TITLE: Integration with *OpenID flow- developers guide*

AUTHOR: Artur Wojnar <artur.wojnar@cyberuslabs.com>
CREATION DATE: 04.10.2019
REVIEWER: Aaron Neugebauer <aaron.neugebauer@cyberuslabs.com>
LAST REVIEW: 05.11.2019

Table of Contents

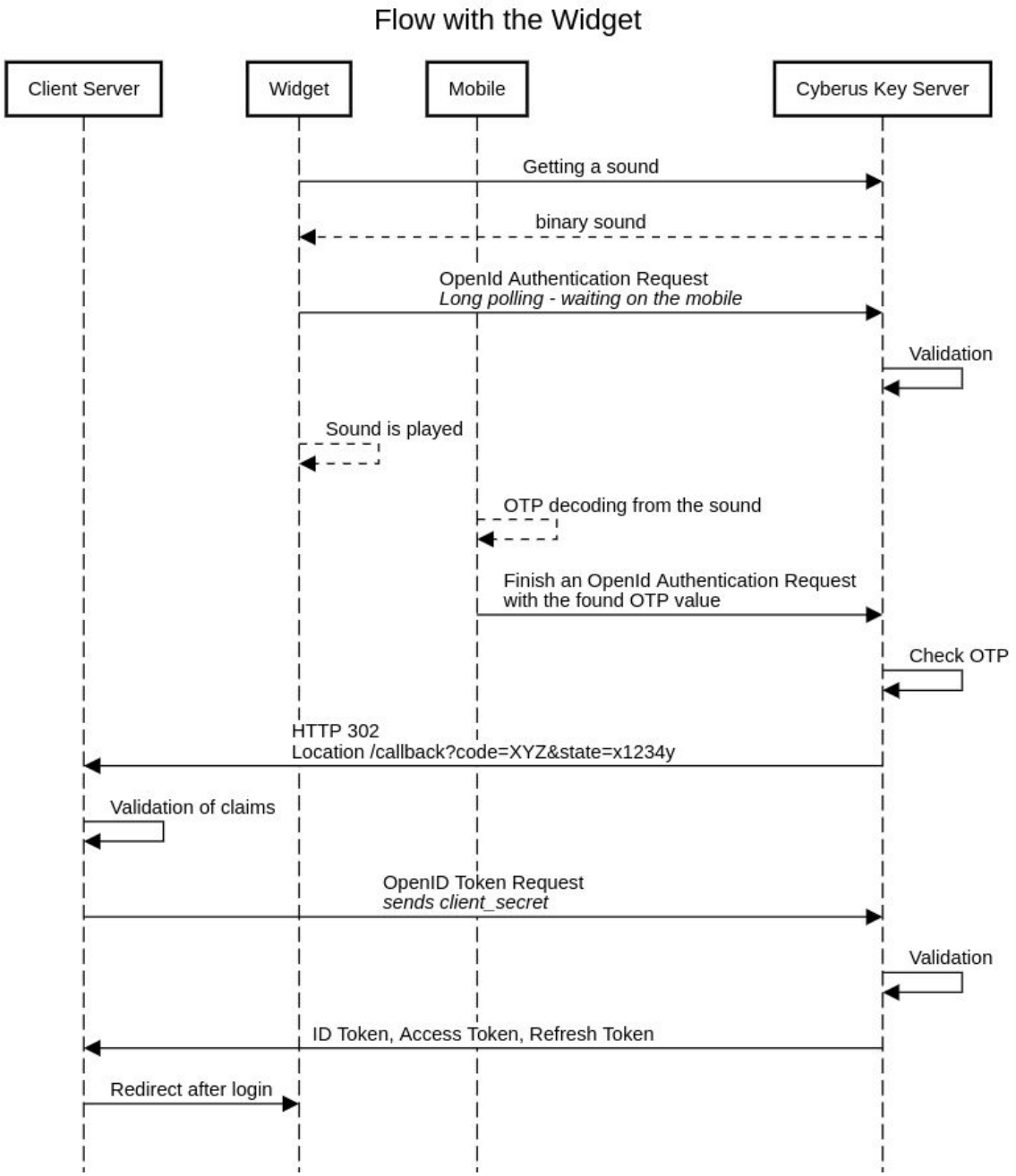
Requirements for integrating with Cyberus Key

This document focuses on integration with a web application, but for the other front channels (e.g. desktop applications) it is possible to integrate with Cyberus Key Server.

1. A client account.
   We will create an account for you and you'll get two credential keys - client ID (public) and client secret key. The last one should be protected and be known only to your backend.
2. A front channel (web site) with the Cyberus Key Widget embedded or your own implementation (see the Custom front channel implementation section).
3. A back channel (web server) with a defined endpoint that is triggered when an Authentication Request ends, either with success or failure.

General flow with the Widget

Cyberus Key supports only OpenID's most secure *Code Flow*. There are two communication channels required for integration: front channel (your site's frontend to Cyberus Key Server) and back channel (your site's backend to Cyberus Key Server).



The flow is a bit more complicated compared to a typical OpenID one, because of the physical layer - a sonic sound that is decoded by the mobile application. This entails some additional work, which is fully automated by the Cyberus Key SDK and Cyberus Key API.
Internally, the Widget first calls the Cyberus Key Server to start a short-lived Cyberus Key session. The server creates an OTP (One-Time Password), then generates a sound with the encoded OTP and returns it to the Widget. The Widget is then able to start the OpenID

Authentication Request, which will end when the mobile app decodes and sends the OTP to the Server (or the session times out).

Side note: From the Token Request you will get ID Token and Access Token. The first one provides information about the user and is part of OpenID. The latter comes from OAuth 2 (which OpenID is based on) and is used for authorization.

API documentation

Our API documentation supports OpenAPI format, so you can view it using Swagger Editor or other tools.
Requests and responses coming in and out of the Cyberus Key Server are validated according to this specification. To ease your work, please feel free to auto-generate code based on the API documentation.

Cyberus Key API V2:
https://app.swaggerhub.com/apis-docs/CyberusLabs/cyberus-key-api-v2/2.0.0

Client Admin AP (API V1):
https://app.swaggerhub.com/apis-docs/CyberusLabs/auth-server/0.1#/app

OpenID: https://openid.net/specs/openid-connect-core-1_0.html#CodeFlowAuth

Example integration

Client keys and transaction data

We will supply you with a Client ID, Client secret and an admin account (with an email and password).

We will also create transaction types based on your requirements. Transaction types define how transactions will be handled in certain situations and are used when setting up redirections (more on those below).

- Is biometric confirmation required?
  For any transaction involving sensitive user information, we recommend requiring the user to confirm with a biometric (currently their fingerprint).
- Should transactions fail on geolocation mismatch?
  The widget and mobile application will independently detect the user's location. If the distance between the two is too large, the transaction will fail. This setting increases security by ensuring the Widget and mobile application are near each other, but also increases transaction failures due to inaccuracies in GPS and IP geolocalization.

You can list existing transaction types with the Client Admin API - please check the documentation.

Integrating your users with Cyberus Key users

Syncing authenticated Cyberus Key users with one of your existing users can be done either with the user's email address or a custom ID. Because Cyberus Key users register through the mobile application using their email (which requires a confirmation), the easiest option is to use an email address.

- Email address
  After a user has authenticated with Cyberus Key, your backend will get the user data (including email) and you'll be able to get the user based on their email.
- Custom Identifier
  If you don't want to or can't rely on emails, then you can connect your users and Cyberus Key users with your custom user identifier. This requires using Cyberus Key Mobile SDK and Client Admin API.
  The first gives you the possibility to adapt your mobile application to our functionalities. The Mobile SDK allows you to create a user without an email but with a custom

identifier.
The latter gives you a chance to register your users in Cyberus Key - the mobile application needs an *application token* and *hash token* to get registered. You will get these tokens after creating a user through the Cyberus Key Admin API.
How you will pass these tokens to the mobile application depends on what is the most suitable for your requirements. Remember, that application token and especially hash token should never be compromised - encrypt these values and delete them as soon as the Mobile SDK is registered. Compromising them could cause a major security breach.

Creating a redirection

Part of the presented endpoints are still V1 and will be deprecated in the future.
In case of doubt ask us to create a redirection for you.

In the Cyberus Key documentation, you will find the description of the Client Admin API.
To use these API you need to confirm your identity:

```
curl 'https://production-api.cyberuskey.com/admin/auth' -H 'Cache-Control: no-cache' -H 'Content-Type: text/plain;charset=UTF-8' --data '{"client_key": "L8unKHwyfhzpG79ZtOCZKBJHbiBNSBjd","email": "john.smith@company-inc.com","password": "AFu5ZHa}q)#k"}'
```

In the response you will get a token you can use to create a redirection:

```
curl 'https://production-api.cyberuskey.com/api/v2/redirections' -H 'Authorization: bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOjE1NjgxMTk1NjIsImNsaWVudF9rZXkiOiJMOHVuS0h3eWZoenBHNzladE9DWktCSkhiaUJOU0JqZCIsImVtYWlsIjoiYXJ0dXlud29qbmFyQGN5YmVydXNsYWJzLmNvbSJ9.P1XJyrysVib0VmcwmWUF8SX9H_sPN2G84pNnj7iLAJs' -H 'Content-Type: application/json' --data '{"uri": "https://company-inc.org/biometric-callback", "transaction_type_id": 4}'
```

In the response you will get a Redirection resource.

Important note: Do NOT mix the same redirection (callback) with different transaction types. If you plan to support these both, create separate redirections (URIs) (e.g. /not-safe-callback and /biometric-callback). The Client Admin API won't let you register the same URI for different transaction types.

Embedding the Widget

If for any reason you can't use the Widget, see the Custom front channel implementation section.

Using the Widget Javascript library (https://www.npmjs.com/package/cyberuskey-widget), create a new CyberusKeyWidget instance with your public client ID and a redirect URI. Feel free to adapt the widget to your needs, it's pure HTML and CSS. The only requirement is to keep the logo (on the left) and the text *Login with Cyberus Key*.

The whole OpenID flow is based on full redirection - the Widget will replace the current URL with Authentication Endpoint URL which then redirects to a final URL.

Callback on the back channel

The last requirement is to ensure that the redirection points a defined endpoint on your server. Example below is written in Python and is only for demonstration purposes - don't treat it as production code.

```
class AuthorizationCallbackHandler(tornado.web.RequestHandler):
    def get(self):
        error = self.get_argument('error', None)
        error_description = self.get_argument('error_description', None)

        if error:
            url=f'/login_page&error={error}&error_description={error_description}'
            self.redirect(url)
            return

        data = {
            'grant_type': 'authorization_code',
            'code': self.get_argument('code'),
```

```
    'redirect_uri': f'{options.host_url}demo/authorization-callback'
}
token=base64.b64encode(str.encode(f"{CLIENT_ID}:{CLIENT_SECRET}"))
headers = {
    'Authorization': f'Basic {token.decode("utf-8")}'
}

url = f'{options.auth_server_url}/api/v2/tokens'
token_response = requests.post(url, data=data, headers=headers)
token_body = token_response.json()

id_token = token_body['id_token']
access_token = token_body['access_token']

encoded_payload = re.search('.+\.(.+)\..+', id_token).group(1)
decoded = base64.b64decode(f'{encoded_payload}==').decode('utf-8')
id_data = json.loads(decoded)

if id_data.get('iss') != CLIENT_DOMAIN or id_data.get('aud') != CLIENT_ID:
    self.redirect(f'/login_page&error=Unauthorized')
    return

session_id = self.get_cookie(options.cookie_name)
session = self._session_store.load_session(session_id)

session['user'] = {
    'first_name': id_data.get('given_name'),
    'last_name': id_data.get('family_name'),
    'email': id_data.get('email'),
    'openid_identifier': id_data.get('sub'),
    'exp': id_data.get('exp'),
    'access_token': access_token
}
self._session_store.save_session(session)
self.redirect(f'/user-page')
```

- First, we check whether an error has been passed in. If so, we redirect to a page displaying an error message.
- Next, we construct the Token Request according to the Cyberus Key API documentation, including the Authorization Code that we got in a query parameter. *Remember to handle errors that can be thrown from Token Request (see Cyberus Key docs).*

- After receiving a response from this request we decode the ID Token and Access Token - both are in JWT format.
    - The ID Token provides you with information about the authenticated user. *It is good practice to verify the at_hash claim.*
        - If you used values *email* and *profile* in OpenID's scope claim for Authentication Request (configurable in the Widget) you will get the following properties in your ID Token - *email*, *given_name*, *family_name*, *name*, *last_logged*. The Widget by default requests these and for now Cyberus Key does not restrict access to them.
        - You will always get a unique OpenID-related value of the user - in the sub(ject) claim.
        - If you defined a custom ID for a given user you will get the *user_id* parameter.
    - The Access Token is used to authorize the user in Cyberus Key Server and access user-related resources.
- Next, we check the issuer and auditory claims.
- Lastly, after saving the user's data to the user's session we redirect to a user page.

*You **should** validate the signature of JWT token (ID Token and ID Token) whether it has been encoded by Cyberus Key Server's private RSA key. The public Cyberus Key key is:*

```
-----BEGIN PUBLIC KEY-----
MIGeMA0GCSqGSIb3DQEBAQUAA4GMADCBiAKBgHElKnuERpCN/WcD6RtS9rKhJODM
Idr2Y1yFrS255cOaG10CLwFPhSVK5z4HQv5/VN3GB2Ft+fbu9OZRTqdA4lHo0PB3
Kaj3yByDUdIoTHd4RmZMLSFVHKR0KAW193nI7s/pzeqDL0oFpHnRNZGUqhRbm2UK
fHHDWKkTn/iGIV7XAgMBAAE=
-----END PUBLIC KEY-----
```

CSRF/XSRF mitigation

You should always use a state claim. The state claim is bound to the Authentication Request with its response. There're a few ways of doing this, but the easiest is to use a secure cookie (which is the method specified in the OpenID specs).

The code below (Python with Tornado) sets a new secure cookie (assuming we've set *cookie_secret* while creating a server), which is done on the page request:

```
state = secrets.token_urlsafe(16)
```

```
self.set_secure_cookie('state', state, httponly=True)
```

It's also important to create the cookie as accessible only by HTTP so it is not possible to get it's value via JavaScript.

You will also need to pass the state value to the Widget:

```
const cyberusButton = new CyberusKeyWidget({
    state: window.CyberusKey.STATE
    // other parameters
});
```

The last thing is to compare the value in the Authentication Request's response with the value in the cookie:

```
state = self.get_argument('state', None)
original_state = self.get_secure_cookie('state').decode("utf-8")

if original_state != state:
    self.handleStateError()
    return
```

Claim: Nonce

You should always use a nonce claim. The nonce claim is used to mitigate replay attacks.

On page request:

```
nonce = secrets.token_urlsafe(16)
self.set_secure_cookie('nonce', nonce, httponly=True)
```

Creating the Widget:

```
const cyberusButton = new CyberusKeyWidget({
    nonce: window.CyberusKey.NONCE,
```

```
    // other parameters
});
```

And the most important part - validating the nonce obtained from ID Token:

```
nonce = id_data.get('nonce')
original_nonce = self.get_secure_cookie('nonce').decode("utf-8")

if original_nonce != nonce:
    self.handleNonceError()
    return
```

Claim: at_hash

With this claim ID Token you can check the correctness of the Access Token. The Access Token value is encoded with the SHA-256 algorithm, then the first 128 bits are taken and base64 encoded (respecting the URL encoding).

To validate this value:

```
original_at_hash = id_data.get('at_hash')

if original_at_hash:
    hash_obj = hashlib.sha256()
    hash_obj.update(access_token.encode('utf-8'))
    first128bits = hash_obj.digest()[0:16]
    computed_at_hash = base64.urlsafe_b64encode(first128bits)

    if original_at_hash != computed_at_hash.decode("utf-8"):
        self.handleAtHashError()
        return
```

Claim: c_hash

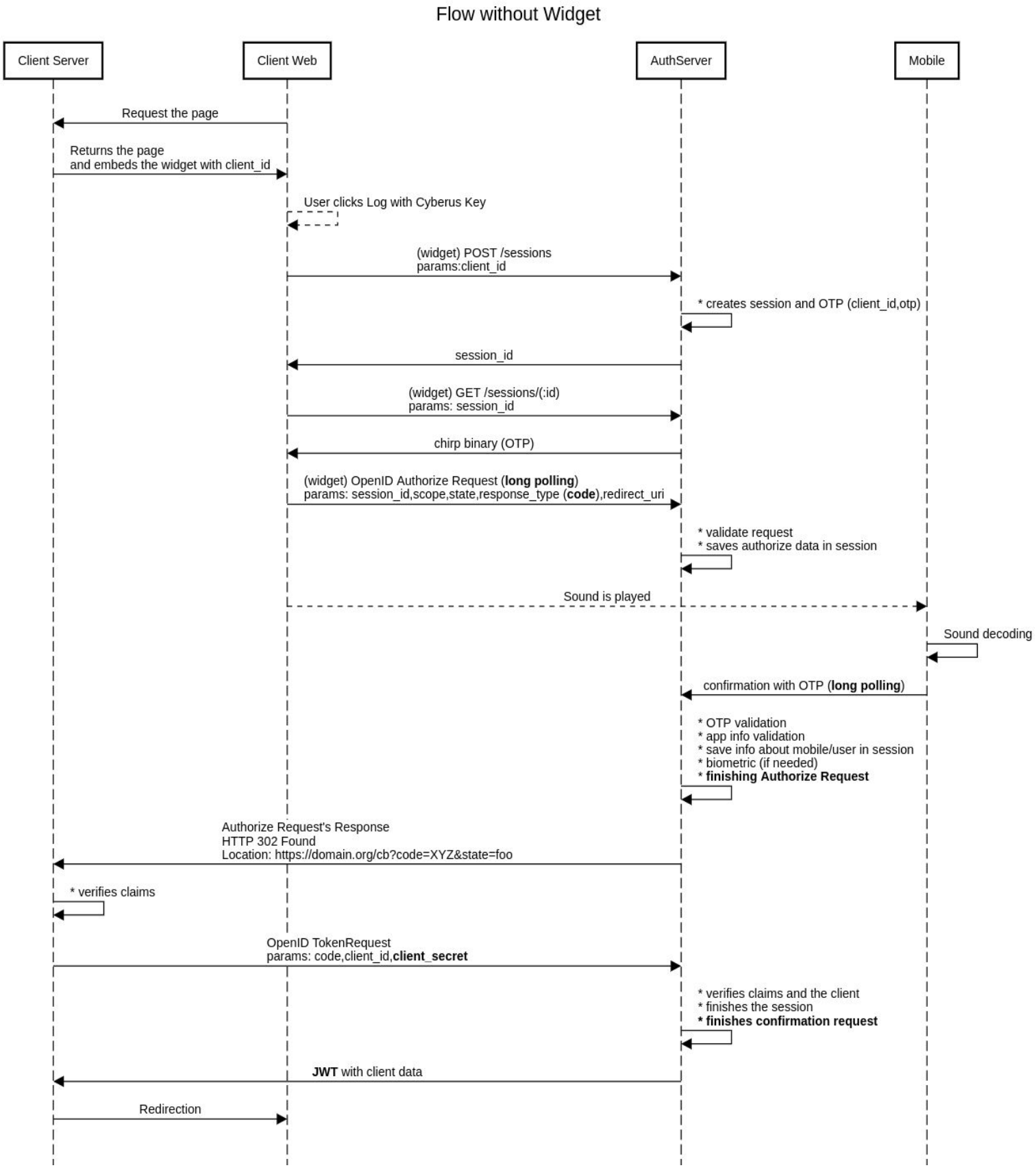
This claim is also contained in the ID Token and it's computed the same way as at_hash, but the value encoded is the Authorization Code.

Custom front channel implementation

If the Widget is not an option for you, you have to use Cyberus Key API (https://www.npmjs.com/package/cyberuskey-sdk) directly.

The process includes:

- Starting short-lived Cyberus Key session
- Obtaining a sound based on the started session
- Preparing and making the Authentication Request


Flow without Widget
Client Server
Client Web
AuthServer
Mobile
Request the page
Returns the page
and embeds the widget with client_id
User clicks Log with Cyberus Key
(widget) POST /sessions
params:client_id
* creates session and OTP (client_id,otp)
session_id
(widget) GET /sessions/(:id)
params: session_id
chirp binary (OTP)
(widget) OpenID Authorize Request (long polling)
params: session_id,scope,state,response_type (code),redirect_uri
* validate request
* saves authorize data in session
Sound is played
Sound decoding
confirmation with OTP (long polling)
* OTP validation
* app info validation
* save info about mobile/user in session
* biometric (if needed)
* finishing Authorize Request
Authorize Request's Response
HTTP 302 Found
Location: https://domain.org/cb?code=XYZ&state=foo
* verifies claims
OpenID TokenRequest
params: code,client_id,client_secret
* verifies claims and the client
* finishes the session
* finishes confirmation request
JWT with client data
Redirection