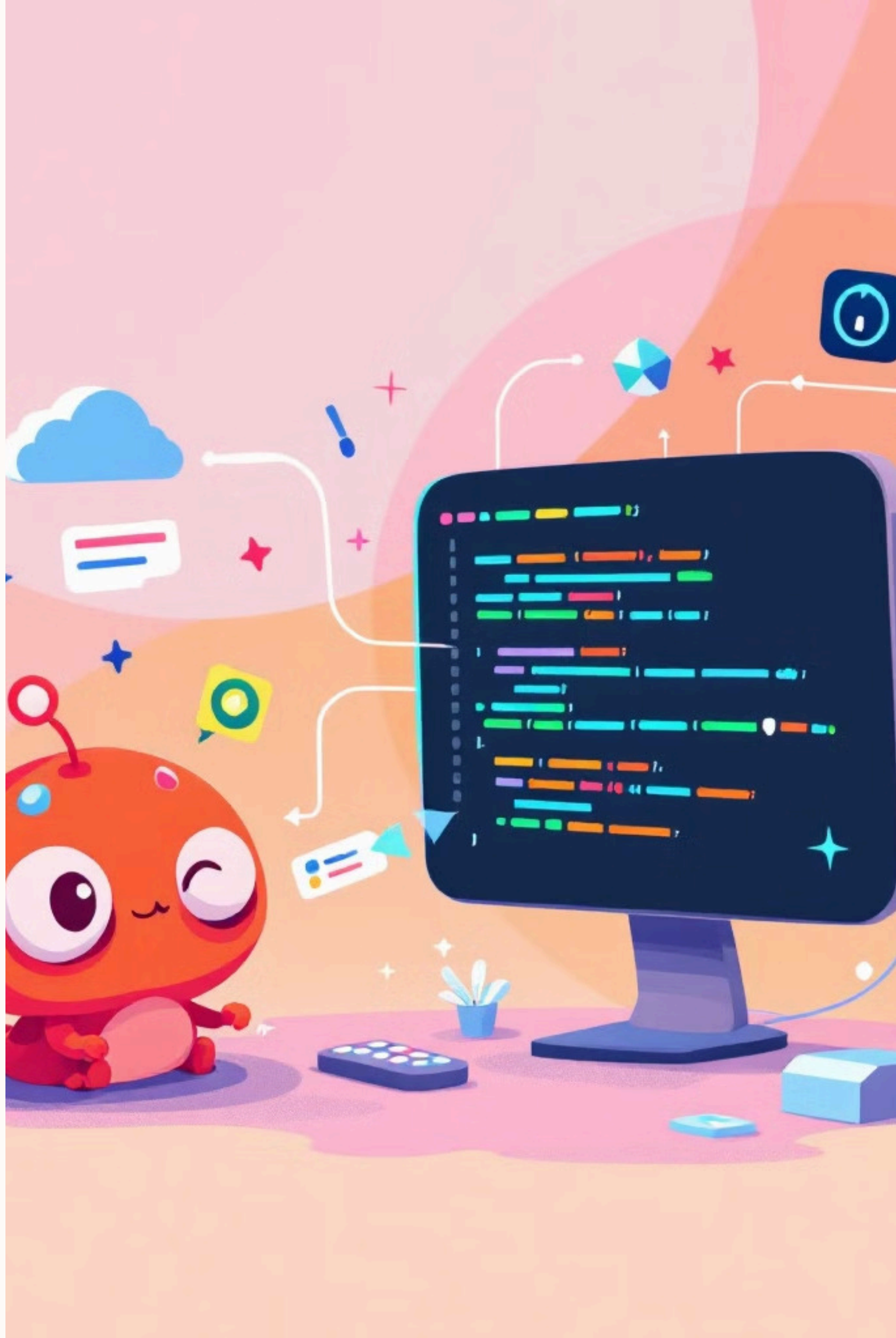
Hacking OAuth: An Attacker's View of your OAuth Flows

SK by Sheshananda Reddy Kandula



Whoami

- @Sheshananda Reddy Kandula
- 15 years in Application Security

Sheshananda Reddy Kandula
Sr Security Engineer at Adobe | AppSec | Product Securtiy | OSWE | OSCP | CISSP

- Member of ISC2 NJ Chapter

Whoami

- @Sheshananda Reddy Kandula
- 15 years in Application Security
- Member of ISC2 NJ Chapter

Agenda

- Introduction - Authentication vs Authorization
- OAuth Protocol
- How OAuth Works
- OAuth Work Flows
- OAuth Play Ground
- Common OAuth Vulnerabilities

Introduction

Authentication vs Authorization

Authentication

The process of verifying the identity of a user, process, or device, often as a prerequisite to allowing access to resources in an information system.

Authorization

Authorization is **the security process that determines a user or service's level of access**. In technology, we use authorization to give users or services permission to access some data or perform a particular action.

Authentication vs Authorization

Authentication (from Greek: αὐθεντικός *authentikos*, "real, genuine", from αὐθέντης *authentes*, "author") is the act of proving an assertion, such as the identity of a computer system user. In contrast with identification, the act of indicating a person or thing's identity, authentication is the process of verifying that identity.[1][2] It might involve validating personal identity documents, verifying the authenticity of a website with a digital certificate,[3] determining the age of an artifact by carbon dating, or ensuring that a product or document is not counterfeit.

Authorization or **authorisation** (see spelling differences) is the function of specifying rights/privileges for accessing resources, which is related to general information security and computer security, and to IAM (Identity and Access Management) in particular.[1]

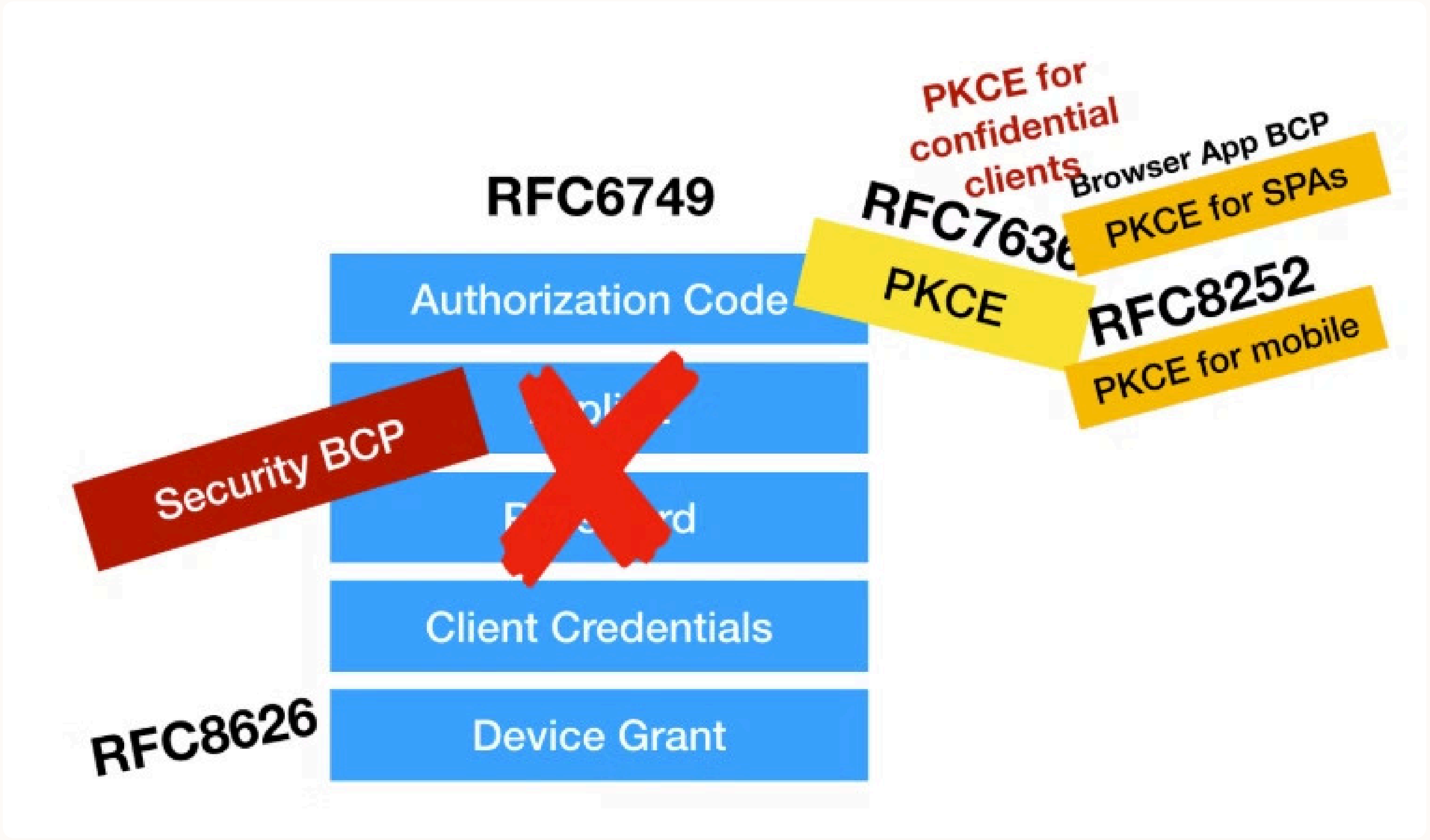
OK. What is OAuth?

OAuth is Open Authorization Protocol

OAuth is **Delegated Authorization**

OAuth (short for **open authorization**[1][2]) is an open standard for access delegation, commonly used as a way for internet users to grant websites or applications access to their information on other websites but without giving them the passwords.[3][4] This mechanism is used by companies such as Amazon,[5] Google, Meta Platforms, Microsoft, and Twitter to permit users to share information about their accounts with third-party applications or websites.

It's a Protocol —> RFCs ?



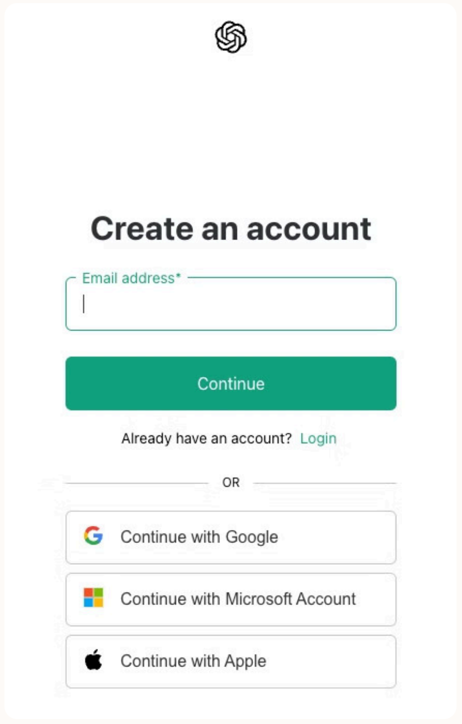
It's Time for OAuth 2.1

Why do we need to share resources securely?

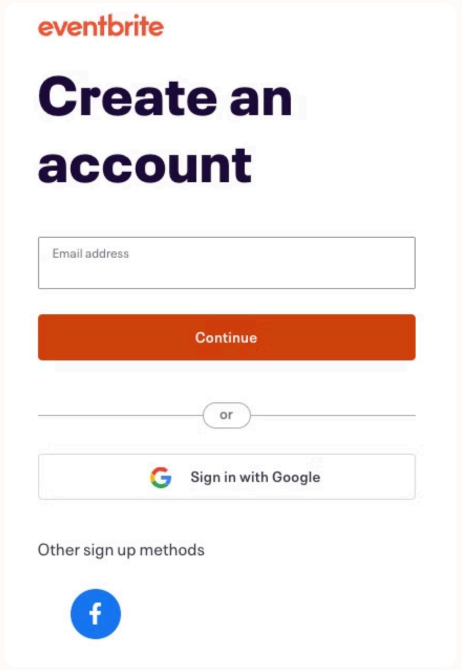
- Users would like to share their photos, videos, files, documents with other apps from One App to Another App.
- From GDrive to WhatsApp/iMessages
- But what if, user data/files are on the Server

Why do we need to share resources ~~securely~~?

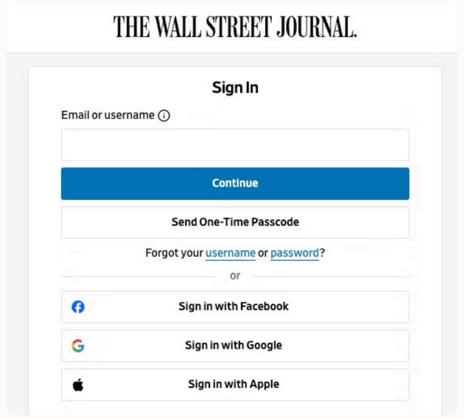
- Users don't want to Type or Enter the details details manually
- Users don't want to remember/store the passwords for every website.
- Resources like Users' First Name, Last Name, Email Address, Physical Address, Preferred Language can be shared easily without compromising security



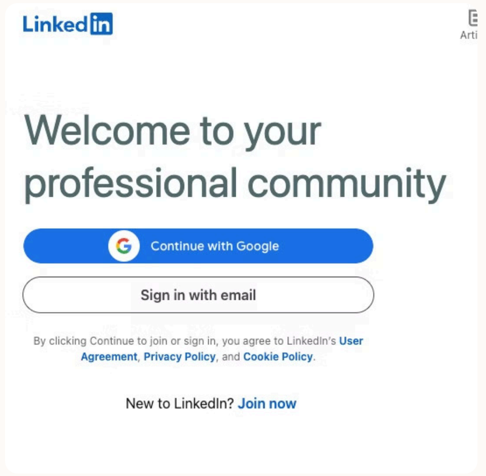
ChatGPT Sign up



Eventbrite



Wall Street Journal


Linked in
Welcome to your professional community
Continue with Google
Sign in with email
New to LinkedIn? Join now

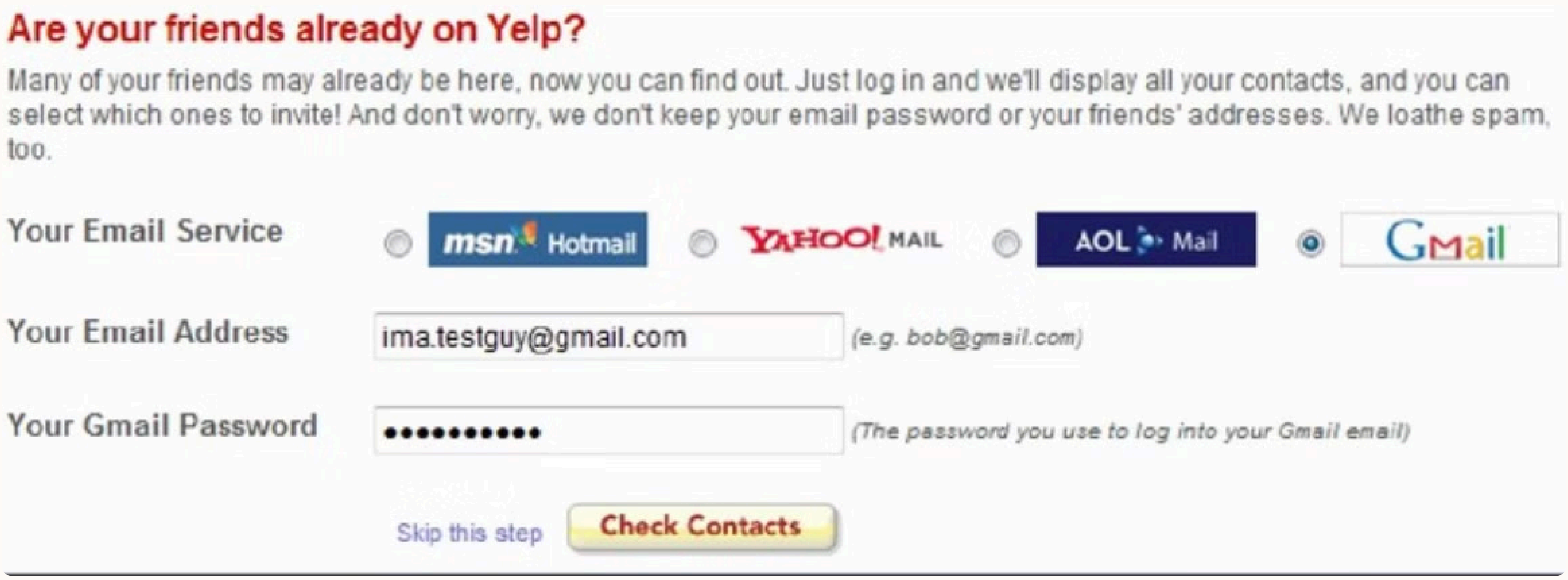
By clicking Continue to join or sign in, you agree to LinkedIn's User Agreement, Privacy Policy, and Cookie Policy.

LinkedIn

Above images only for Sign up, Mostly user share sharing name and email etc.

But OAuth can be used to share the images/files/etc as well.

OK. OAuth Solved it. What we did before OAuth?



Are your friends already on Yelp?

Many of your friends may already be here, now you can find out. Just log in and we'll display all your contacts, and you can select which ones to invite! And don't worry, we don't keep your email password or your friends' addresses. We loathe spam, too.

Your Email Service	○ msn Hotmail ○ YAHOO! MAIL ○ AOL Mail [✓ GMail]
Your Email Address	ima.testguy@gmail.com (e.g. bob@gmail.com)
Your Gmail Password	••••••••••• (The password you use to log into your Gmail email)

Ref: https://speakerdeck.com/nbarbettini/oauth-and-openid-connect-in-plain-english?slide=5

It's not a good idea to share your passwords with some random registration page

How OAuth works?

- You have an account with Google, Facebook, Apple or any company you trust

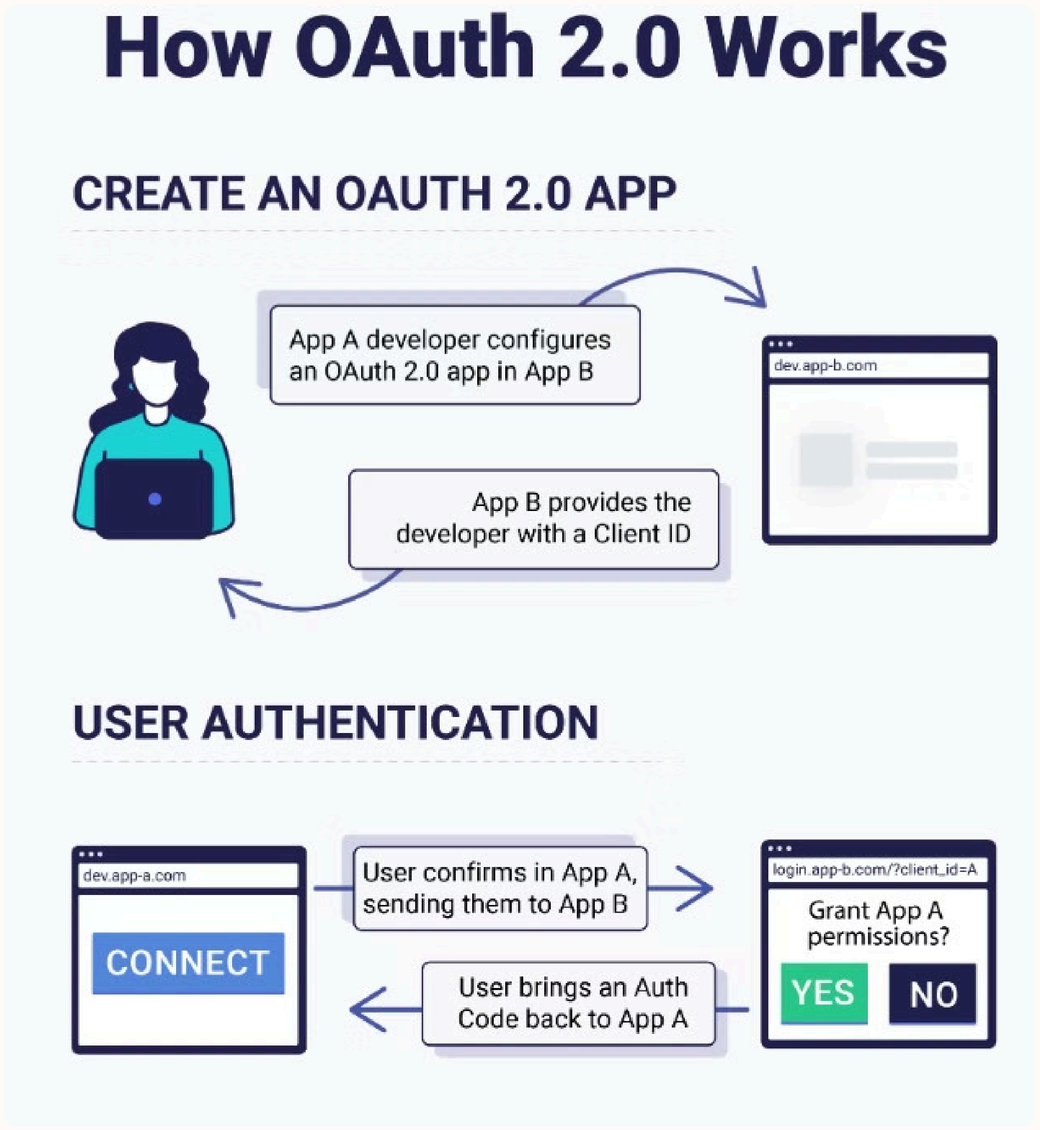
How OAuth works?

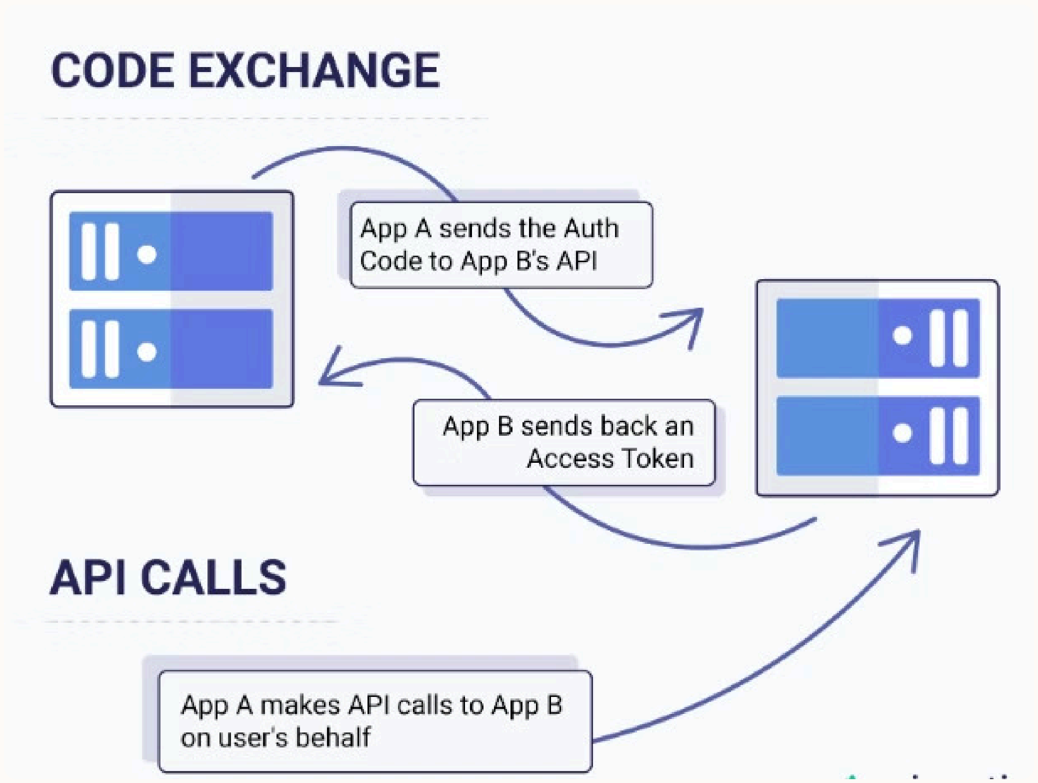
- You have already an account with Google, Facebook, Apple or any company you trust
- Some registration page ask you to Sign Up with OAuth Links as an option (as shown above)

How OAuth works?

- You have already an account with Google, Facebook, Apple or SaaS or any company you trust
- Some registration page ask you to Sign Up with OAuth Links as an option (as shown above)
- Once you logged into anyone of the Google/Facebook/Apple accounts and agree to share the data to New website, then only OAuth Servers share the data.

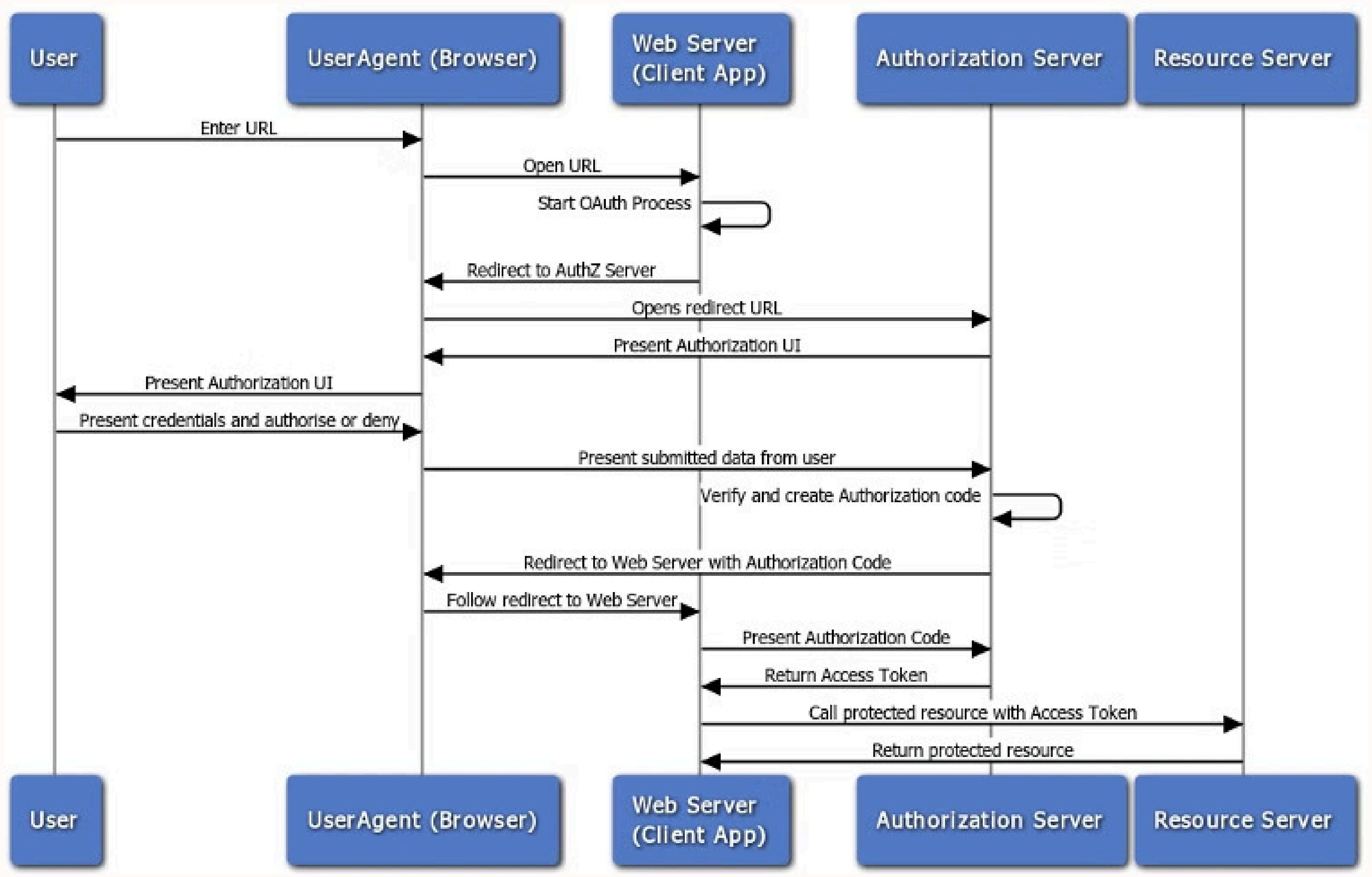
OAuth Flow at High Level


How OAuth 2.0 Works
CREATE AN OAUTH 2.0 APP
App A developer configures an OAuth 2.0 app in App B
dev.app-b.com
App B provides the developer with a Client ID
USER AUTHENTICATION
dev.app-a.com
CONNECT
User confirms in App A, sending them to App B
login.app-b.com/?client_id=A
Grant App A permissions?
YES
NO
User brings an Auth Code back to App A


CODE EXCHANGE
App A sends the Auth Code to App B's API
App B sends back an Access Token
API CALLS
App A makes API calls to App B on user's behalf

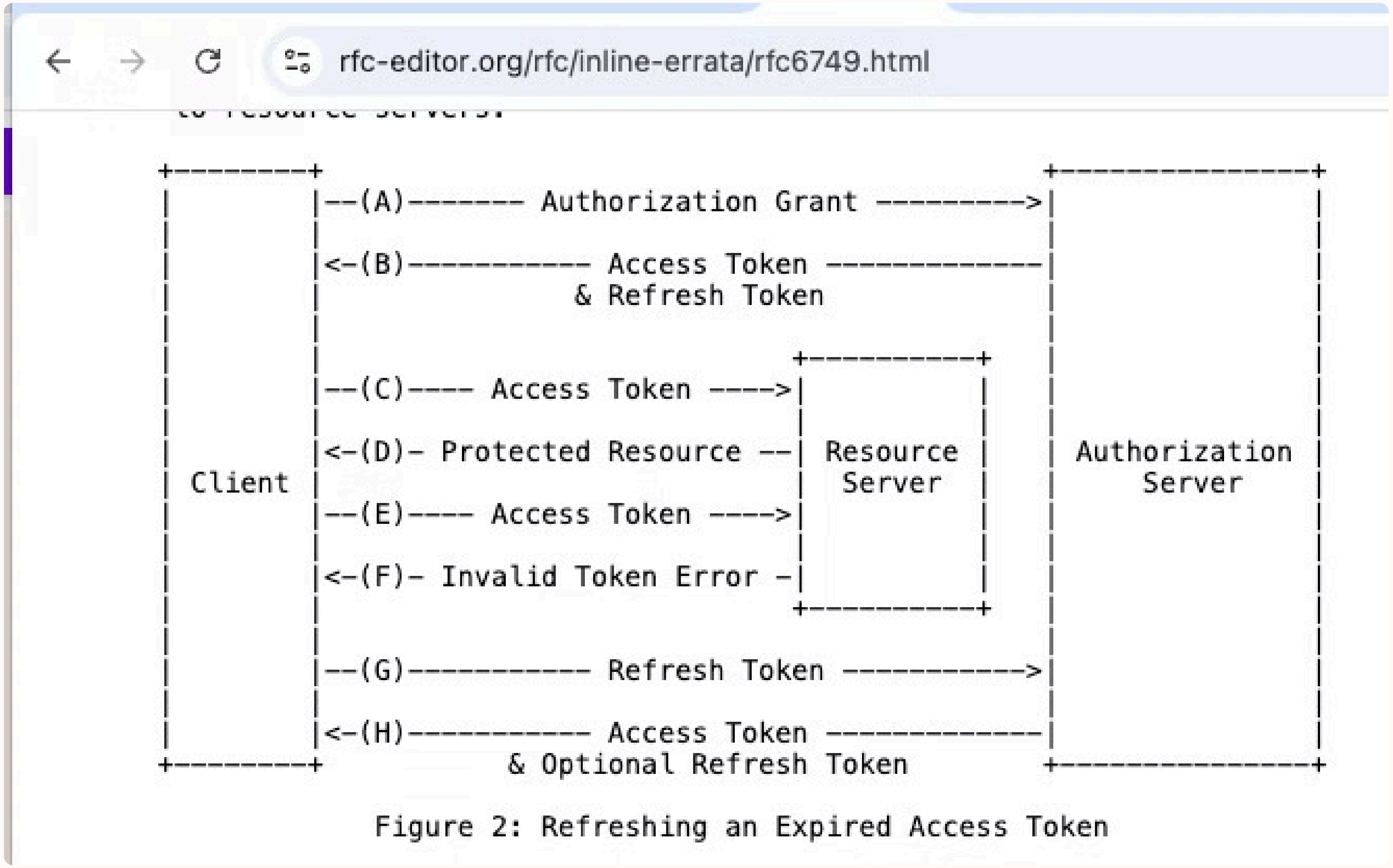
Ref: https://prismatic.io/docs/getting-started/oauth2/what-is-oauth2/

OAuth Flow



[API Gateway OAuth 2.0 Authentication Flows]()

Refreshing Access Token with Refresh Token



rfc-editor.org/rfc/inline-errata/rfc6749.html

```
  +--------+                                           +---------------+
  |        |--(A)------- Authorization Grant --------->|               |
  |        |                                           |               |
  |        |<-(B)----------- Access Token -------------|               |
  |        |               & Refresh Token             |               |
  |        |                                           |               |
  |        |                            +----------+   |               |
  |        |--(C)---- Access Token ---->|          |   |               |
  |        |                            |          |   |               |
  |        |<-(D)- Protected Resource --| Resource |   | Authorization |
  | Client |                            |  Server  |   |     Server    |
  |        |--(E)---- Access Token ---->|          |   |               |
  |        |                            |          |   |               |
  |        |<-(F)- Invalid Token Error -|          |   |               |
  |        |                            +----------+   |               |
  |        |                                           |               |
  |        |--(G)----------- Refresh Token ----------->|               |
  |        |                                           |               |
  |        |<-(H)----------- Access Token -------------|               |
  +--------+           & Optional Refresh Token        +---------------+

               Figure 2: Refreshing an Expired Access Token
```

rfc6749

OAuth Terminology

- **Resource Owner** : User
- **Resource Server** : The server that contains users information
- **Authorization Server** : Handles the user data and interactions with 3rd party app
- **Client** : 3rd Party app which is requesting access or we are registering now
    - client_id : 3rd party app
    - client_secret : it's a secret
- **Access Token and Refresh Token**
- **Authorization Code :** Intermediate to get the tokens

Sample OAuth Request

```
https://authorization-server.com/authorize?
  response_type=code
  &client_id=4SDzdxh-nbx0eq7fBK0jh8-w
  &redirect_uri=https://www.oauth.com/playground/authorization-code.html
  &scope=photo+offline_access
  &state=PVPrz3naUgOsmFxz
```

OAuth Code Flows

Authorization Code Flow

The Authorization Code flow is suitable for server-side applications, utilizing an intermediary authorization server to exchange a code for an access token.

Client Credentials Flow

The Client Credentials flow is used for machine-to-machine authentication, without user involvement.

Authorization Code Flow with PKCE (Proof Key for Code Exchange)

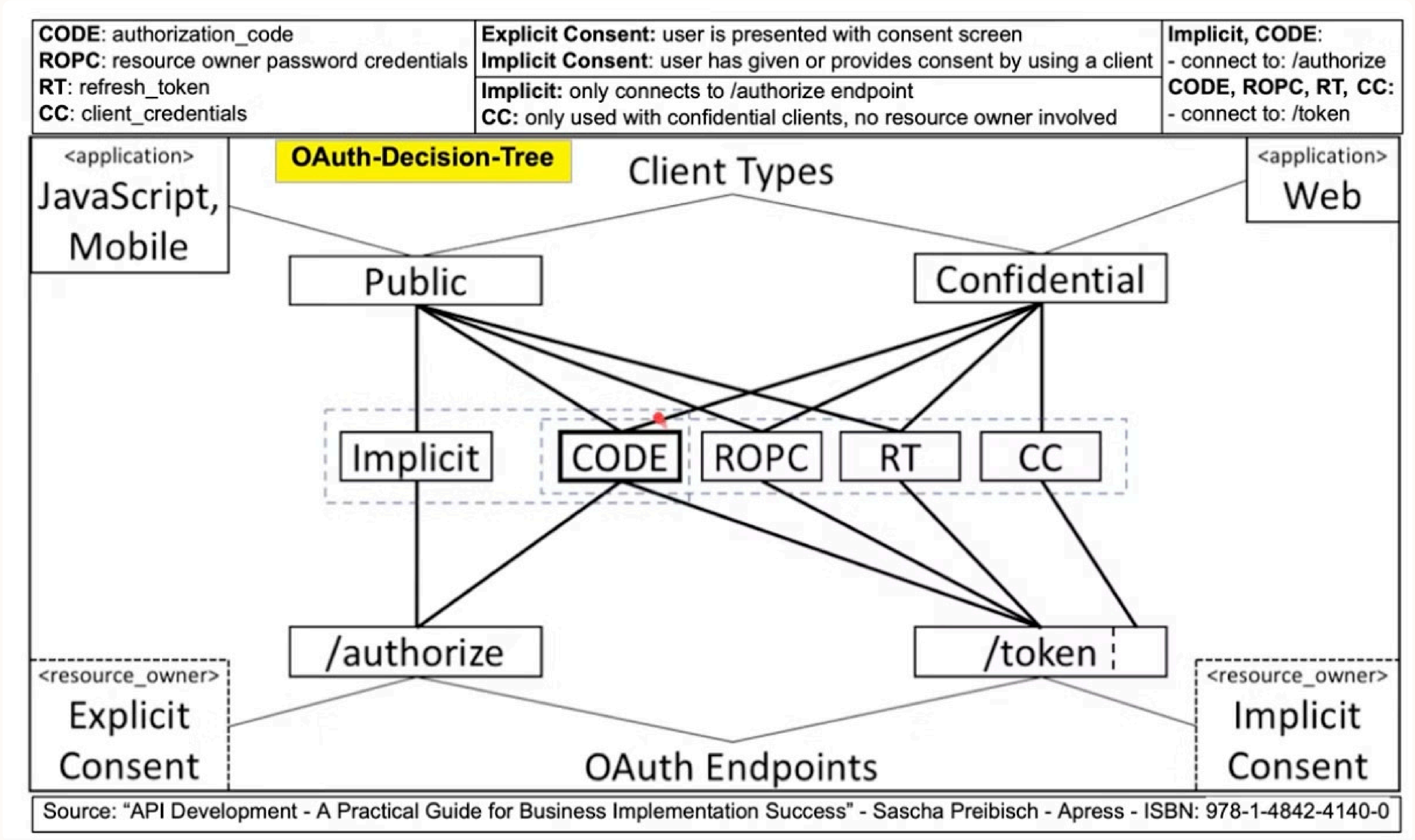
Native mobile apps, single-page applications (SPAs), or other clients that can’t keep secrets.

Protect against Authorization Code Injection attacks.

Implicit Flow (Insecure/deprecated)

The Implicit flow, designed for client-side applications, **directly returns an access token,** introducing unique risks.

OAuth Flows

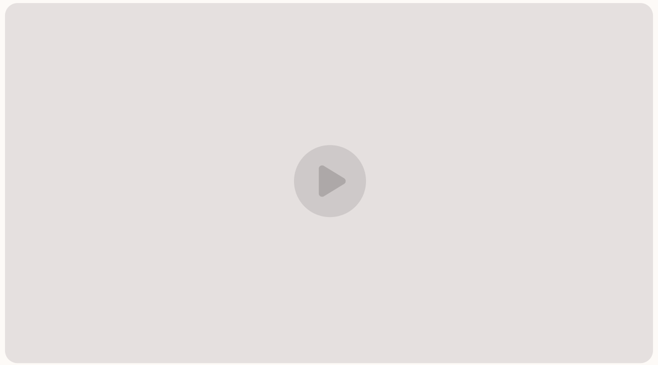


CODE: authorization_code **ROPC**: resource owner password credentials **RT**: refresh_token **CC**: client_credentials	**Explicit Consent:** user is presented with consent screen **Implicit Consent**: user has given or provides consent by using a client **Implicit:** only connects to /authorize endpoint **CC:** only used with confidential clients, no resource owner involved	**Implicit, CODE**: - connect to: /authorize **CODE, ROPC, RT, CC:** - connect to: /token

Source: "API Development - A Practical Guide for Business Implementation Success" - Sascha Preibisch - Apress - ISBN: 978-1-4842-4140-0

OAuth Playground

Demo Website: https://www.oauth.com/playground/index.html



[OAuth 2.0 Security Best Current Practice]()

OAuth Reqeusts

```
https://authorization-server.com/authorize?
  response_type=code
  &client_id=4SDzdxh-nbx0eq7fBK0jh8-w
  &redirect_uri=https://www.oauth.com/playground/authorization-code.html
  &scope=photo+offline_access
  &state=PVPrz3naUgOsmFxz
```

Take a close look at the request, You can guess few possible attacks

Common OAuth Vulnerabilities

Misconfigurations in Redirect URIs

Attackers can exploit flaws in redirect URI configurations to gain unauthorized access to sensitive user data, impersonate legitimate users, or even escalate privileges within the system.

Broken Authorization Grants

Vulnerabilities in the way authorization grants are handled can lead to account takeovers and data breaches.

Weak Client Authentication

Inadequate client authentication can allow attackers to masquerade as legitimate applications and gain unauthorized access to protected resources.

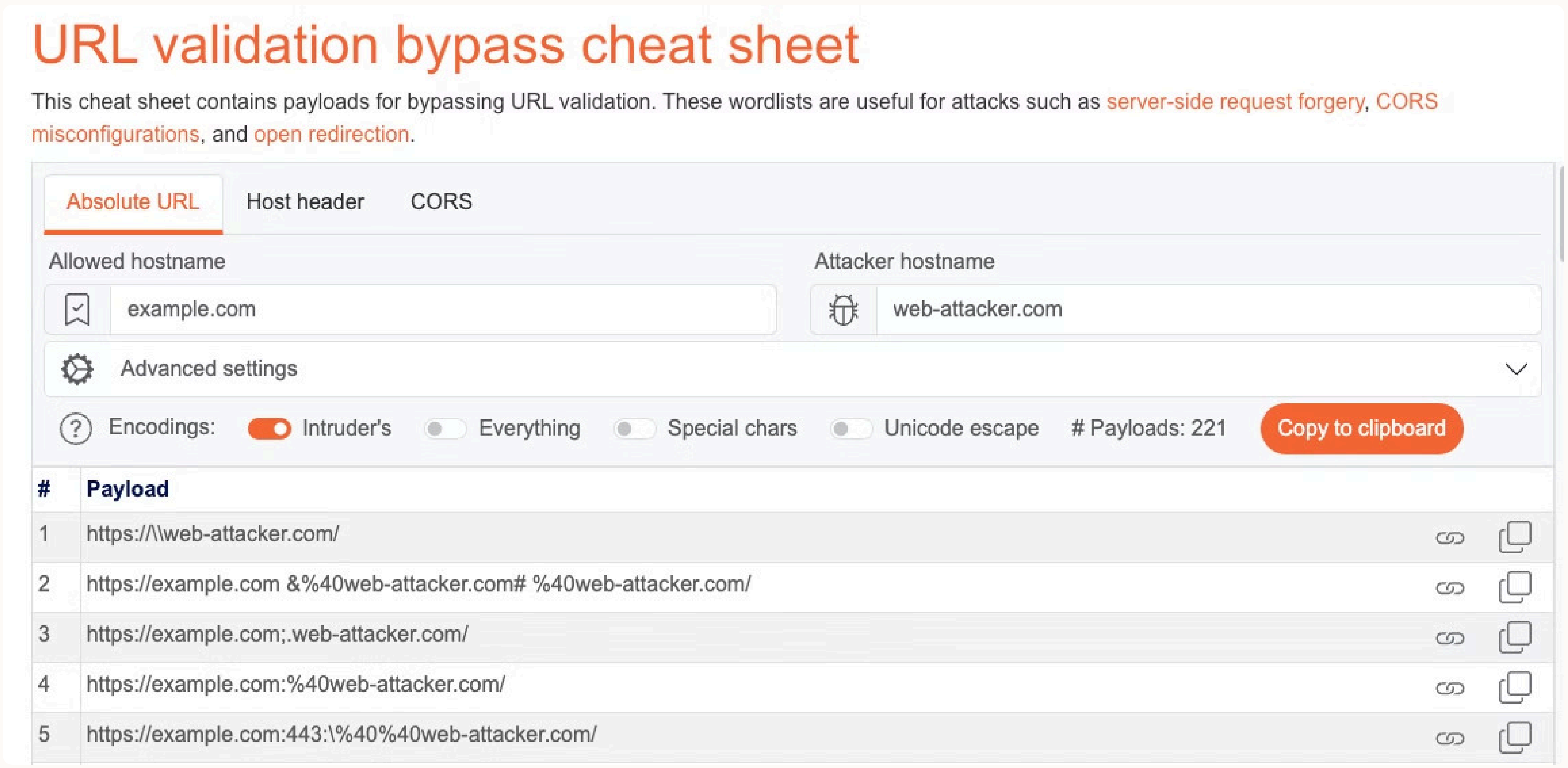
Redirect URI Misconfigurations

- No Open Redirects
- Careful validation and allowlisting of redirect URIs is critical to prevent such attacks.
- Compare client redirect URIs against the pre-registered URIs with **Exact String Match**

Misconfigurations:

- https://*.somesite.example/* —> https://attacker.example/.somesite.example
- target.com.attacker.com
- Double submit redirect_uri.
- https://default-host.com &@foo.evil-user.net#@bar.evil-user.net/

Portswigger has URL Validation bypass cheat sheet:



URL validation bypass cheat sheet

This cheat sheet contains payloads for bypassing URL validation. These wordlists are useful for attacks such as server-side request forgery, CORS misconfigurations, and open redirection.

Allowed hostname: example.com

Attacker hostname: web-attacker.com

Encodings: Intruder's (on) Everything (off) Special chars (off) Unicode escape (off) # Payloads: 221

#	Payload
1	https://\\web-attacker.com/
2	https://example.com &%40web-attacker.com# %40web-attacker.com/
3	https://example.com;.web-attacker.com/
4	https://example.com:%40web-attacker.com/
5	https://example.com:443:\%40%40web-attacker.com/

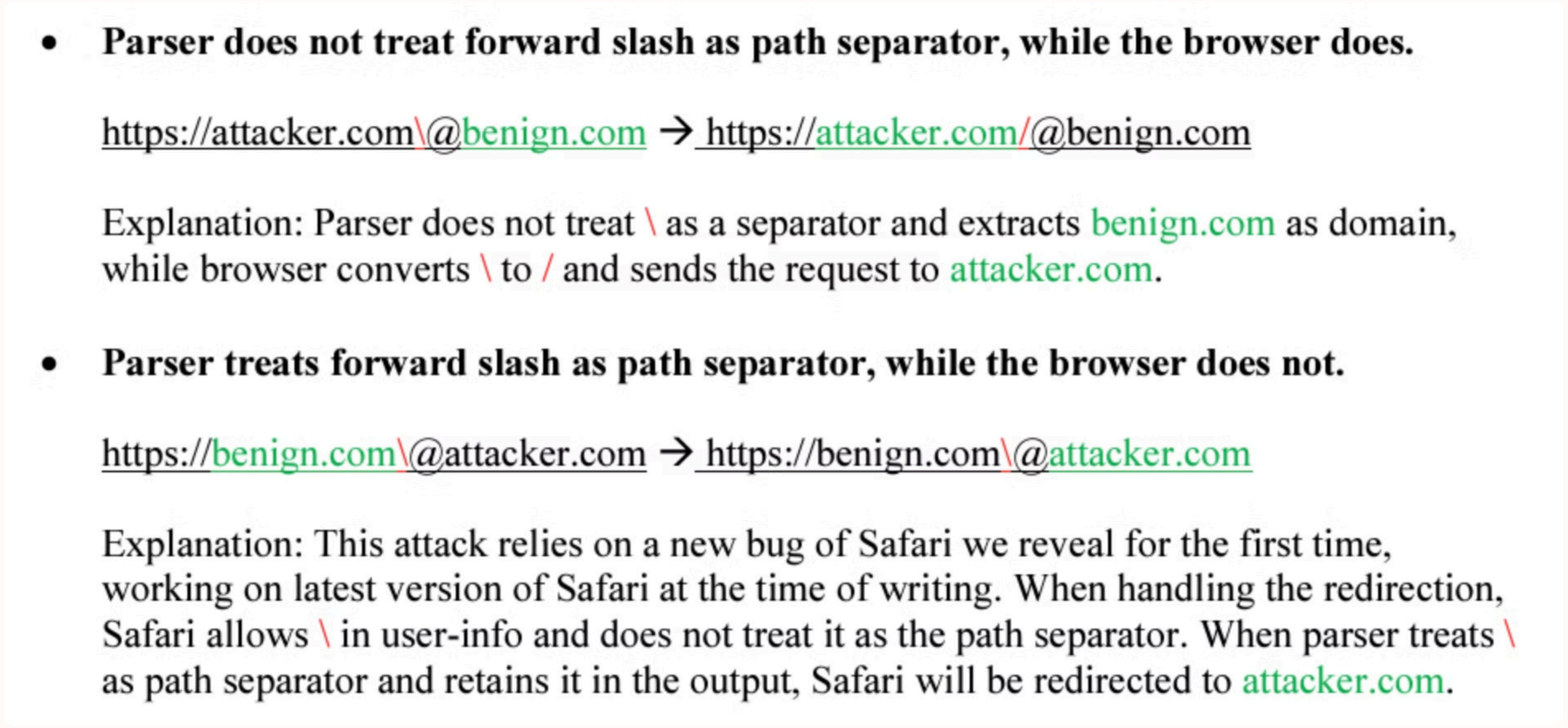
URL validation bypass cheat sheet for SSRF/CORS/Redirect - 2024 Edition | Web Security Academy

Lab: OAuth account hijacking via redirect_uri | Web Security Academy

https://www.acsac.org/2023/files/web/slides/innocenti-97-oauthvalidation.pdf

Redirect URI Misconfigurations

Due to Parser validations



- **Parser does not treat forward slash as path separator, while the browser does.**

  https://attacker.com\@benign.com → https://attacker.com/@benign.com

  Explanation: Parser does not treat \ as a separator and extracts benign.com as domain, while browser converts \ to / and sends the request to attacker.com.

- **Parser treats forward slash as path separator, while the browser does not.**

  https://benign.com\@attacker.com → https://benign.com\@attacker.com

  Explanation: This attack relies on a new bug of Safari we reveal for the first time, working on latest version of Safari at the time of writing. When handling the redirection, Safari allows \ in user-info and does not treat it as the path separator. When parser treats \ as path separator and retains it in the output, Safari will be redirected to attacker.com.

[Lab: OAuth account hijacking via redirect_uri | Web Security Academy]()

https://www.acsac.org/2023/files/web/slides/innocenti-97-oauthvalidation.pdf

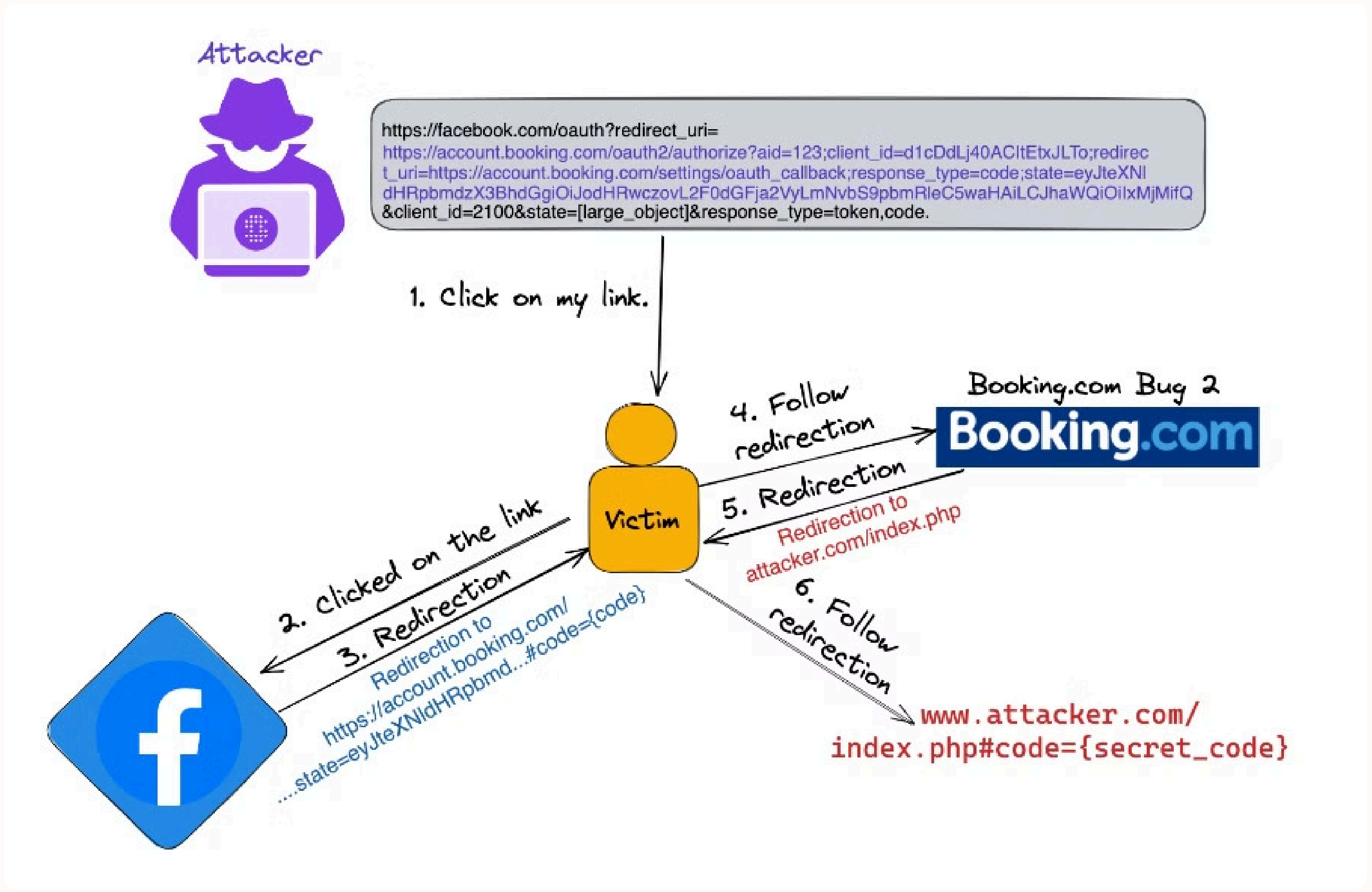
https://i.blackhat.com/asia-19/Fri-March-29/bh-asia-Wang-Make-Redirection-Evil-Again-wp.pdf

Redirect URI Misconfigurations examples

Redirect URI + XSS/Open Redirection/HTML Injection —> Account takeover

Multiple bugs chained to takeover Facebook Accounts which uses Gmail.

MAY 14, 2022


Attacker
https://facebook.com/oauth?redirect_uri=
https://account.booking.com/oauth2/authorize?aid=123;client_id=d1cDdLj40ACltEtxJLTo;redirec
t_uri=https://account.booking.com/settings/oauth_callback;response_type=code;state=eyJteXNl
dHRpbmdzX3BhdGgiOiJodHRwczovL2F0dGFja2VyLmNvbS9pbmRleC5waHAiLCJhaWQiOiIxMjMifQ
&client_id=2100&state=[large_object]&response_type=token,code.
1. Click on my link.
Victim
2. Clicked on the link
3. Redirection
Redirection to
https://account.booking.com/
....state=eyJteXNldHRpbmd...#code={code}
4. Follow redirection
Booking.com Bug 2
Booking.com
5. Redirection
Redirection to
attacker.com/index.php
6. Follow redirection
www.attacker.com/
index.php#code={secret_code}

[OAuth Account Takeover - Account Takeover on Booking.com](#)

[Multiple bugs chained to takeover Facebook Accounts which uses Gmail.](#)

OAuth Authorization Grant Vulnerabilities

- **OAuth Authorization Grants** (e.g., Authorization Code, Implicit Flow) are vulnerable to exploitation.
- Improper implementation can allow attackers to bypass the authorization process.

Weakness in Grant Validation Logic:

- Failure to validate key parameters like **state** can lead to unauthorized access.
- Flaws in token exchange mechanisms can allow attackers to **obtain valid tokens** without user consent.

Attacking Weak Client Authentication

Exploiting Hardcoded Secrets

- **Weak Client Authentication**: Many OAuth implementations use inadequate client authentication methods, allowing attackers to impersonate legitimate clients and gain unauthorized access to sensitive data.
- **Insecure Secret Storage**: Hardcoding client secrets in application code or storing them insecurely makes it easy for attackers to retrieve and misuse these secrets to authenticate as valid clients.

Mitigating Weak Authentication

- **Enforce Strong Client Authentication**: Use cryptographic keys instead of client secrets, rotate secrets regularly, and ensure credentials are securely stored, such as in a secure vault.
- **Access Control and Monitoring**: Implement strict access control measures and actively monitor access logs to detect and respond to unauthorized client access attempts.
- **Use OAuth Best Practices**: Follow OAuth security guidelines, including using PKCE (Proof Key for Code Exchange) for public clients and restricting access scopes to minimize exposure.

Discovering and Leveraging Insecure Token Storage

- **Improper Token Storage**: Storing access or refresh tokens in plain text or without encryption can expose them to attackers, leading to unauthorized access.
- **Token Rotation & Revocation Risks**: Failing to rotate or revoke refresh tokens could allow attackers to maintain persistent access even if a token is compromised.
- **Security Best Practices**: Tokens must be encrypted at rest, rotated regularly, and revoked promptly when sessions end or accounts are deactivated.

Mitigating OAuth Risks: Best Practices

- **Enforce Strict Redirect URI Validation**: Ensure all redirect URIs are **allowlisted** and properly **sanitized** to prevent hijacking. This minimizes the risk of redirect-based attacks.
- **Strengthen Client Authentication**: Use **cryptographic keys** instead of client secrets when possible, rotate credentials regularly, and store them securely (e.g., using a secure vault).
- **Robust Authorization Grant Validation**: Verify the integrity of the **state parameters** to prevent CSRF attacks and validate token exchanges to ensure requests come from legitimate sources.
- **Secure Token Storage and Handling**: Use **encrypted storage** for tokens, enforce **regular rotation and prompt revocation** when tokens are no longer needed, and avoid storing tokens in accessible client storage (e.g., in localStorage for JavaScript).

IETF Datatracker

OAuth 2.0 Security Best Current Practice

This document describes best current security practice for OAuth 2.0. It updates and extends the threat model and security advice given in RFC 6749, RFC 6750, and RFC...

RFC Security Best Practices

References

RFC 6749: The OAuth 2.0 Authorization Framework

OAuth 2.0 — OAuth

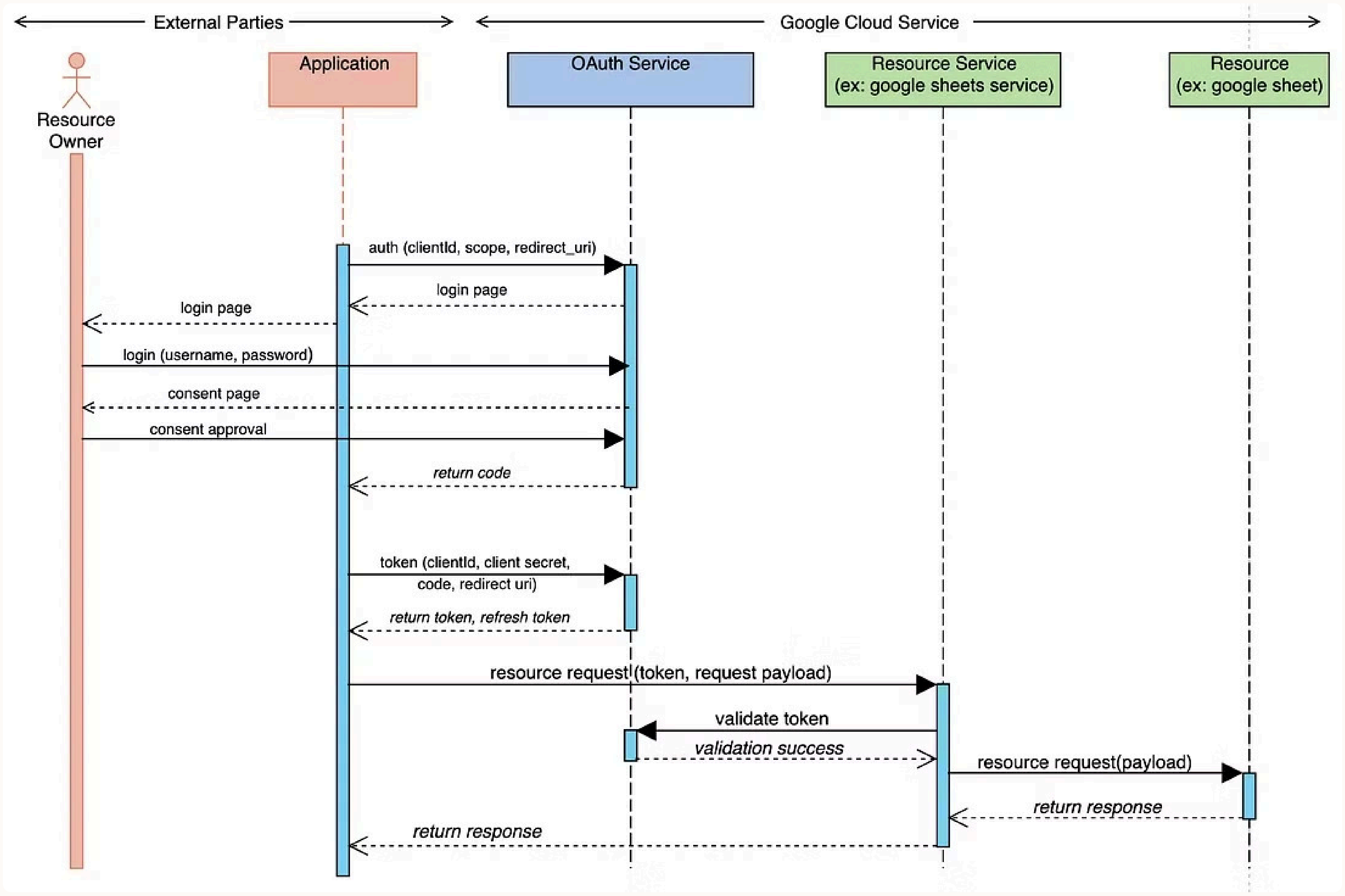
OAuth 2.0 Playground

OAuth 2.0 authentication vulnerabilities | Web Security Academy

OAuth2 - OWASP Cheat Sheet Series

OAuth 2.0 and OpenID Connect (in plain English)

Google OAuth Flow



External Parties
Google Cloud Service
Resource Owner
Application
OAuth Service
Resource Service (ex: google sheets service)
Resource (ex: google sheet)
auth (clientId, scope, redirect_uri)
login page
login page
login (username, password)
consent page
consent approval
return code
token (clientId, client secret, code, redirect uri)
return token, refresh token
resource request (token, request payload)
validate token
validation success
resource request(payload)
return response
return response

[Google Oauth2 API Explained]

Thank You

Sheshananda Reddy Kandula
Sr Security Engineer at Adobe | AppSec | Product Securtiy | OSWE | OSCP | CISSP