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

Auteur : Mathonet, Grégoire
Promoteur(s) : Leduc, Guy
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité spécialisée en "computer systems and networks"
Année académique : 2016-2017
URI/URL : http://hdl.handle.net/2268.2/2602

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

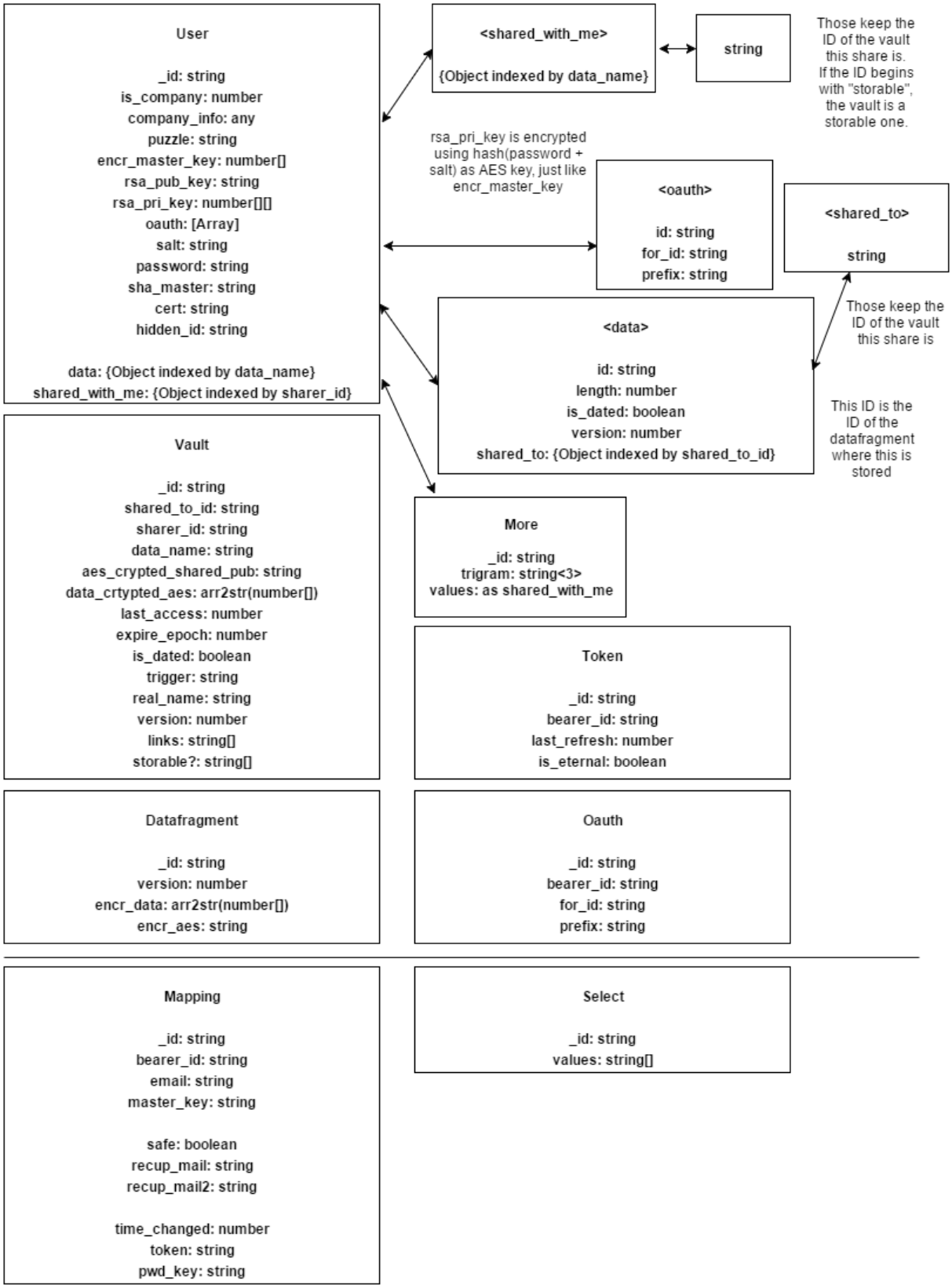
All users have the following keys, stored as explained:

- 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. 1024 bits to 4096 bits according to the Whigi implemntation, 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. Users can have several such keys if coming from merged accounts.

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
_id: string
is_company: number
company_info: any
puzzle: string
encr_master_key: number[]
rsa_pub_key: string
rsa_pri_key: number[][]
oauth: [Array]
salt: string
password: string
sha_master: string
cert: string
hidden_id: string
data: {Object indexed by data_name}
shared_with_me: {Object indexed by sharer_id}
<shared_with_me>
{Object indexed by data_name}
string
Those keep the ID of the vault this share is. If the ID begins with "storable", the vault is a storable one.
rsa_pri_key is encrypted using hash(password + salt) as AES key, just like encr_master_key
<oauth>
id: string
for_id: string
prefix: string
<shared_to>
string
Those keep the ID of the vault this share is
<data>
id: string
length: number
is_dated: boolean
version: number
shared_to: {Object indexed by shared_to_id}
This ID is the ID of the datafragment where this is stored
Vault
_id: string
shared_to_id: string
sharer_id: string
data_name: string
aes_crypted_shared_pub: string
data_crtypted_aes: arr2str(number[])
last_access: number
expire_epoch: number
is_dated: boolean
trigger: string
real_name: string
version: number
links: string[]
storable?: string[]
More
_id: string
trigram: string<3>
values: as shared_with_me
Token
_id: string
bearer_id: string
last_refresh: number
is_eternal: boolean
Datafragment
_id: string
version: number
encr_data: arr2str(number[])
encr_aes: string
Oauth
_id: string
bearer_id: string
for_id: string
prefix: string
Mapping
_id: string
bearer_id: string
email: string
master_key: string
safe: boolean
recup_mail: string
recup_mail2: string
time_changed: number
token: string
pwd_key: string
Select
_id: string
values: string[]

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.

More table

The more table is actually stored within the user table. As Mongo documents are limited in size to 16MB, assuming normal user with our ID’s, a user could have about 80K datas or shared datas before running out of space. This table allows to grow much more, by maintaining sub documents when a user reaches 50K shares. At that point, the shares are themselves split by trigrams, so all the shares coming from people whose username begins with aaa, aab, etc, will go into the same More. For 10billion users, this still allows such a batch of people to once again share 50K data.

When a user reaches such weight, it cannot anymore change his username (hum), nor retrieve the whole of its shares in one shot, it must specify which users it wishes to access.

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,that can be slightly modifies for generic data that can exist in several instances. Anyways, the filesystem is browsable in the same fashion..

Dated field conventions

Fields can be dated or not in Whigi. A dated field stores 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(dk), where dk is sha256(password + salt) self hashed either 0 or 600+ times. You can test whether this was the correct number of times because of the sha_master element, which is sha256(sha256(str2arr(master_key))). Have a look at the codes afterwards to help yourself.
- 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. Note that the 32 first bytes of what you recover/encrypt ought to be the SHA256 of the rest. This is done to ensure the use of the good RSA key for merged accounts. Not all third parties need

to support this if you know you will never use merge accounts, but this should be clearly stated.

Code samples

This small paragraph gives an example of how to decrypt the several keys using different languages. The first picture shows the helper functions, whereas the second uses them to produce the master key and the main RSA private key.

Javascript

```
/**
 * Return an array from the first values of a string giving an AES key.
 * @function toBytes
 * @public
 * @param {String} data String.
 * @return {Bytes} Bytes.
 */
toBytes(data: string): number[] {
    function num(e) {
        if(e >= 65)
            return e - 55;
        else
            return e - 48;
    }

    var ret: number[] = [];
    try {
        for(var i = 0; i < 32; i++) {
            ret.push((num(data.charCodeAt(2*i)) * 16 + num(data.charCodeAt(2*i + 1))) % 256);
        }
    } catch(e) {
        return ret;
    }
    return ret;
}

/**
 * Turns an array of nums to a string.
 * @function arr2str
 * @public
 * @param {Number[]} arr Array.
 * @return {String} String.
 */
arr2str(arr: number[]): string {
    var result = '';
    for(var i = 0; i < arr.length; i++) {
        result += String.fromCharCode(arr[i]);
    }
    return result;
}
```

```
/**
 * Decrypts the master key once and for all.
 * @function decryptMaster
 * @public
 */
decryptMaster() {
    try {
        var kd = localStorage.getItem('key_decryption');
        for(var i = 0; window.sha256(window.sha256(this.arr2str(this.master_key || []) || '')) != this.profile.sha_master; i++) {
            var key = this.toBytes(kd);
            var decrypter = new window.aesjs.ModeOfOperation.ctr(key, new window.aesjs.Counter(0));
            this.master_key = decrypter.decrypt(this.profile.encr_master_key);
            kd = window.sha256(kd);
            if(i == 1) {
                //We set to 1 or far more...
                for(var j = 0; j < 600; j++)
                    kd = window.sha256(kd);
            }
        }
        for(var i = 0; i < this.profile.rsa_pri_key.length; i++) {
            decrypter = new window.aesjs.ModeOfOperation.ctr(this.master_key, new window.aesjs.Counter(0));
            this.rsa_key[i] = window.aesjs.util.convertBytesToString(decrypter.decrypt(this.profile.rsa_pri_key[i]));
        }
    } catch(e) {
        this.notif.alert(this.translate.instant('error'), this.translate.instant('noKey'));
    }
}
```

PHP

```php
public static function pkcs1unpad2($b, $bits = 1024) {
    $i = 0;
    $n = ($bits + 7) >> 3;
    $l = strlen($b);
    while($i < $l && ord($b[$i]) == 0)
        ++$i;
    if(ord($b[$i]) != 2)
        return null;
    ++$i;
    while(ord($b[$i]) != 0)
        if(++$i >= $l)
            return null;
    $ret = '';
    while(++$i < $l) {
        $c = ord($b[$i]) & 255;
        if($c < 128) {
            $ret .= chr($c);
        } elseif(($c > 191) && ($c < 224)) {
            $ret .= chr((($c & 31) << 6) | (ord($b[$i+1]) & 63));
            ++$i;
        } else {
            $ret .= chr((($c & 15) << 12) | ((ord($b[$i+1]) & 63) << 6) | (ord($b[$i+2]) & 63));
            $i += 2;
        }
    }
    return $ret;
}

//Used for tuning keys
public static function toBytes($str) {
    $ret = array();
    for($i = 0; $i < 32; $i++) {
        array_push($ret, (num(ord(substr($str, 2*$i, 1))) * 16 + num(ord(substr($str, 2*$i + 1, 1)))) % 256);
    }
    return $ret;
}

public static function isoUTF($str) {
    $ret = '';
    $l = strlen($str);
    for($i = 0; $i < $l; $i++) {
        $code = mb_convert_encoding('&#' . ord($str[$i]) . ';', 'utf-8', 'HTML-ENTITIES');
        for($j = 0; $j < strlen($code); $j++) {
            $ret .= $code[$j];
        }
    }
    return $ret;
}
```

```php
$mk = '';
$dk = hash('sha256', get_option('whigi_whigi_secret') . $result_obj['salt']);
for($i = 0; $result_obj['sha_master'] != hash('sha256', hash('sha256', WHIGI::isoUTF($mk))); $i++) {
    $mk = @openssl_decrypt(implode(array_map("chr", $result_obj['encr_master_key'])), 'AES-256-CTR', implode(array_map("chr", WHIGI::toBytes($dk))), true);
    $dk = hash('sha256', $dk);
    if($i == 1) {
        //We set to 1 or far more...
        for($j = 0; $j < 600; $j++)
            $dk = hash('sha256', $dk);
    }
}
update_option('whigi_master_key', base64_encode($mk), true);
update_option('whigi_rsa_pub_key', $result_obj['rsa_pub_key']);
$new = @openssl_decrypt(implode(array_map("chr", $result_obj['rsa_pri_key'][0])), 'AES-256-CTR', $mk, true);
if(isset($new) || get_option('whigi_rsa_pri_key') == '')
    update_option('whigi_rsa_pri_key', $new, true);
```

Python

```
def pkcs1unpad2(b, bits = 1024):
  i = 0
  n = (bits + 7) >> 3
  l = len(b)
  while i < l and ord(b[i]) is 0:
    i += 1
  if ord(b[i]) != 2:
    return None
  i += 1
  while ord(b[i]) is not 0:
    i += 1
    if i >= l:
      return None
  ret = ""
  while i < l - 1:
    i += 1
    c = ord(b[i]) & 255
    if c < 128:
      ret += chr(c)
    elif c > 191 and c < 224:
      ret += chr(((c & 31) << 6) | (ord(b[i+1]) & 63))
      i += 1
    else:
      ret += chr(((c & 15) << 12) | ((ord(b[i+1]) & 63) << 6) | (ord(b[i+2]) & 63))
      i += 2
  return ret

def num(e):
  if e >= 65:
    return e - 55
  else:
    return e - 48
def to_bytes(msg):
  str = bytearray(msg, "utf-8")
  ret = bytearray()
  for i in range(32):
    ret.append((num(str[2*i]) * 16 + num(str[2*i + 1])) % 256)
  return ret

def isoUTF8(mm):
  import sys
  sys.setdefaultencoding('iso-8859-1')
  mm = bytearray(mm)
  msg = bytearray()
  for m in mm:
      val = bytearray(unichr(m), 'utf-8')
      for v in val:
          msg.append(v)
  return msg
```

```
def ba2h(ba):
  ret = hashlib.sha256(isoUTF8(ba)).hexdigest()
  return hashlib.sha256(ret).hexdigest()

def decrypt_key():
  h = httplib2.Http(".cache")
  resp, content = h.request(settings.WHIGI_HOST + "/api/v1/profile", "GET",
      headers={'Authorization': 'Basic ' + b64encode(settings.WHIGI_USER_ID + ':' + hashlib.sha256(settings.WHIGI_USER_PWD).hexdigest())})
  c = json.loads(content)
  if "error" in c:
    settings.WHIGI_RSA_PRI_KEY = None
  else:
    settings.WHIGI_SHA_MASTER = c['sha_master']
    dk = hashlib.sha256(settings.WHIGI_USER_PWD + c['salt']).hexdigest()
    i = 0
    while ba2h(settings.WHIGI_MK) != str(settings.WHIGI_SHA_MASTER):
      enc = to_bytes(dk)
      settings.WHIGI_MK = pyaes.AESModeOfOperationCTR(enc, counter=pyaes.Counter(initial_value=0)).decrypt("".join(map(chr, c["encr_master_key"])))
      dk = hashlib.sha256(dk).hexdigest()
      if i == 0:
        for j in range(600):
          dk = hashlib.sha256(dk).hexdigest()
      i += 1
    settings.WHIGI_RSA_PRI_KEY = pyaes.AESModeOfOperationCTR(settings.WHIGI_MK, counter=pyaes.Counter(initial_value=0)).decrypt("".join(map(chr, c["rsa_pri_key"][0])))
```

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/restore” 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 send 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 operation requires 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 note that we have been using AES256 from npm aes-js, and RSA from npm node-rsa. Although those algorithms are supposed to be clearly defined standards which any library should define 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, and maybe have local knowledge of the user, such as orders, etc.

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!

If you make the user browse to a request link (namely, account creation and grant request that he has already allowed, the URL OK will be browsed directly, providing a seamless login experience, although this might be an account creation on your side).

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])/true. 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. Please note that account is the only such endpoint that allows chaining requests, that is, if you provide as return URL a URL that contains another HTTPS one, it will re-encode the contained one so that the request goes fine. This can be used for registration and logging in in a single flow. By using the URL as described in "Request for grant", you can in a single step create an account and register vaults. If you want to always include the account creation by facility, just issue "-" as data list to tell Whigi to not read any data. If you issue as last parameter to this URL the word "flow", the return URL ok is assumed to be hosted by Whigi and parsed accordingly, making the transition even faster. The word "false" is used for granting without account, and any other word uses a regular return URL.

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 "r64", your "challenge" encoded with the decrypted data of keys/auth/[your_id] according to this user, and obviously base64 encoded. The "response" is also available, which are the plain bytes values, '-' separated along with "hidden_id", the true user identity. 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 base64-decoded 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]/account/encodeURIComponent([your_id])/encodeURIComponent([return_url_ok])/encodeURIComponent([return_url_deny])/false/encodeURIComponent([::-separated-list-of-data])/expire-epoch/encodeURIComponent([trigger_url]). You should browse the publicly available list of data maintained by Whigi admins to see where the data you want is located. If you want to request a whole folder, request its name with the trailing slash, but doing so, you won't be able to set where the data will be saved. 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. The trigger URL is defined for the vaults that will be created, and is fetched when the vault contents are modified over HTTPS. You are not forced to give a trigger URL.

To register a data under a special name, you can specify the name under which you would like an instantiable data to be registered (defaults to the name of the generic itself) by appending //full/path/to/shared to the requested generic. Using this, you can set some data as optional by prepending * to its name, and request several times the same data for instance for billing, shipping, etc. You can also prepend a number and a pipe (|) to override more mandatory fields within the shares, this number will behave as a binary mask for fields you'd like to add mandatory, indexed from 1 (0 will add no new mandatory fields, 1 the first, etc). Note that those can only guarantee client side what is given, and you should always check what you have received.

Whigi Smart services

Whigi can now also be used on android. As such, you can interface with it inside your application nearly the same way as you do with your website. Note that on mobile, grant and login flows are separated, because they are already fast. The grant flow consists in creating the following code:

```
Intent i = new Intent("com.envict.whigi.GrantActivity");

i.putExtra("...", ...);

startActivityForResult(i, MyActivity.WHIGI_CALLBACK);
```

The extras you must add are nearly the same as for the web: id_to is you ID, with_account is a boolean to create an account, trigger is a URL you want to be triggered on vault change, expire_epoch is the epoch when grant expires, data_list in an array of strings that represent the data you need. This list can be formatted as with the website parameters, with save name and non mandatory fields. Please note that on mobile devices, the user cannot create new data that has pre-requirements, and must therefore already have the data you require to complete the action. If you receive this error, you should use a webview.

Your activity that launches Whigi must override the onActivityResult method where it will receive a standard Activity.RESULT_CANCELED or Activity.RESULT_OK code.

For authentication, the principle is the exact same but you only need to specify your ID, and the challenge as extras to intent to the class RemoteActivity.

Whigi WebView services

Because you cannot be sure that your user has the Whigi application installed on his smartphone, and because Whigi does not provide native iOS app at the moment, you can have the support required for your grant/login flows with WebViews.

Please note that you should first try to use the native app, and fallback to WebView only if this failed, for best user experience.

- On Android, create a WebView as described here: https://developer.android.com/guide/webapps/webview.html and load it with a URL that looks like the ones web services use. Please note that the return URL's will be browsed, so you should probably set them to about:blank. You have to define three callback hooks in an injected object called "Android": "deny" which is called if grant is not possible, "ok" which is called if grant is successful, and "remote" which is called with a string "response:user:hidden_id" which concatenates answers a web service might receive.
- On iOS, create a WKWebView as described here: http://www.kinderas.com/technology/2014/6/15/wkwebview-and-javascript-in-ios-8-using-swift and do the same as for Android. You do not have to specify a name for the injected global object, as you cannot do so.

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 can use OAuth without being registered in our services, but for company security, you should first register at our side, to forbid creation of tickets on your behalf without your server acknowledging, and we deeply restrict what can be done on behalf of the user, as only reads and writes to prefixed folder are allowed.

To use OAuth, you can 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. This would allow for greater security when issuing tickets.

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)”.

You can also directly create and remove tickets from the main client, in this case you will have to give the ticket to another Whigi account, though.

Synchronization

Please refer to the author’s master thesis to know more about how synchronization and caching of user data can be done.

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 three bytes of the sha256 hash are the ASCII character 0. 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 418 if authentication is needed but is not successful (used so that the browser does not display an auth popup).

The responses contain an error field if the response is not 2XX that describes in the user’s browser language the error. The documentation here provides the expected JSON format if the request is successful, and does not document the fact that requests may embed a “puzzle” field with the freshest user’s puzzle.

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))”, or “Basic b64encode(id + ‘:’ + sha256(arr2str(master_key)))”. When using a token, send “Bearer b64encode(token)”. The server only has doubled hashed versions of your password and master key, so it cannot impersonate you.

generics.json, generics_paths.json, selects/<select>.json, schemas/<schema>.json, helps/<help>.json

Retrieves the definitions.

Method: GET, Auth: NO, Puzzle: NO

Post data: -

Response: 200

JSON: Generics definition file, or selectable options, or transition schema.

Other endpoints

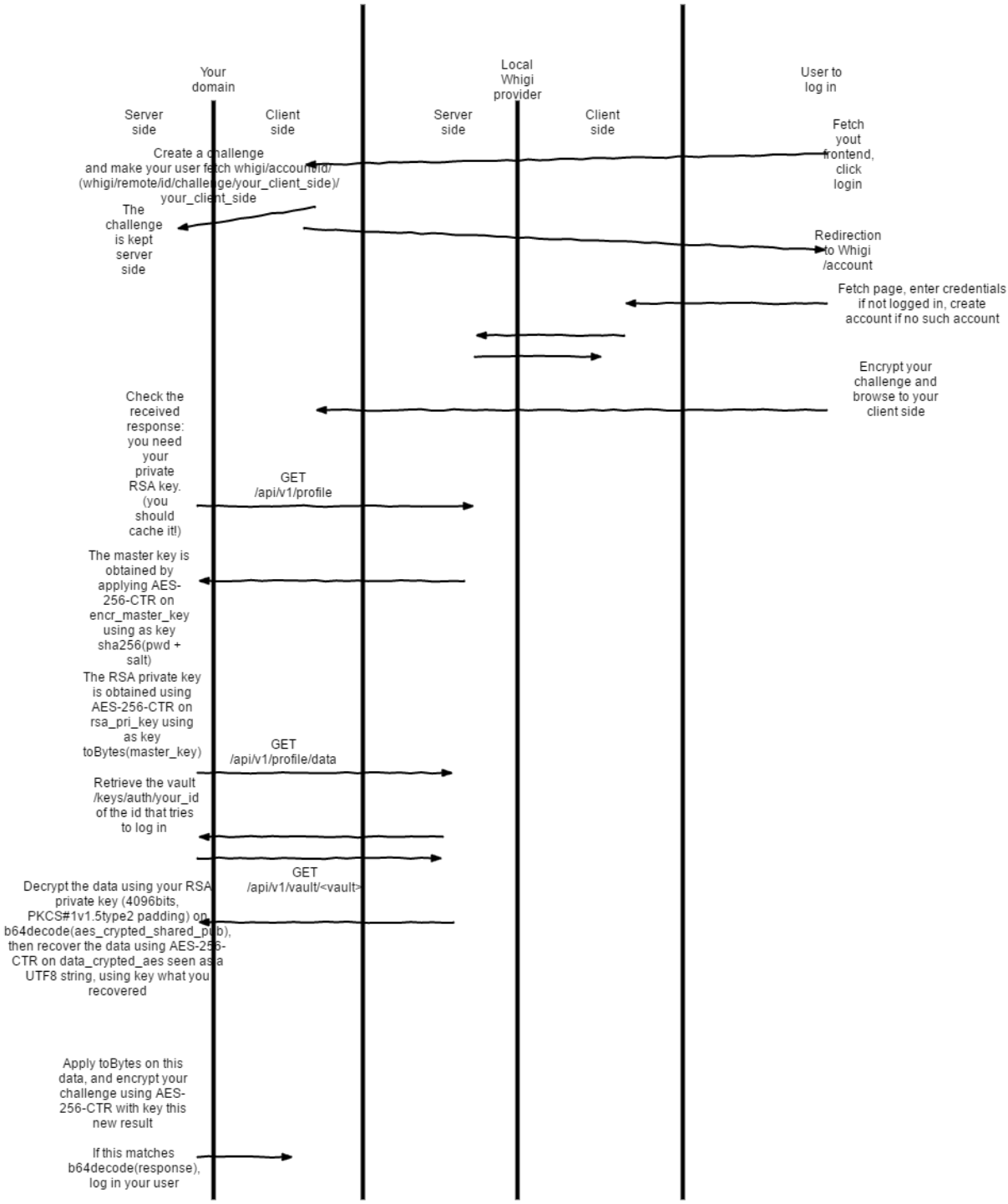
Please review the HTML documentation in the Whigi git repository, “doc” folder, “api.html” file.

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.

The following diagram shows a typical login flow, using the combined facility of return URL to create an account on the fly if none was there yet. The frontend and backend requests are clearly separated, and the decryption algorithms used at each step are also underlined.


Your domain
Server side
Client side
Local Whigi provider
Server side
Client side
User to log in
Fetch yout frontend, click login
Create a challenge and make your user fetch whigi/accountId/(whigi/remote/id/challenge/your_client_side)/your_client_side
The challenge is kept server side
Redirection to Whigi /account
Fetch page, enter credentials if not logged in, create account if no such account
Encrypt your challenge and browse to your client side
Check the received response: you need your private RSA key. (you should cache it!)
GET /api/v1/profile
The master key is obtained by applying AES-256-CTR on encr_master_key using as key sha256(pwd + salt)
The RSA private key is obtained using AES-256-CTR on rsa_pri_key using as key toBytes(master_key)
GET /api/v1/profile/data
Retrieve the vault /keys/auth/your_id of the id that tries to log in
GET /api/v1/vault/<vault>
Decrypt the data using your RSA private key (4096bits, PKCS#1v1.5type2 padding) on b64decode(aes_crypted_shared_pub), then recover the data using AES-256-CTR on data_crypted_aes seen as a UTF8 string, using key what you recovered
Apply toBytes on this data, and encrypt your challenge using AES-256-CTR with key this new result
If this matches b64decode(response), log in your user