3rd Party Integration Guide for Whigi

This guide aims at providing the suitable and necessary information for anyone to interface smoothly to Whigi. You will hear a bit about the data model, and Whigi restore as such an example. Then we will introduce you to best practices and detailed explanation of interfacing.

# How Whigi is built

Whigi is built with the idea in mind that users should regain control over their personal data. As such, in the Whigi world, all encryptions and decryptions take place in the browser. We know, this can be quite lengthy, but this is the ultimate proof we can give to end users.

The Whigi server provides bare services for creating accounts, storing, retrieving and sharing pieces of data. It *expects* this data to be encrypted, but it cannot even be sure about that.

## Encryption scheme

All users have the following keys, stored as explained:

* The password. The password is not really a key in itself, but the sha256(password + salt) is used as key to encrypt using AES256 the master key. The password is never stored anywhere, except if a scheme involving Whigi restore is used, and the matching for authentication is done on sha256(sha256(password) + salt) to not have to store this;
* The master key. Randomly generated AES256 key, this key is stored encrypted as explained above in the user table. A logged in user can retrieve it and decrypt it. It is used for encrypting and decrypting the user’s own copy of a data, and for his private RSA key.
* A public RSA key. 4096 bits, please. This key is used indirectly for encrypting the copy of a data shared with a remote user, called now and later a “vault”.
* A private RSA key. Encrypted with the master key so that Whigi does not know it, it allows to decrypt shared data.

This introduced, we must stress out the fact that vaults are not directly encrypted using the RSA key pair, but a unique AES256 key is generated, encrypted using the RSA public key and shipped along. This AES key is used to encrypt the plain data, for obvious performances reasons.

## Database

The first figure introduces the database. We will explain in more detail all relevant fields so that it becomes clear what developers had in mind while writing this.

### User table

The user table focuses on maintaining a small set of data associated to a user, all what is not essential is considered as a regular data.

* The \_id is the unique identifier, the plain old username
* Is\_company is a field we update manually to a confidence level to display when 3rd parties require grants. You should ask for validation of your account so that we improve your confidence level!
* Company\_info is all your public information. Please note that modifying it will reset your confidence level, so you should update it before asking for validation.
* Puzzle is a mechanism used to make request to the server. It is explained later.
* Encr\_master\_key is the master\_key encrypted as explained above. Refer to “Decrypting keys and data” to know how to extract it.
* Rsa\_pub\_key is the public key of the user, always sent along. It is stored as openssl public key export format.
* Rsa\_pri\_key is the private key, encrypted. Refer to “Decrypting keys and data” to know how to extract it.
* Oauth is an array of tokens, id of the grantee and folder to which access is allowed.
* Salt is the typical salt used for making rainbow tables less effective.
* Password is actually sha256(shaa256(password) + salt), against which authentication is done.
* Data in an object storing where an information can be found, and who it is shared with, refer to “Path naming” to know how data\_name is created.
* Shared\_with\_me in an object indexed by the id of the sharer, then by data name, that points to a vault id where this share can be found.

### Token table

The token table is used to store authentication tokens. These tokens are NOT OAuth tokens, are rather act as session cookies, can be cleared in any fashion by the user for logging out, and making the browser not remember his password.

## Path naming

In Whigi, all data are accessed using a path, often referred to as data\_name. The paths have the following naming conventions: the root folder is the empty string, and accessing a data at the root is the name itself. If there are folders in between, they are prefixed and separated with a forward slash. For instance, to access the data “name” from the folder “profile” of the folder “usual”, refer to it as “usual/profile/name”. When accessing shared data, vaults, you have access to a subset of the file system of the sharer, therefore the same path is used, but the id of the sharer must be given.

## Dated field conventions

Fields can be dated or not in Whigi. A dated field stored new values from the time of change when updated, rather than forgetting the past. The convention for discriminating them is the following:

* A non-timed value is encrypted directly, and its is\_dated field is set to false
* A timed value has its is\_dated field set to true, and the string encrypted is actually the stringification of the following JSON object: an array of object that have two keys, a “from” field, which is an epoch since when this value is valid (an epoch is the time of milliseconds since 1/1/1970,0:0AM) and a “value” field which is the typical associated string value.

## Decrypting keys and data

It may seem easy to decrypt data once we know which algorithm has been used to encrypt it and that we have the key. Sadly though, this is not true: algorithms expect some data type, and therefore type conversions happen all the time. In this section, you will find the type conversions that the keys and data undergo, so that you will be able to decrypt pulled data, or to push cleanly encrypted one.

* The master key. It is actually a 32-byte array. It is encrypted directly using AES256 in CTR mode from position 0, and stored as-is, as another 32-byte array. To decrypt it, instantiate an AES256 decrypter in CTR mode from 0, using as key toBytes(sha256(password + salt)). You will find an implementation of toBytes in our code, in the utils/utils module.
* The RSA private key. It is actually a string representing the openssl export of the private key. It is converted to a byte array using aes-js convertStringToBytes function, before being encrypted using AES256 CTR(0) with as key, the master key.
* A personal piece of data is always considered a string, being dated or not. It is encrypted using AES256 CTR(0) with the master key, but must undergo aes-js convertStringToBytes first to be processed. The result is then turned back to a string using our own arr2str that you will find in the app/app.service module of the client. To decrypt a received personal data, you must thus turn it to an array of bytes using str2arr, decrypt it using AES256 CTR(0) with the master key, and apply convertBytesToString on the result.
* A data stored in a vault is encrypted using a temporary AES256 key that is shipped in the vault. It is encrypted/decrypted the same way as a personal data. The temporary AES256 key is encrypted using the RSA public key of the user to be granted access. Because the RSA implementation of JSEncrypt expects string to encrypt, the AES256 key, which is a 32-byte array, is turned to a string using arr2str, then encrypted and the result is shipped. To recover the AES256 key, simply use your RSA private key then apply str2arr on the result.

# A 3rd party example, Whigi restore

Whigi restore is nothing but a 3rd party which we fully trust. The frontend of Whigi has some special functionalities reserved to be used with it, but these are few.

Upon account creation, a user is asked whether he wants his account to be recoverable or not. If he declines the feature, his password is never stored, and its lost means the termination of the account.

However, if he accepts, some data fields are created for him:

* The path “profile/email” that contains the user’s email is created and granted to whigi restore
* The path “profile/recup\_id” that contains a trusted person’s id is created and granted to whigi restore if the safest method is chosen
* The path “keys/pwd/mine1” is created, contains the first half of the user’s password, and is granted to whigi restore.
* The path “keys/pwd/mine2” is created, contains the other part, and is granted either to whigi restore, or another person.

If the user forgets his password, by entering his id, whigi restore will be able to browse his shared directory for the user’s email, and send him a link with his password if it knows both parts. Otherwise, it must browse the other user’s shared repo for his email, and send him a link asking the user to inform whigi restore of his part temporarily. Whigi restore can then concatenate both parts and sent the poor user a mail with his password.

Actually the mail sent redirects him to a traditional password change page, and the frontend issues a password change request to the server. As you should have understood, such an operations required the frontend to send the new password against which to match, but the newer version of the encrypted master key as well.

# Getting started

You should first define what your goals for interfacing are:

* If you want to create a full frontend, refer to the API endpoints definition, how to log in a user, create him a token, and log him out.
* If you want to use Whigi services as a plugin, or just as an authentication mechanism, continue right here.

In both cases, you should not that we have been using AES256 from npm aes-js, and RSA4096 from npm node-rsa. Although those algorithms are supposed to be clearly defined standards which any library should defined the same way, we encourage you to not deviate from those.

# Using Whigi as 3rd party

Whigi can be used for several purposes: it ranges from the sole authentication, to the collection of some user’s data if he accepts, to even full access in read mode.

## Whigi for authentication

If you plan on relying on Whigi for authenticating people, you should first make them create a local account, right? This local account will be minimized to their id, but still exist to remember them later.

Remember that on all those pages, your company public information is displayed. You may thus use them as a 3rd party that is not even a company, but your confidence level from Whigi will be 0. You should undergo some validation from Whigi to make it raise, and have users click OK!

### Account creation

The first flow creates an account: make your dear nobody browse to https://[Whigi]/account/encodeURIComponent([your\_id])/encodeURIComponent([return\_url\_ok])/ encodeURIComponent([return\_url\_deny]). You can include query parameters in your URL’s as they are encoded, but make sure to provide https, or Whigi will deny the action. The user will be prompted with a screen asking him if he wants to create an account. If this is the case, your return\_url\_ok will be browsed, and you will have access to keys/auth/[your\_id] shared by the newly registered user.

### Authentication

Once again, it is as easy as sending your would-be user to https://[whigi]/remote/encodeURIComponent([your\_id])/[challenge]/encodeURIComponent([return\_url]). Your challenge should be letters and digits, and you should record it on your server alongside the user id. You should still use https URL, the user will log in to Whigi, then your return\_url will be browsed with additional query parameters than those you might have supplied: “user”, the id of the user that has logged in, and “response”, your “challenge” encoded with the decrypted data of keys/auth/[your\_id] according to this user. To generate an AES256 key from this data, the toBytes function is applied to it. This can thus be null if the user does not have an account at your side. To authenticate the user, decrypt the data from your vaults, you will retrieve a string, apply toBytes on it to have the AES key and decrypt the response; if this matches your challenge, your user is who he claims to be.

## Request for grant

You need a user’s data, such as his address for sending him his goods, but you do not have it because you use Whigi? No troubles, just make him browse to https://[whigi]/grant/encodeURIComponent([your\_id])/encodeURIComponent([//-separated-list-of-data])/encodeURIComponent([return\_url\_ok])/encodeURIComponent([return\_url\_deny])/expire-epoch. You should browse the publicly available list of data maintained by Whigi admins to see where the data you want is located. The expire\_epoch parameter is a number representing until when you will need this data. Setting it low will usually provide the user more confidence. If you require some fields that are standardized ones by Whigi, the user will be prompted to create them on the fly if he does not possess them yet. You may require more than one data at once. One of the two URL’s given will be browsed to depending on success or not.

## OAuth

Whigi provides OAuth, but using OAuth gives you the user’s master key, thus making you as responsible as a full frontend writer. You cannot use OAuth without being registered in our services, and we deeply restrict what can be done on behalf of the user, as only reads to a predefined folder and data listing are allowed.

To use OAuth, first register as OAuth client to your local Whigi authority. You must provide a for\_id, which can be your Whigi id, as well as a checkback URL while subscribing. This checkback URL should allow the query parameter “token” and return a JSON response with a success field set to true or false if this token is known.

Send your user to https://[whigi]/ oauth/[for\_id]/encodeURIComponent([folder\_name])/[token]/encodeURIComponent([return\_url\_ok])/encodeURIComponent([return\_url\_deny]). The URL’s should be HTTPS, and you can require a folder, or a plain data. If the user grants use access, you ok URL is browsed with query parameters “token”, the OAuth token, and “key\_decryption”, sha256(password + salt). You can use it to decrypt the user’s master key, refer to “Decrypting keys and data” to see how. On deny, the other URL is visited with a “reason” query parameter. To use the OAuth token, as explained in the RFC, use The Authorization HTTP header, with value “Bearer btoa(token)”.

# Communicating with the API

To communicate with the API, always prefer using DNS resolution, as the API is CDNized. You will find below the precise documentation of all the endpoints. One of them is marked with “captcha”, but you cannot replicate the account creation, thus it is irrelevant for you. However, some of them are marked with “puzzle”. This is important.

## Handling puzzles

You want a smooth service, we want smooth users. To make everyone agree, the server responds to long-lived commands only if the user has done some work for him before. This is known as puzzle. Basically, when retrieving the user’s profile, you retrieve his puzzle as well. This is a small random string.

When you are about to call an API endpoint that requires the puzzle, send a query parameter “puzzle” with a string. This string should be such that, when appended to the user’s puzzle, the first four bytes of the sha256 hash are null. You can try to find a suitable string by any means, but iterating over the numbers is a good choice.

If you give a faulty puzzle, or if the user’s puzzle has changed since your last change because he is for instance connected to several frontends, don’t worry, the endpoints that require a puzzle issue it back upon error, with a 412 HTTP code. Just make sure to always record the newest puzzle whenever you see one, then you can retry if you get a 412.

## Endpoints

Below is a description of the endpoints callable from any third party. All posted data should be posted in JSON format, and the responses are also JSON formatted. If puzzle is required, the response always also includes the user’s puzzle, and can be 412 if the puzzle check fails. The response can also always be 401 if authentication is needed but is not successful.

The response contains an error field if the response is not 2XX that describes in the user’s browser language the error.

All endpoints begin with /api/v<any>/ for now, but maybe some features will require a specific version at a time.

To authenticate, use HTTP basic auth, send the Authorization header with value “Basic b64encode(id + ‘:’ + sha256(password))”. When using a token, send “Bearer b64encode(token)”.

### peek/<id>

Retrieves whether a user exists.

Method: GET, Auth: NO, Puzzle: NO

Post data: -

Response: 200, 404, 500

### user/<id>

Retrieves public info of a user.

Method: GET, Auth: YES, Puzzle: YES

Post data: -

Response: 200, 404, 500

### profile

Retrieves own profile, but not data not shared data.

Method: GET, Auth: YES, Puzzle: NO

Post data: -

Response: 200

### profile/info

Updates public info. Resets confidence level.

Method: POST, Auth: YES, Puzzle: NO

Post data: Any JSON object

Response: 200, 500

### profile/data

Retrieves list of own data.

Method: GET, Auth: YES, Puzzle: NO

Post data: -

Response: 200, 500

### profile/data/new

Posts new own data.

Method: POST, Auth: YES, Puzzle: YES

Post data: {“name”: “string of the data name”, “encr\_data”: “string of the encrypted data as should be”, “is\_dated”: “boolean if data is dated”}

Response: 201, 500

### profile/update

Updates user password.

Method: POST, Auth: YES, Puzzle: NO

Post data: {“new\_password”: “sha256 hash of the new password”, “encr\_master\_key”: “updated encryption of the master key, as explained above”}

Response: 200, 500

### profile/token/new

Creates a token for authenticating.

Method: POST, Auth: YES, Puzzle: YES

Post data: {“is\_eternal”: “boolean setting whether this token expires if not used for 30minutes”}

Response: 201, 500

### profile/token?token=<token>

Removes authentication token. Only remove the one specified if one is specified.

Method: DELETE, Auth: YES, Puzzle: NO

Post data: -

Response: 200, 403, 500

### oauth/new

Creates an OAuth token.

Method: POST, Auth: YES, Puzzle: NO

Post data: {“for\_id”: “granted organization”, “prefix”: “data\_name or folder to which access is given”, “token”: “the token that will be created”}

Response: 201, 403, 500

### oauth/<token>

Deletes an OAuth token.

Method: DELETE, Auth: YES, Puzzle: NO

Post data: -

Response: 200, 403, 500

### data/<id>

Retrieves a data.

Method: GET, Auth: YES, Puzzle: NO

Post data: -

Response: 200, 404, 500

### data/<data\_name>

Removes a personal data.

Method: DELETE, Auth: YES, Puzzle: NO

Post data: -

Response: 200, 500

### vault/new

Posts new own data.

Method: POST, Auth: YES, Puzzle: YES

Post data: {“data\_name”: “data\_name that is shared”, “shared\_to\_id”: “id of person with who to share”, “aes\_crtypted\_shared\_pub”: “AES256 key encrypted as explained above”, “data\_crypted\_aes”, “data crypted as explained above”, “expire\_epoch”: “epoch in milliseconds when this vault is no longer valid. <=0 never expires”}

Response: 201, 404, 500

### vault/<id>

Deletes a created vault.

Method: DELETE, Auth: YES, Puzzle: NO

Post data: -

Response: 200, 403, 404, 500

### vault/<id>

Retrieves a vault.

Method: GET, Auth: YES, Puzzle: NO

Post data: -

Response: 200, 403, 404, 417(expired), 500

### vault/time/<id>

Retrieves info about the access to a vault. Namely, last access and expiration time.

Method: GET, Auth: YES, Puzzle: NO

Post data: -

Response: 200, 403, 404, 417(expired), 500

## Typical login process on a frontend

A typical login flow involves several API calls: the first one should be the creation of a token if refused, the supplied credentials were invalid. This allows to use the token for the next commands and make the browser not remember the credentials.

Then, you should get your profile to decrypt the user’s master key. The last step would often be to get the data list of the user to be able to start processing own and shared data.