case (also if the key wasn't found or isn't verified yet) it will return a false value for the check:

```
{"confirmed": false}
```

Errors

There are no custom errors.

1.3.6 Confirm many

URL: /v1/confirm

Supported Methods: POST

Post Data:

```
{
  "hashes": [
    "397ee3ee893ba686b8f228078803ce34911b35c8bf15a7986310de1225589fe13706a3242376da92c144a0e38e4693a
    "e5d20f91694fde312aeb9e784178c8bd8a386d8c2789dfed7dc14a35fb8ea88fd0a1583a0a98b80058e8c9e6d7c8acd
  ],
  "target": "jid@example.com"
 }
```

Confirm that the given *hashes* all link to the provided *target*. Returns a json if so:

```
{"confirmed": true}
```

If at least one hashkey wasn't found, isn't confirmed yet or doesn't match the provided target, it will return a falsy value instead:

```
{"confirmed": false}
```

Errors

There are no custom errors.

1.3.7 Version

URL: /version

Supported Methods: GET

Returns a json-response confirming the server version. Mainly for debugging purposes:

```
{"version": "0.9-rc2"}
```

1.4 XMPP Extensions

In order to allow an automatic, user-less flow for verification and confirmation of apps, the It-se.me XMPP client sends slightly modified versions of messages, containing extra information for seamless integration. The extensions are done to the normal message-stanza and are wrapped into the *it-se.me*-namespace and are of either *verify* or *contact*

1.4.1 Sending codes

When the it-se.me service sends the verification code to the provided jabber id, it does this through the message (for display) but also attaches the verify-tag (in the *it-se.me*-namespace). This tag contains to tags: *code* and *hash*. The *hash* contains the generated SHA512-Sum while *code* contains the provided code:

```
<message from='itseme@jabber.ccc.de/XXXXX' to='xmmp@example.com' type='chat' xml:lang='en'>
 <body>It-se, you xmmp@example.com? Verify with: 3991</body>
 <verify xmlns='it-se.me:verify'>
     <code>3991</code>
     <hash>fb1a59cd5e804c48d89f4267956518ac47b7d0e645fd92fcee85cbd9dbebb3eef4ad12a2d5baaa216ade5c698
 </verify>
</message>
```

When the app receives such a message and has a registration with the given hash pending, it may automatically call the verify-url providing the under *code*-lying content as a parameter to automatically authenticate the service. In which case it is also recommended, to prevent the message from being shown to the user.

Note: Though you there is a jabber account provided here, this might be subject to change and the implementation should not rely on it coming from this account.

1.4.2 Contacts Request

When an authenticated user sends a *contact*-request to the API and the service finds matching hashes, it then forwards the _confirmed_ contact details alongside the provided jabber-id to the jabber account stored for that hash via an xmpp message, too. Those messages contain a *contact*-tag of the *it-se.me*-namespace, similar to this:

```
<message from='itseme@jabber.ccc.de/XXXXX' to='xmmp@example.com' type='chat' xml:lang='en'>
   <body>Hey there,
     xmpp2@example.com asked whether they are allowed to get in
     contact with you. They provided the following contact
     details for you to confirm their identity:
       * phone: +123456789
       * email: xmpp2@example.com
   </body>
   <contact xmlns='it-se.me:contact'>
         <info>[{"provider": "phone", "id": "+123456789",
         "confirmed": true}, {"provider": "email", "id":
         "xmpp2@example.com", "confirmed": true}]</info>
         <jid>random@example.com</jid>
   </contact>
 </message>
```

In which the *info*-tag contains an encoded JSON-Array with a list of hashes in the form of:

```
{
  "provider": "email",
  "id": "xmpp2@example.com",
  "confirmed": true
}
```

And the provided-*jid*-tag contains the corresponding target account requesting the contact. As discussed in :ref:concept, the App may use these provided information to look up the account in the Users address book and (if it chooses to) add the account to their roster. It is generally recommended to also subscribe to their presence and immediatly send them a similar request containing the *contact*-tag with your own verified contact details because up to this point the other party has no information about ones account information.

Note: If the App receives a *contact* request from any other account than the specified one, it should always check back the provided contact information with the it-se.me service using the confirm-many API call. As this fails if at least one isn't confirmed the App may not send any contact information on its own that hasn't been verfified before hand.

1.5 The Technologies

This project, as soo many of our time, stands of the shoulders of giants. In particular, it is too mention that the server itself is build and run in Python using the amazing Flask-Web-Framework.

For background queueing and processing, it uses Celery, which itself relies on Redis for queueing and reporting. The Data itself is stored in the Document-NoSQL-Database Couchdb. And deployment is done using Docker.

Frontend-Side (for Mario, the little helper), the project relies on React and jQuery.

Thanks to all these amazing projects

1.6 Common questions

1.6.1 Tech questions

Why aren't you using salts? Salts make everything more secure!

Though this might be technically right, in our current use case we are using the hash for lookup and confirmation. If we used a salt to generate them then people wouldn't be able to generate them anylonger on their own unless you'd make them either server- wide or generally distribute them, in any case rendering it kinda useless.

It would be as if you'd put a salt into a hash of a file. Either you redistribute it, too, or now one will be able to confirm it.

If you aren't storing my data, how is Mario (the frontend) able to show it?

Mario_ indeed knows about this information but the It-se.me API-Server doesn't. Mario stores it on your System via LocalStorage for convienience.

1.6.2 API questions

Is there a reference implementation?

Yes. The **'frontend'_** itself uses the same API (and nothing else). Please look at the **'Mario Source code'_** for a complete registration flow implementation. A second also contact-flow-complete version is in the works for **SecureChat_**.

I found a bug in the API, where can I report it?

Please do report them on our Github Issue Tracker.