Authentication

Introduction

There are many security threats an API can face, and how to secure your API is a very crucial topic, so in this article, we'll talk a bit about authentication, that will help us to eliminate user identity threats and enhance our API security.

Authentication along with authorization, are the protective barrier of any API. It makes sure that the right people enter the system and access the right information. In this article we'll talk about the most common ways for authentication in web APIs.

Authentication and Authorization

Authentication and Authorization often gets mixed up, so it's a good idea to clear any confusion before explaining different ways of authentication.

Authentication is the process to verify the identity of the user or the process trying to access a resource, Authentication usually gets done before authorization, and it simply answers the question of "Who are you?"

On the other hand, Authorization is the process of ensuring if this authenticated user has the right to access this resource, authorization limits the access to resources according to the user trying to access it, it provides access control and can be specified through policies and rules.

AUTHENTICATION
Who are you?
Verify the user's identity
AUTHORIZATION
What are you allowed to do?
Determine user permissions

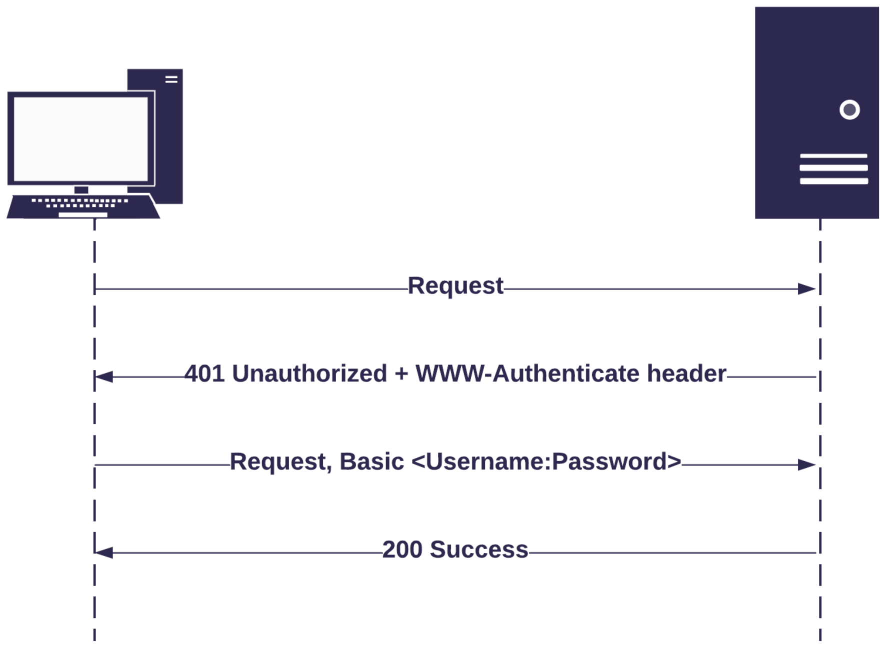
HTTP Authentication Schemes

Basic Authentication

It is the simplest of authentication mechanisms. It is a part of HTTP protocol. It is a challenge response scheme where the server challenges the client to provide necessary information to get authenticated and gain access to resources.

Flow:

1. The client enters their credentials.
2. Concatenate the credentials in a string as <username:password>.
3. Encode it using the base64 algorithm.
4. Attach it to the authorization header and send it along each HTTP Request.
5. The server decodes the credentials string, and verifies if the credentials are valid.
6. The server sends a success response.


Request
401 Unauthorized + WWW-Authenticate header
Request, Basic <Username:Password>
200 Success

Basic authentication is very simple, easy to implement and fast to run, but as you can see, since there's no encryption/decryption process for the credentials, it makes it very easy to decode the encoded credentials and anybody can get them, so it has to be used with other security mechanisms such as HTTPS (HTTP with TLS), where TLS (Transport Security Layer) is a standard technology for keeping an internet connection secure and safeguarding any sensitive data that is being sent between two systems, preventing eavesdropping and tampering the data transferred between two the two parties, ensuring the message integrity and confidentiality, this is done by encrypting the connection between them, this encryption is done between the TCP and HTTP.

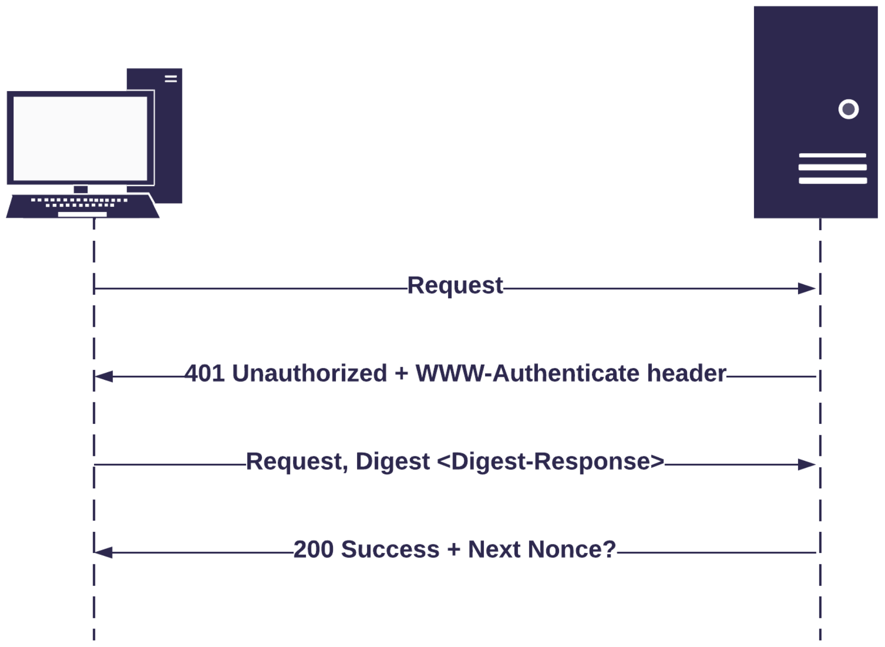
Authorization header for basic authentication can be generated using this tool, also, you can decode the authorization string using this tool.

Digest Authentication

Digest authentication is slightly more secure than basic authentication, instead of sending the credentials in the request header as an encoded string, the client will send them as an encrypted string, this encrypted string contains not only the cerdinitials but other information that will talk about them later.

Flow:

1. Request a server for a resource with no authentication.
2. Server sets the header with certain digest information in the WWW-Authenticate response header.
3. Hash/ encrypt the username and password along with other info and send it to the server.
4. Server reapplies the hashing for the specified user and checks if they’re equal.
5. If they’re equal the server sends 200 code, and other information.


Request
401 Unauthorized + WWW-Authenticate header
Request, Digest <Digest-Response>
200 Success + Next Nonce?

The digest response string will contain the username, password, and nonce encrypted using MD5, there is other information that will be encrypted, these will be specified by the digest information sent in the server www-authentication header.

After authenticating the user, the server could send the next nonce so the user can use it for the next request, this is important so we won't need 2 calls for each request.

Using this way of authentication means that no usernames or passwords are sent to the server in plaintext, so no eavesdropping or sniffing, it prevents replay attacks since it could contain a timestamp for this purpose, and to ensure its uniqueness. Also it prevent phishing, since plain password is never sent to any server, be it the correct server or not

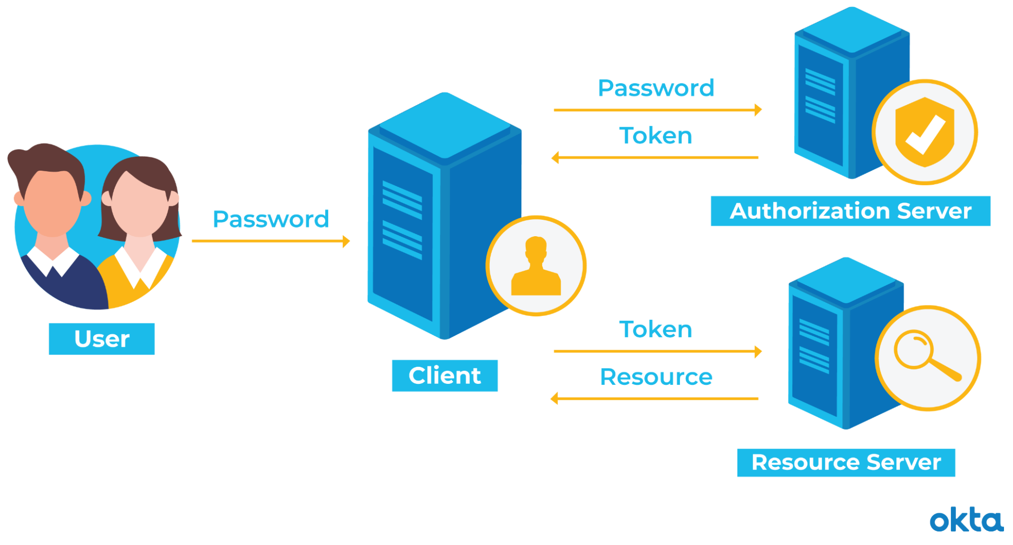
But it has some disadvantages, it's vulnerable to man-in-the-middle attacks, since digest authentication provides no mechanism for clients to verify the server's identity. Also Digest authentication prevents the use of a strong password hash when storing passwords, since either the password, or the digested username, realm and password must be recoverable.

Token Based Authentication

Tokens act like a stamped ticket, when the client has the access token it can send valid requests to the server without adding the ceditals.

Flow:

1. The client sends their credentials to the server.
2. The server verifies the credentials and sends a token for the user.
3. The user stores the token and uses it in future requests until it expires or they logout.
4. In future requests with the token, the server will check if it's valid and hasn't expired yet, then grants access to the user if they have authorization.
5. Users can get refresh token to extend token lifetime.



If the client attempts to visit a different part of the server, the token communicates with the server again. Access is granted or denied based on the token.

Tokens are not stored in the server, this means that they're stateless, also they can reduce database lookups.

The statelessness in token based authentication can cause an issue, that the server cannot revoke a valid token until it expires.

JWT

There are different ways to implement token based authentication, JSON (JavaScript Object Notation) Web Tokens (JWT) is the one of the most common ones.

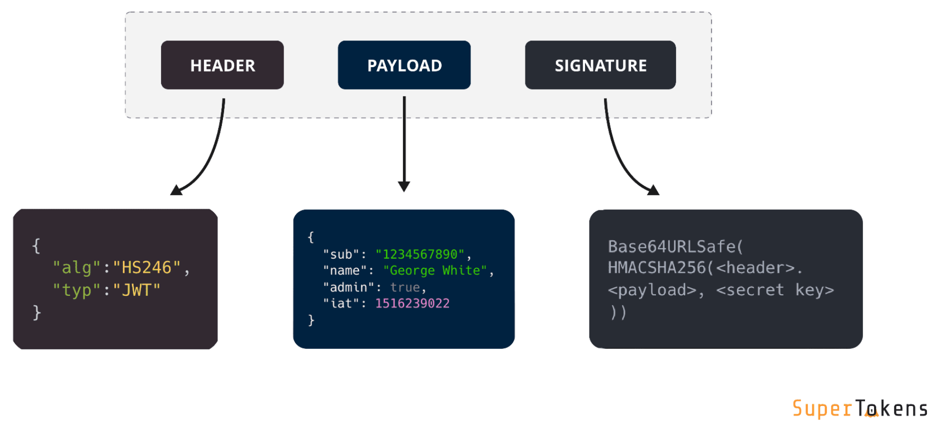
A JSON web token (JWT) is an open standard, the finished product allows for safe, secure communication between two parties.

Data is verified with a digital signature using a secret, this guarantees that the token was not tampered.

The JWT has the following structure

1. Header: contains the signing algorithm and the token type.
2. Payload: contains the claims which typically contains information about the user, but no sensitive or private information, since it will be encoded only.
3. Signature: contains the signatures that will be used to verify that the token is valid.

Structure of a JSON Web Token (JWT)



```
{
  "alg":"HS246",
  "typ":"JWT"
}
```

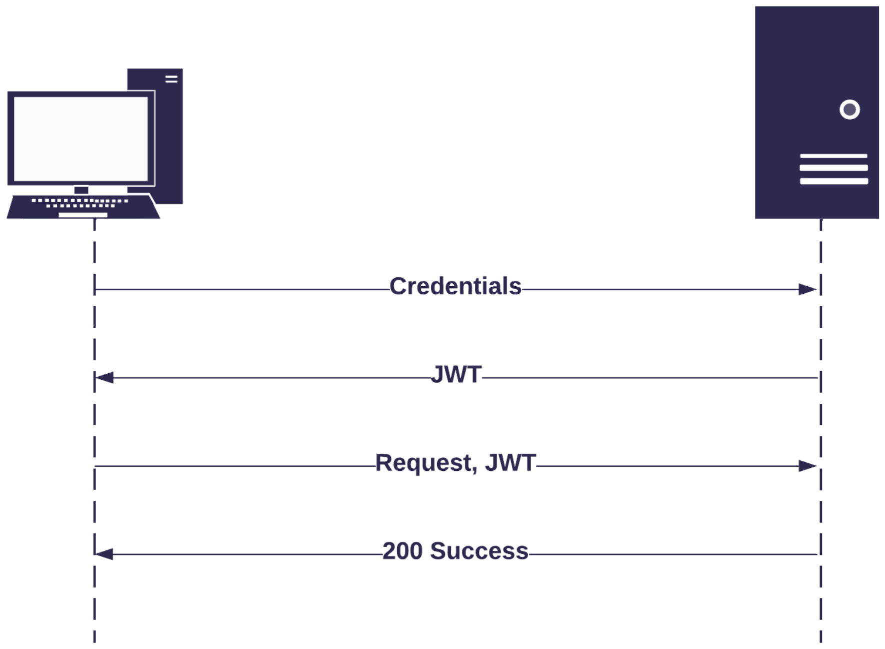
```
{
  "sub": "1234567890",
  "name": "George White",
  "admin": true,
  "iat": 1516239022
}
```

```
Base64URLSafe(
HMACSHA256(<header>.
<payload>, <secret key>
))
```

The encoding of the token is for transmission and you can use this debugger to encode and decode a JWT.

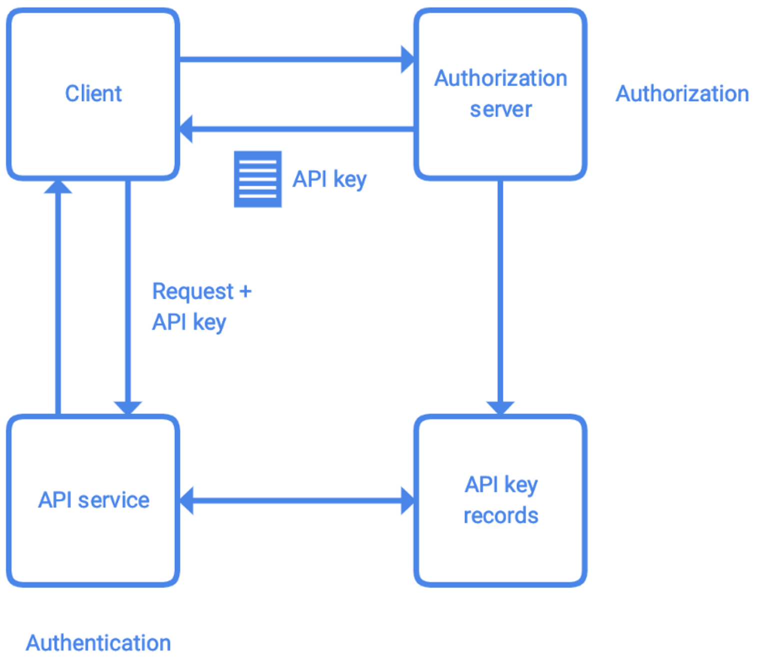
Flow:

1. Client submits a username and password
2. Server validates and returns a JWT token
3. Use the token to allow future requests



Authentication Using API keys

"An API key is a string of characters used to identify and authenticate an application or user who requests the service of an API"



An API key is the standard security mechanism for any application that provides a service to other applications, API keys are easy to use, ease of use is paramount to API usage, all API-first companies want to minimize all friction in using their API products. This includes making access to their APIs easy and secure, which is exactly what API keys are designed to provide. In addition to authentication and identification, API keys provide access rights (authorization).

API keys are used widely in paid services, so if the client has a free plan they will have access to the API but with a limited number of scopes and/or rights, and a client with a premium plan will have more scopes and/or rights, API key can give both types of client access to the API (authentication) and a way to define the access based on their plan (authorization).

API keys are unique for every client, every first time user gets a unique key, sometimes it's generated using some of the user data such as hardware combination and IP data, and other times randomly generated by the server which knows them.

So when a client acquires an API key, they can use it to access API endpoints, the server will verify the key and check if it exists in the database, then comes the authorization, the server will check the client's rights and scopes, to determine how they will use the API services.

Many API keys are sent in the query string as part of the URL, which makes it easier to discover for someone who should not have access to it. A better option is to put the API key in the Authorization header.

There are definitely some valid reasons for using API Keys. As we mentioned earlier, the ease of use. The use of a single identifier is simple, and for some use cases it’s the best solution. Such as when the access for the API is a read access.

The problem is that anyone who makes a request to the API they send their key and the key can be picked up just as easily as any network transmission, and if any point in the entire network is insecure, the entire network is exposed.

It’s advised to use API keys for application identification instead of user authentication, where you can identify the source of the request and apply monitoring and limitations for each type of application accessing your API and block anonymous traffic.

So API keys authenticate the application, not the user. Of course, you could use API keys for user-level authorization, but it’s not well-designed for that – an ecosystem would need to generate and manage API keys for every user or session id, which is an unnecessary burden for a system.

OAuth And OpenID

OAuth

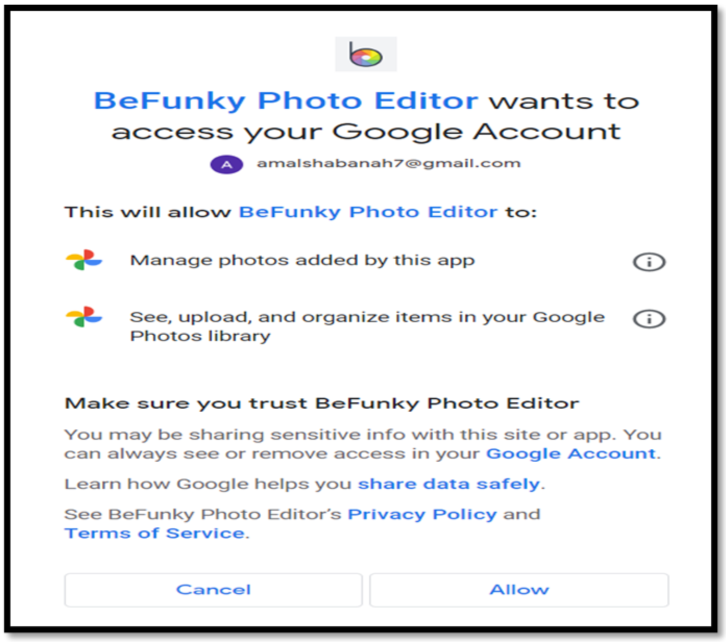
OAuth is an open industry - standard authorization protocol which aims to give limited access to some services to a third party on behalf of the user. In other words, OAuth gives access delegation which is to give a person permission to act on your behalf.

The currently-used version of OAuth right now is OAuth 2.0. Before there was OAuth 1.0 but it had extensive cryptographic requirements, only allowed three flows, and was incapable of scaling. OAuth 2.0 is more efficient and straightforward to implement.

Before moving to how exactly OAuth works. Let's first take a look at the components of the Oauth flow!

1. Actors:
    a. Resource: data (pictures, documents, contacts), services, or any other resource needing access limitations.
    b. Resource Owner: The resource owner is the person who is granting access to some portion of their account.
    c. Client: an application asking for access to some data from the resource owner account.
    d. Authorization server: the system used to grant access to the client.
    e. Resource server: The API that holds the data we need to access (it could be the same as the authorization server)
2. Scopes and consents
    a. Scopes: defines the access that the client is asking for. The scopes contain all the details of what exactly is needed to access. It's written as something like Photos.Read, Photos.Write, etc.

b. Consents: is the message received to the resource owner asking him if he wants to allow access for this client or not. It looks something like this:



BeFunky Photo Editor wants to access your Google Account

amalshabanah7@gmail.com

This will allow BeFunky Photo Editor to:

Manage photos added by this app

See, upload, and organize items in your Google Photos library

Make sure you trust BeFunky Photo Editor

You may be sharing sensitive info with this site or app. You can always see or remove access in your Google Account.

Learn how Google helps you share data safely.

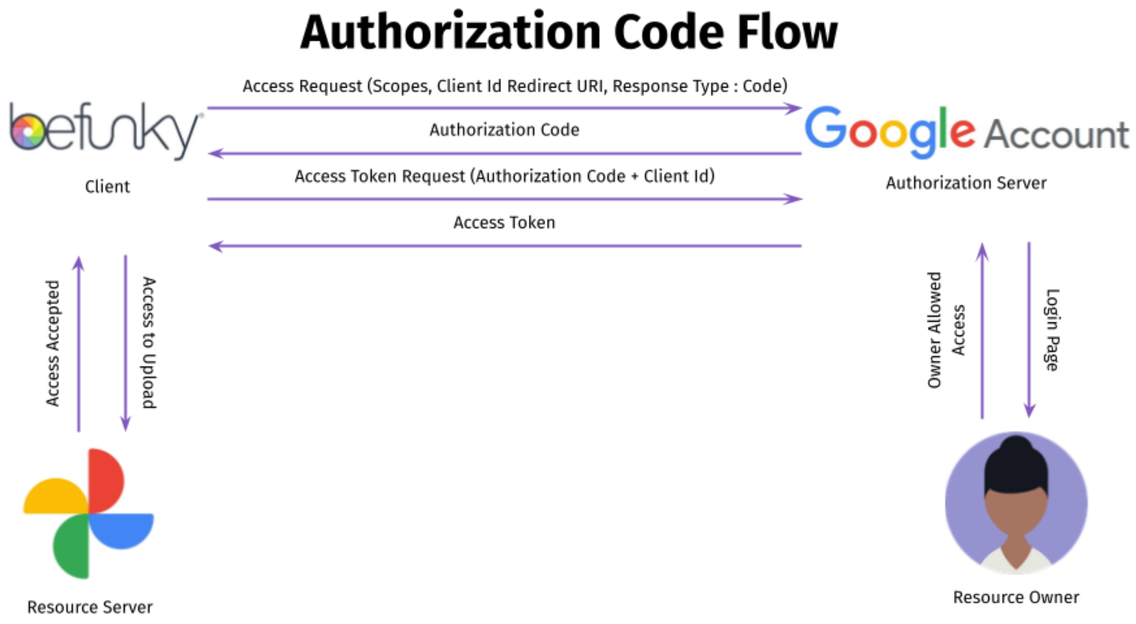
See BeFunky Photo Editor's Privacy Policy and Terms of Service.

c. And as we can notice from the above image, the scopes are defined in the consent message.

3. Access Token:
   a. the token is the piece of that when the client has it can grant the access he needed. containing just enough information to be able to verify a user's identity or authorize them to perform a certain action.
   b. The access token doesn't have to be in a specific format in OAuth. But most often the token used is the JWT token.

OAuth Workflows

1. Authorization Code Flow

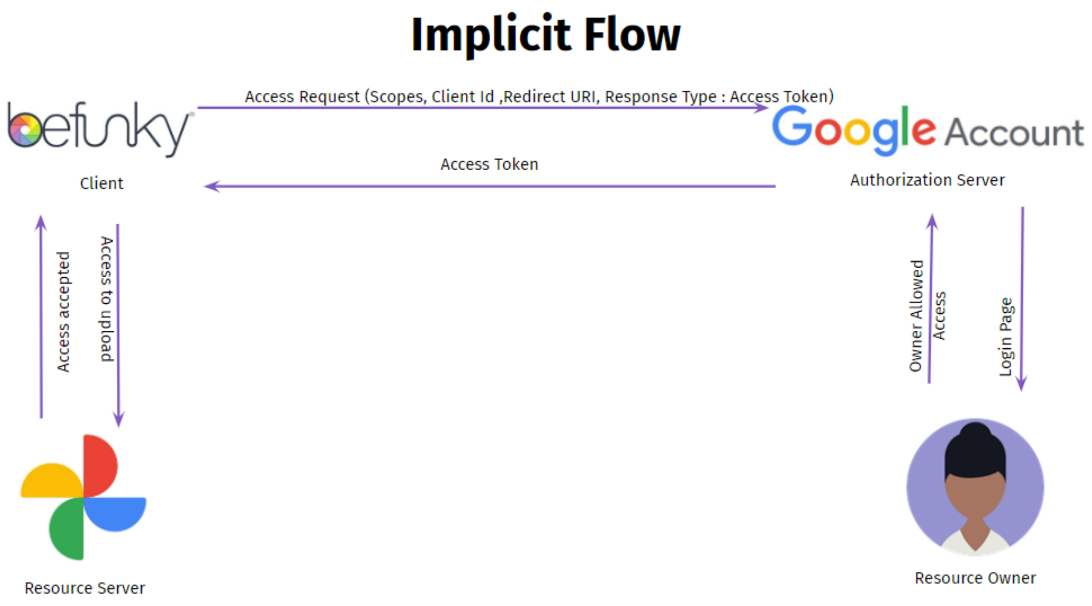


a. The client sends the request to the authorization server. This request contains the client id, and the scope of the access he is requesting for.
b. The Authorization server shows a login page to the resource owner.
c. The resource owner will receive a message that asks if he allows access or not.
d. The Authorization server sends the authorization code to the client.
e. The client receives the authorization code and sends another request containing the authorization code and the client id.
f. The authorization server sends an access token to the client.
g. Now the client has access to the resource specified in the access token.

Now Let's Take Another OAuth workflow,

2. The Implicit flow

   What this workflow differs from the authorization code flow is that instead of first giving the authorization code and then exchanging it with the access token. The access token is given directly.



   But how do we decide if we do the exchange process or not?

   To answer that, Let us take a little about Front Channels and Back Channels

   The Back Channel is a highly secure communication channel. The API request to another server happens using HTTPS.

   The front Channel is a slightly less secure communication channel such as browsers. Anyone can open the developer tool and check out our javascript.

   If we look back to the flow we discussed above, the front channel contains the steps from the beginning until sending the authorization code, notice that data are not secrets and there are no risks if this data had been stolen. Unlike the data in the back channel which contains important secret data like the access token.

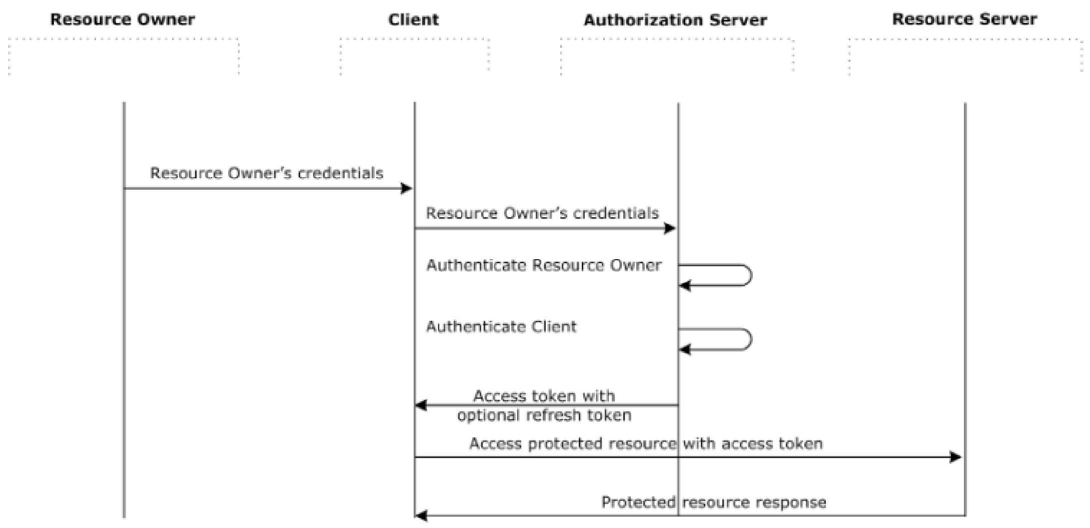
Let's now move to the other workflow of OAuth:

3. Resource owner password credentials flow

   The owner's credentials, such as login and password, are exchanged for an access token in this flow. The user directly provides the app with their credentials, which the app then uses to obtain an access token from a service.

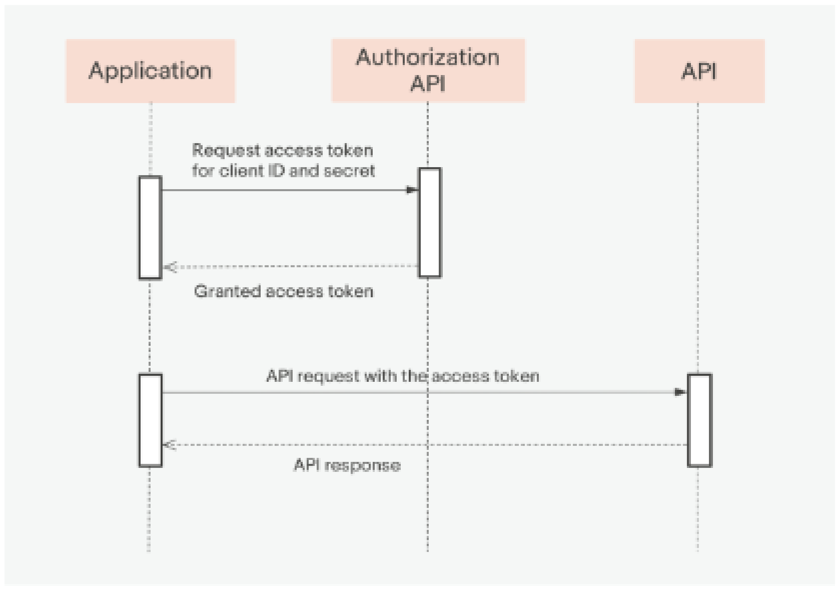
   It is excellent for resource owners who have a strong rapport with their clients. It is not advised for use with third-party apps that have not been formally released by the API provider.

   a. Impersonation: Because anybody may pose as the user to seek the resource, there is no method to verify that the request was made by the owner.
   b. Phishing Attacks - A random client application requests credentials from the user. When a program seeks your Google login and password, instead of forwarding you to your Google account.
   c. The user's credentials might be deliberately revealed to an attacker.
   d. A client application can ask the authorization server for whatever scope it wants. Despite limited scopes, a client program may be able to access user resources without the user's knowledge.


Resource Owner
Client
Authorization Server
Resource Server
Resource Owner's credentials
Resource Owner's credentials
Authenticate Resource Owner
Authenticate Client
Access token with optional refresh token
Access protected resource with access token
Protected resource response

4. Client Credentials Flow

   The Client credentials flow allows a client service to access protected data by using its own credentials rather than impersonating a user. The permission scope in this situation is limited to client-controlled protected resources.


Application
Authorization API
API
Request access token for client ID and secret
Granted access token
API request with the access token
API response

OAuth has been an efficient protocol to deal with access delegation purposes. But what if we wanted to use it further than the access delegation? In the login process for example, here we can notice that we got a problem. As OAuth is an Authorization Protocol and a process like the login is an authentication process, meaning that it needs some users credentials that oauth cannot support.

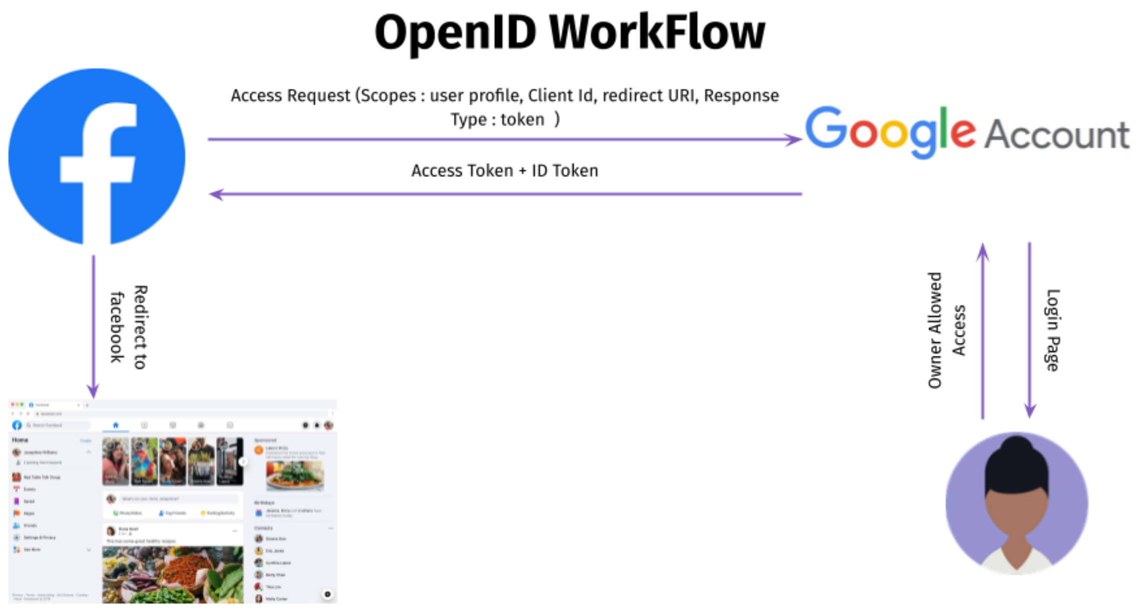
To solve this issue, let’s take the next protocol!

OpenID Connect

OpenID Connect is an extension of OAuth which came to solve what OAuth couldn't. By using

- ID token which adds some user information.
- UserInfo endpoint which is used to access more information.
- Standard Specs.

Meaning instead of just taking the access token from an authorization server, if the server understands OpenID connect we can ask for an ID token and the information at UserInfo, which is pretty much everything we need to know.



Resources

1. HTTP Authentication Schemes
    a. RFC 2617 - Basic and Digest Access Authentication
    b. RFC 7519: JSON Web Token (JWT)
    c. 4 Most Used REST API Authentication Methods
    d. What Is Token-Based Authentication? | Okta
    e. Digest access authentication - Wikipedia
    f. Authentication on the Web (Sessions, Cookies, JWT, localStorage, and more)
2. API Keys
    a. Learn what an API key is, how it's used, and how it provides security | Algolia Blog
    b. Why and when to use API keys | Cloud Endpoints with OpenAPI
3. OAuth and OpenID
    a. Workflow of OAuth 2.0 - GeeksforGeeks
    b. An Illustrated Guide to OAuth and OpenID Connect | Okta Developer
    c. OAuth 2.0 and OpenID Connect (in plain English)
    d. Is the OAuth 2.0 Implicit Flow Dead? | Okta Developer