[Th 9] OAuth

Oauth is a common used authorization framework that enables websites and web applications to request limited access to a user's account on another application.

The basic OAuth process is widely used to integrate third-party functionality that requires access to certain data from a user's account.

Basically it allows users to access multiple services without logging in multiple times.

- in this case the attacker forces the Oauth server to send the access token to the malicious website and not to the legit one

- if the attacker gets the token then he can impersonate the legit user on the Oauth server.

How does OAuth 2.0 work?

Oauth is a token-based protocol that in general is built using OAuth 2.0

It works by defining a series of interactions between three distinct parties, namely a client application, a resource owner, and the OAuth service provider:

- **Client application**
  - The website or web application that wants to access the user's data.
- **Resource owner**
  - The user whose data the client application wants to access.
- **OAuth service provider**
  - The website or application that controls the user's data and access to it. They support OAuth by providing an API for interacting with both an authorization server and a resource server.

The basic idea is:

1. The user chooses the option to log in with their social media account.
   a. The client application then uses the social media site's OAuth service to request access to some data that it can use to identify the user.
   b. This could be the email address that is registered with their account, for example.

2. After receiving an access token, the client application requests this data from the resource server, typically from a dedicated `/userinfo` endpoint.
3. Once it has received the data, the client application uses it in place of a username to log the user in.
    a. The access token that it received from the authorization server is often used instead of a traditional password.

OAuth scopes

For any OAuth grant type, the client application has to specify which data it wants to access and what kind of operations it wants to perform.

It does this using the `scope` parameter of the authorization request it sends to the OAuth service.

As the name of the scope is just an arbitrary text string, the format can vary dramatically between providers.

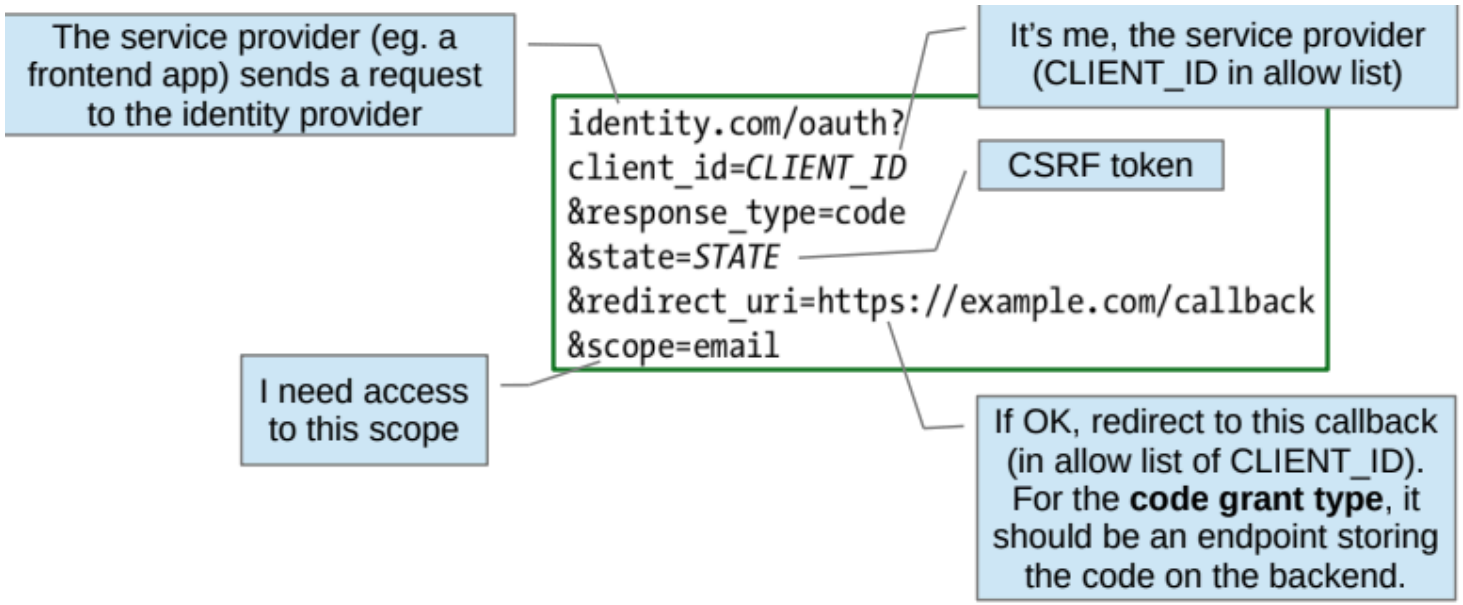
Some even use a full URI as the scope name, similar to a REST API endpoint.

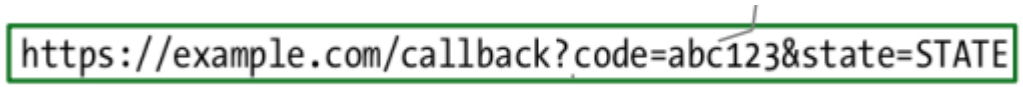
For example, when requesting read access to a user's contact list, the scope name might take any of the following forms depending on the OAuth service being used:

Auto (SQL)

```
scope=contacts
scope=contacts.read
scope=contact-list-r
scope=https://oauth-authorization-server.com/auth/scopes/user/contacts.readonly
```

STEPS OF THE AUTHENTICATION:

1. **The service provider sends a request to the identity provider, this request contains:**
   a. **CLIENT_ID** is the ID of the service provider that is in the allow list of the identity provider
   b. **state**, that is basically a CSRF token
   c. **redirect**, that contains the URI on which the identity provider must send back the code
      i. this is generally the endpoint used to store the code in back-end
   d. **scope**, that is the data we want access of the user logged in the identity provider
   e. 

2. **The identity provider replies on the redirect URI sending an authorization code**
   a. 

   b. **FROM NOW ALL COMMUNICATION IS BACK-END to BACK-END**
3. **The service provider can use the code to obtain from the authorization code to obtain an access token from the identity provider sending a request that contains:**

sending a request that contains:

a. **CLIENT_SECRET**, a secret value binded to the CLIENT_ID and that is known only by the CLIENT back-end and the identity provider
b. **redirect**, that is the URI on which the identity provider will send the access token
    i. it is a endpoint on which the cliend back-end stores the access token
c. **authorization code**, that is the code obtained by the identity provider before
d.

```
identity.com/oauth/token?
client_id=CLIENT_ID
&client_secret=CLIENT_SECRET
&redirect_uri=https://example.com/callback
&authorization_code=abc123
```

The service provider can exchange the authorization code for an **access token**

4. **The Identity provider sends to the client back-end the access token**
    a.

```
{
    "access_token": "z0y9x8w7v6u5",
    "token_type": "Bearer",
    "expires_in": 3600,
    "scope": "openid profile",
    …
}
```

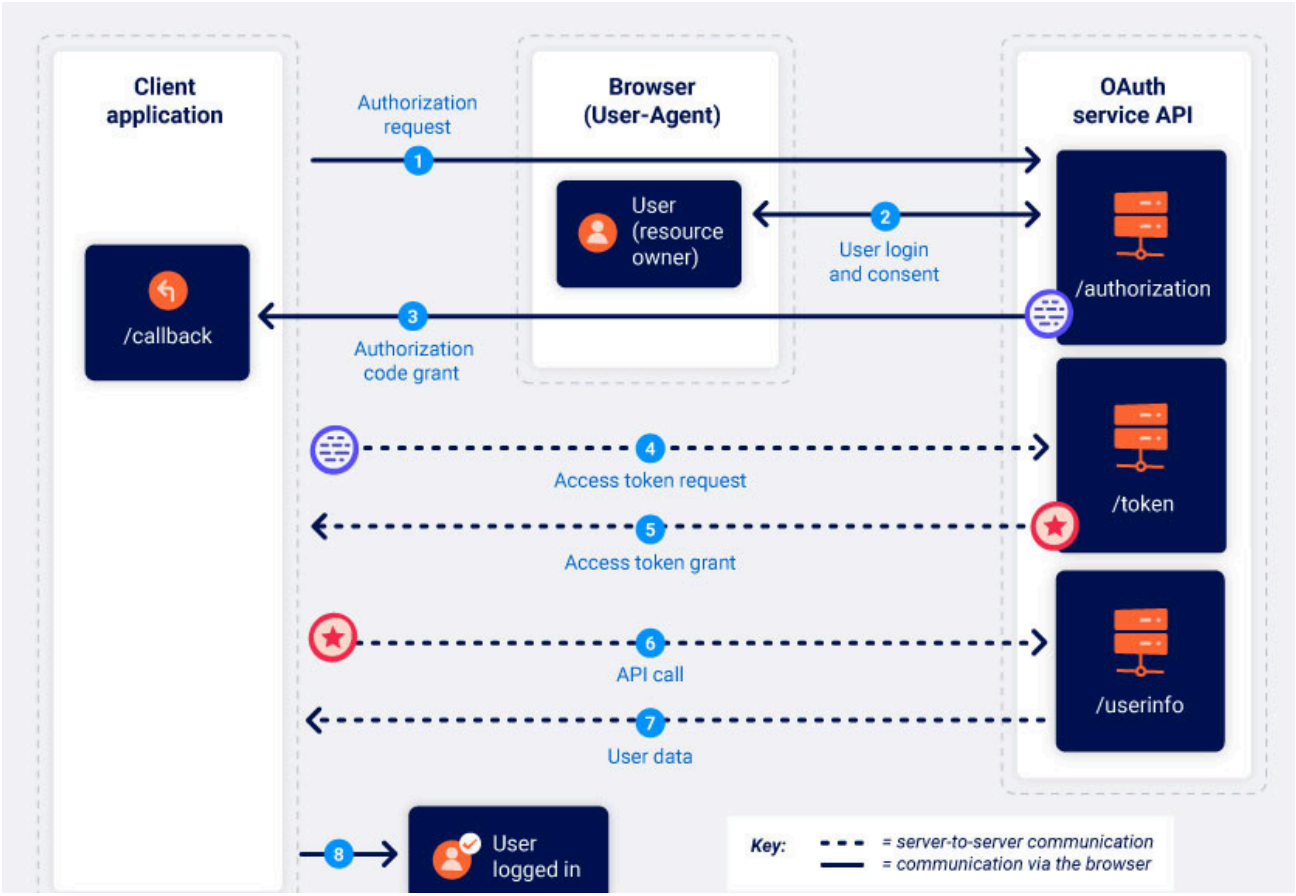
The **access token** must be stored in the backend. Usually there is also a **refresh token**.

    b. we can use the refresh token to ask for a new access token without executing the entire OAuth flow, this because the access token expires
5. **The client can use now the access token to obtain info about the user and to obtain a witness authentication**
    a.

```
GET /userinfo HTTP/1.1
Host: oauth-resource-server.com
Authorization: Bearer z0y9x8w7v6u5
```

NOTE THAT: the CLIENT_SECRET is used to avoid attacks in which the attacker controls the authorization code, this because the authorization code is sent and stored using the browser, so it can be accessed if we have a XSS. The CLIENT_SECRET is stored in the back-end and it is not possible to retrieve it.

SUMMARY OF CODE GRANT TYPE



1) AUTHORIZATION REQUEST

The client application sends a request to the OAuth service's `/authorization` endpoint asking for permission to access specific user data.

Auto (SQL)

```
GET /authorization?client_id=12345&redirect_uri=https://client-
app.com/callback&response_type=code&scope=openid%20profile&state=ae13d489bd00e3c24 HTTP/1.1
Host: oauth-authorization-server.com
```

2) USER LOGIN CONSENT

The authorization server redirect the client browser to OAuth provider login, this because the user must consent the data exchange between Service provider and Identity provider.

3) AUTHORIZATION CODE GRANT

If the user authorizes the data exchange, the identity provider sends to the client browser a redirect to the client back-end, passing to this redirect an authorization code.

Auto (Visual Basic .NET)

```
GET /callback?code=a1b2c3d4e5f6g7h8&state=ae13d489bd00e3c24 HTTP/1.1
Host: client-app.com
```

The client back-end stores this code, that has an exipiration time.

4) ACCESS TOKEN REQUEST

From now every communication is done back-end to back-end.

The client back-end (service provider) sends to the identity provider a request to obtain an access token, using the authorization code get before:

Auto (Bash)

```
POST /token HTTP/1.1
Host: oauth-authorization-server.com
…
client_id=12345&client_secret=SECRET&redirect_uri=https://client-
app.com/callback&grant_type=authorization_code&code=a1b2c3d4e5f6g7h8
```

5) ACCESS TOKEN GRANT

The identity provider sends to the client back-end the access token, that can be used then to obtain user info, according to the scope

Auto (Bash)

```
{
    "access_token": "z0y9x8w7v6u5",
    "token_type": "Bearer",
    "expires_in": 3600
```

```
    "expires_in": 3600,
    "scope": "openid profile",
    …
}
```

6) API CALL

Now the client back-end has the access token and can use it to perform calls to the identity provider API.

Auto (Visual Basic .NET)

```
GET /userinfo HTTP/1.1
Host: oauth-resource-server.com
Authorization: Bearer z0y9x8w7v6u5
```

7) USER DATA

The identity provider checks the validity of the token and sends to the client back-end the user information required:

Auto (Bash)

```
{
    "username":"carlos",
    "email":"carlos@carlos-montoya.net",
    …
}
```

8) USER LOGGED IN

Now the client knows that the user is legit by the fact that the identity provider authenticates it.

From now everything is in the application hands, now it can create a session for the user using session id or use the token to authenticate each period of time the user.

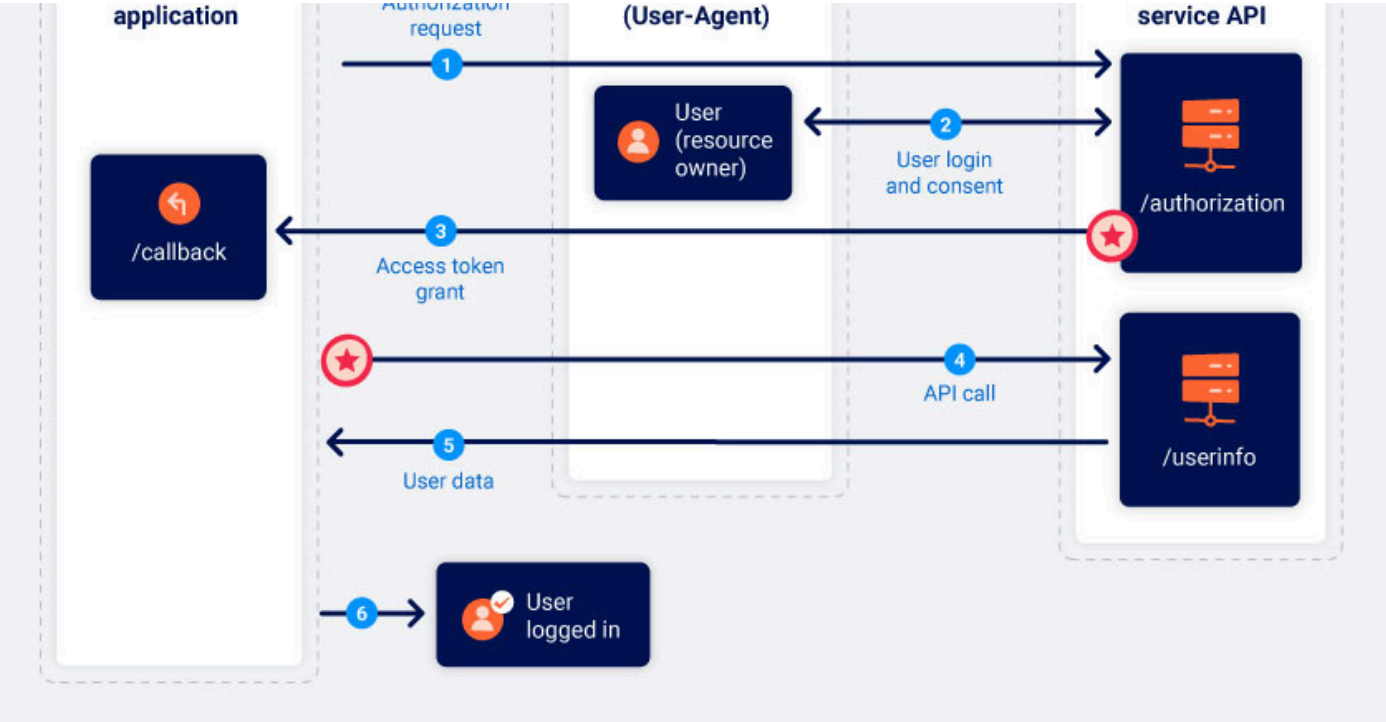
In general from this point the application can also forget the identity provider.

SUMMARY OF IMPLICIT CODE GRANT TYPE

If we don't have a back-end we can use the weaker implicit grant type.

In this case we don't have any authorization token.

By the fact we don't have a back-end we have to consider that the access token is exposed in the browser, so if there is a XSS vulnerability we are fucked.



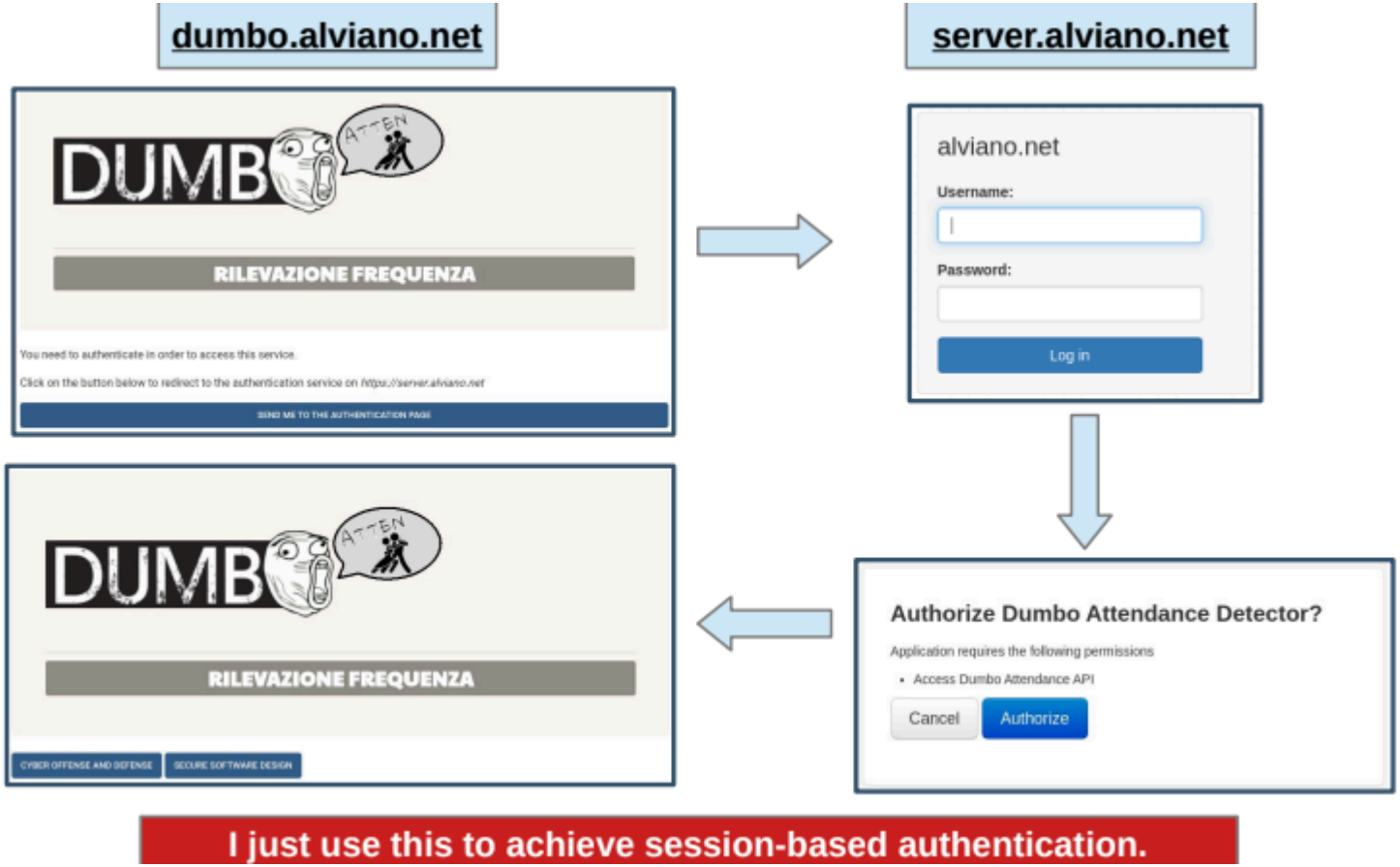
So we don't have an authorization token and every communication is done using the browser.

After the first request we get directly the access token.

It is deprecated in favor of Authorization Code Flow with Proof Key for Code Exchange (PKCE, that is pronunced pixie).

Now the idea is that if the single-page app is hosted in the same domain of the identity provider then we can use session-based authentication because it is safer.

Example



I just use this to achieve session-based authentication.

- from dumbo... we perform an authentication request to server....
- after the login on sever (that strores a session cookie) the server asks to the user to authorize the data transmission
- when the user authorizes the transmission teh server... sends the session cookie to dumbo...
- dumbo now can use this session cookie to perform requests to the server API

Identifying OAuth authentication

Regardless of which OAuth grant type is being used, the first request of the flow will always be a request to the `/authorization` endpoint containing a number of query parameters that are used specifically for OAuth.

In particular, keep an eye out for the `client_id`, `redirect_uri`, and `response_type` parameters.

For example, an authorization request will usually look something like this:

Auto (SQL)

```
GET /authorization?client_id=12345&redirect_uri=https://client-
app.com/callback&response_type=token&scope=openid%20profile&state=ae13d489bd00e3c24 HTTP/1.1
Host: oauth-authorization-server.com
```

RECON

Once you know the hostname of the authorization server, you should always try sending a `GET` request to the following standard endpoints:

Auto (Bash)

```
/.well-known/oauth-authorization-server
```

```
/.well-known/openid-configuration
```

These will often return a JSON configuration file containing key information, such as details of additional features that may be supported.

Exploiting OAuth authentication vulnerabilities

Improper implementation of the implicit grant type

In this flow, the access token is sent from the OAuth service to the client application via the user's browser as a URL fragment.

The client application then accesses the token using JavaScript.

The trouble is, if the application wants to maintain the session after the user closes the page, it needs to store the current user data somewhere.

To solve this problem, the client application will often submit this data to the server in a `POST` request and then assign the user a session cookie, effectively logging them in.

In the implicit flow, this `POST` request is exposed to attackers via their browser. As a result, this behavior can lead to a serious vulnerability if the client application doesn't properly check that the access token matches the other data in the request.

Flawed CSRF protection

The `state` parameter should ideally contain an unguessable value, such as the hash of something tied to the user's session when it first

The `state` parameter should ideally contain an unguessable value, such as the hash of something tied to the user's session when it first initiates the OAuth flow.

Therefore, if you notice that the authorization request does not send a `state` parameter, this is extremely interesting from an attacker's perspective because we can trigger here a CSRF attack at all.

Leaking authorization codes and access tokens

Depending on the grant type, either a code or token is sent via the victim's browser to the `/callback` endpoint specified in the `redirect_uri` parameter of the authorization request.

If the OAuth service fails to validate this URI properly, an attacker may be able to construct a CSRF-like attack, tricking the victim's browser into initiating an OAuth flow that will send the code or token to an attacker-controlled `redirect_uri`.

However a protection mechanism here is performed using the CLIENT_SECRET that is stored in the client back-end.

Flawed redirect_uri validation

If you can append extra values to the default `redirect_uri` parameter, you might be able to exploit discrepancies between the parsing of the URI by the different components of the OAuth service.

For example, you can try techniques such as:

Auto

```
https://default-host.com &@foo.evil-user.net#@bar.evil-user.net/
```

You may occasionally come across server-side parameter pollution vulnerabilities. Just in case, you should try submitting duplicate `redirect_uri` parameters as follows:

Auto (Bash)

```
https://oauth-authorization-server.com/?client_id=123&redirect_uri=client-app.com/callback&redirect_uri=evil-user.net
```

In some cases, any redirect URI beginning with `localhost` may be accidentally permitted in the production environment.

This could allow you to bypass the validation by registering a domain name such as:

Auto

```
localhost.evil-user.net
```

How to prevent OAuth authentication vulnerabilities

To prevent OAuth authentication vulnerabilities, it is essential for both the OAuth provider and the client application to implement robust validation of the key inputs, especially the `redirect_uri` parameter.

For OAuth service providers

For OAuth service providers

Require client applications to register a whitelist of valid `redirect_uris`. Wherever possible, use strict byte-for-byte comparison to validate the URI in any incoming requests. Only allow complete and exact matches rather than using pattern matching.

Enforce use of the `state` parameter. Its value should also be bound to the user's session by including some unguessable, session-specific data, such as a hash containing the session cookie. This helps protect users against CSRF-like attacks.

On the resource server, make sure you verify that the access token was issued to the same `client_id` that is making the request.

For OAuth client applications

Use the `state` parameter even though it is not mandatory.

Send a `redirect_uri` parameter not only to the `/authorization` endpoint, but also to the `/token` endpoint.

When developing mobile or native desktop OAuth client applications, it is often not possible to keep the `client_secret` private. In these situations, the `PKCE` (`RFC 7636`) mechanism may be used to provide additional protection against access code interception or leakage.

Be careful with authorization codes - they may be leaked via `Referer` headers when external images, scripts, or CSS content is loaded

OAUTH VULNERABILITIES

There are some vulnerabilities that are related to the OAuth process:

1. open redirect leads to token theft
    a. we can use an allow list
2. improper validation
    a. we can use an allow list
3. custom implementation (sometimes wrong)
    a. we can use well-known libraries
4. excessive trust on confidentiality of tokens stored in the browser (implicit grant type)
    a. there is no way to make implicit grant type 100% safe
    b. application storage is better then cookies
        i. at least data is not transmitted with every request
    c. session storage is better than application storage
        i. we have different tabs and so different data
    d. browser memory is better thab session storage
        i. we need to study the source code to find data
    e. however in any case we can steal information