Presentation of Authentication Concepts of *Open Authorization 2*

Flows in OAuth 2

Timo Pagel
About Me

- DevSecOps Consultant
- Lecturer for *Security in Web Applications* at University of Applied Sciences Kiel/Wedel
- Open Source / Open Knowledge Enthusiast
About Me

- DevSecOps Consultant
- Lecturer for *Security in Web Applications* at *University of Applied Sciences Kiel/Wedel*
- Open Source / Open Knowledge Enthusiast
  - OWASP Juice Shop
  - DevSecOps Maturity Model
  - OWASP Security Pins Project
  - Full University Module Security in Web App.
  - OWASP Software Assurance Maturity Model
Agenda

- Introduction
- Flows
- Conclusion
Agenda

- Introduction
- Flows
- Conclusion
Classic Username/Password

Username / Password

Login Successful / Unsuccessful

Blog A
Classic Username/Password

Username / Password
Login Successful / Unsuccessful

Blog A

Blog B

OWASP
Classic Username/Password

Username / Password
Login Successful / Unsuccessful

Blog A

Blog B

Username / Password
Login Successful / Unsuccessful
Classic Username/Password

Username / Password
Login Successful / Unsuccessful

Blog A

Blog B

Role_Permission

<table>
<thead>
<tr>
<th>Role</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publisher</td>
<td>Write</td>
</tr>
<tr>
<td>Publisher</td>
<td>Read</td>
</tr>
<tr>
<td>Publisher</td>
<td>Publish</td>
</tr>
<tr>
<td>Writer</td>
<td>Read</td>
</tr>
<tr>
<td>Writer</td>
<td>Write</td>
</tr>
</tbody>
</table>

Username_Role

<table>
<thead>
<tr>
<th>Username</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tux</td>
<td>Publisher</td>
</tr>
<tr>
<td>Tuxine</td>
<td>Writer</td>
</tr>
</tbody>
</table>
# Classic Username/Password

## Username / Password

- Login Successful / Unsuccessful

### Blog A

### Blog B

## Role_Permission

<table>
<thead>
<tr>
<th>Role</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publisher</td>
<td>Write</td>
</tr>
<tr>
<td>Publisher</td>
<td>Read</td>
</tr>
<tr>
<td>Publisher</td>
<td>Publish</td>
</tr>
<tr>
<td>Writer</td>
<td>Read</td>
</tr>
<tr>
<td>Writer</td>
<td>Write</td>
</tr>
</tbody>
</table>

## Username_Role

<table>
<thead>
<tr>
<th>Username</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tux</td>
<td>Publisher</td>
</tr>
<tr>
<td>Tuxine</td>
<td>Writer</td>
</tr>
</tbody>
</table>
## Classic Username/Password

![Classic Username/Password Diagram](image)

### Username_Role

<table>
<thead>
<tr>
<th>Username</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tux</td>
<td>Publisher</td>
</tr>
<tr>
<td>Tuxine</td>
<td>Writer</td>
</tr>
</tbody>
</table>

### Role_Permission

<table>
<thead>
<tr>
<th>Role</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publisher</td>
<td>Write</td>
</tr>
<tr>
<td>Publisher</td>
<td>Read</td>
</tr>
<tr>
<td>Publisher</td>
<td>Publish</td>
</tr>
<tr>
<td>Writer</td>
<td>Read</td>
</tr>
<tr>
<td>Writer</td>
<td>Write</td>
</tr>
</tbody>
</table>
Classic Username/Password

Username / Password
Login Successful / Unsuccessful

Blog A

Usage of the UID/Password-Anti Pattern

<table>
<thead>
<tr>
<th>Role</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publisher</td>
<td>Write</td>
</tr>
</tbody>
</table>

Tuxine  Writer
Writer  Write
How do we solve the UID-Password-Anti-Pattern?

-> Tokens
OAuth Idea

Username / Password

Token

Client_Permission

<table>
<thead>
<tr>
<th>Client</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>X…</td>
<td>Read</td>
</tr>
<tr>
<td>Y…</td>
<td>Write</td>
</tr>
</tbody>
</table>

Blog A (Client)

Blog B (Provider)
OAuth Idea

<table>
<thead>
<tr>
<th>Client</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>X…</td>
<td>Read</td>
</tr>
<tr>
<td>Y…</td>
<td>Write</td>
</tr>
</tbody>
</table>

Blog A (Client)

Blog B (Provider)

Token

Username / Password

Client_Permission

Token

Token Accepted
Agenda

● Introduction
● Flows
● Conclusion
Client Credentials Flow

Client App (Server) 

Authorization Server

Resource Server
Client Credentials Flow

Client App (Server) → Authorization Server

Client Credentials (client_id/client_secret) → Authenticate Client
Access token

Access protected resource with access token → Protected resource response

Resource Server
Overview Client Credentials Flow

- No user-based Authentication
  Scope/Permissions: Bound to clients
- Usage: Intranet
Resource Owner Password Credentials Flow

Resource Owner

Client App

Authorization Server

Resource Server
Resource Owner Password Credentials Flow

1. Resource owner credentials
2. Resource owner credentials
3. Authenticate resource owner
4. Authenticate client
5. Access token
Please enter your Google Username and Google Password
Resource Owner Password Credentials Flow

1. Resource owner credentials
   - Authenticate resource owner
   - Authenticate client
   - Access token

2. Access protected resource with access token
   - Protected resource response

Usage of the UID/Password-Anti Pattern
Resource Owner Password Credentials Flow

Resource owner credentials

Authenticate resource owner

Authenticate client

Access token

Access protected resource with access token

What happens after the access token has expired?
Resource Owner Password Credentials Flow

1. Resource owner credentials
   - Authenticate resource owner
   - Authenticate client
   - Access token (optional refresh token)

2. Access protected resource with access token

3. Refresh token
   - Access token (refresh token)

4. Resource owner credentials

Client App

Authorization Server

Resource Server
[...] The **resource owner password credentials** grant type is suitable in cases where the

*resource owner has a trust relationship with the client*, such as the device operating system [...] 

Source: RFC 6749 The OAuth 2.0 Authorization Framework - Section 4.3
Interpretation of OAuth ROPC-Specification

- The client and the device are completely under your control
- All other flows are not supported by the client
Interpretation of OAuth ROPC-Specification

- Use Case: To move legacy application into the OAuth2-Universe
  - Scope
  - Expiration of tokens
  - ...

ROPC Main Risks Overview

- UID/password anti-pattern
  -> client, eavesdroppers, or endpoints could eavesdrop the user id and password
- Validation of the client's identity not possible
- Client app might issue a not needed scope

- Token revocation nearly useless
## Scopes

<table>
<thead>
<tr>
<th>Action</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>View own email</td>
<td>profile.email:view</td>
</tr>
<tr>
<td>Modify own email</td>
<td>profile.email:update</td>
</tr>
<tr>
<td>Delete own email</td>
<td>profile.email:delete</td>
</tr>
</tbody>
</table>
Request profile.email:view

Single factor required

Credentials

Token (scope: profile.email:view)

View profile with Token (profile.email:view)

E-Mail
Request profile.email:view

Single factor required

Credentials

Token (scope: profile.email:view)

View profile with Token (profile.email:view)

E-Mail

Update profile with Token (profile.email:view)

Error: Insufficient scope

Request profile.email:update

Multiple factors required
Implicit Flow

- Use Case: Browser
- Client Secret: Confidentiality can not be guaranteed
Implicit Flow

Client/Browser

Resource Owner

JavaScript

Authorization Server

Frontend Server

Resource Server
Implicit Flow

Client/Browser → Page with JS → Authorization Server → Frontend Server → Resource Server

- Enter URL
- Present UI
- Present Credentials
- (Performs Redirect)
- Present Authorization UI
- Present credentials
- Open redirect URL
Implicit Flow

Enter URL
- Page with JS
- (Performs Redirect)
- Open redirect URL
- Present Authorization UI
- Present credentials
- Verify credentials and create access token
- Redirect to frontend server (access token in # fragment)

Client/Browser

JavaScript

Authorization Server

Frontend Server

Resource Server

Resource Owner

[Diagram showing the flow of implicit authentication with JavaScript performing redirects and presenting credentials]
**Implicit Flow**

- **Client/Browser**
  - Enter URL
  - Present UI
  - Present Credentials

- **JavaScript**
  - Page with JS
    - Open redirect URL
  - Present Authorization UI
  - Present credentials
  - Verify credentials and create access token
  - Redirect to frontend server (access token in # fragment)

**Resources**

- **Resource Owner**
- **Authorization Server**
- **Frontend Server**
- **Resource Server**
Implicit Flow

Client/Browser

Enter URL

(Performs Redirect)

Present Authorization UI

Present credentials

Verify credentials and create access token

Redirect to frontend server (access token in # fragment)

Follow redirect URL (without access token) and get page with JS

Extract and temp. store access token

Call protected resource with access token

Return protected resource

JavaScript Server

Authorization Server

Frontend Server

Resource Server

Page with JS

Enter URL
Threats Implicit Flow

- Resource owners might issue a token to a malicious client (e.g. via phishing)
- Attackers might steal token via other mechanisms

Source: RFC 6749 The OAuth 2.0 Authorization Framework - Section 10.16

- Main Risk: Whom is a token issued to?
Further Risks/Info

- Use Case: Browser-Applications
- Silent Refresh
- Disadvantages: Man-in-the-Middle can fetch tokens

-> No refresh tokens
Authorization Code Grant

[...] the Authorization Code flow should only be used [...] where the Client Secret can be safely stored. [...]
Authorization Code Grant Flow

1. Resource Owner
2. Open Client (App)
3. User-Agent
4. Authorization Server
5. Resource Server

- Resource Owner
- Open Client (App)
- Present authorization UI
- Present credentials
- Open redirect URL with client identifier
- Validate request
- Present authorization UI
- Present submitted credentials
- Open redirect URL with client identifier
- Present authorization UI
- Validate request
- Present submitted credentials
**Authorization Code Grant Flow**

1. **Open Client (App)**
   - Resource Owner
   - User-Agent

2. **Open redirect URL with client identifier**
   - Client
   - Resource Server

3. **Present authorization UI**
   - Present credentials
   - Authorization code
   - Redirection URI and Authorization code

4. **Present authorization UI**
   - Present submitted credentials
   - Authorization code

5. **Validate request**
   - Open redirect URL with client identifier

6. **Present credentials**
   - Present authorization UI

**OWASP**
Authorization Code Grant Flow

Why isn’t the access token directly issued?
Authorization Code Grant Flow

- Resource Owner
- Open Client (App)
- Present authorization UI
- Present credentials
- Authorization code
- Redirection URI and Authorization code
- Client
- Open redirect URL with client identifier
- Validate request
- Present authorization UI
- Present submitted credentials
- Authorization code
- Authorization Server
- Open redirect URL with client identifier
- Present authorization UI
- Client
- Open redirect URL with client identifier
- Present authorization UI
- Present credentials
- Authorization code
- Authorization Server
- Present submitted credentials
- Authorization code
- Redirection URI and Authorization code

Threats to URIs:
- Referrer headers
- Request logs
- Browser history

Why isn’t the access token directly issued?
Authorization code has a very short lifetime (seconds) to make replay attacks hard

Why isn’t the access token directly issued?
Authorization Code Grant Flow

1. **Open Client (App)**
   - Present authorization UI

2. **Present credentials**
   - Authorization code
   - Redirection URI and Authorization code

3. **Authorization code**
   - Access token (optional refresh token)

4. **Access token (optional refresh token)**
   - Call protected resource with access token
   - Return protected resource

5. **Refresh token**

6. **Open redirect URL with client identifier**
   - Validate request
   - Present authorization UI

7. **Present submitted credentials**
   - Authorization code

8. **Authorization code**

9. **Open redirect URL**

10. **Open redirect URL with client identifier**

11. **Resource Owner**
12. **Client**
13. **User-Agent**
14. **Authorization Server**
15. **Resource Server**
Native App Flow

Mainly: Proof Key for Code Exchange - PKCE (RFC 7636)
Authorization Code Grant Flow: Flaws

1. Open Client (App)
2. Present authorization UI
3. Present credentials
4. Open redirect URL with client identifier
5. Validate request
6. Present authorization UI
7. Present submitted credentials
8. Authorization code
9. Open redirect URL with client identifier

Resource Owner
User-Agent
Authorization Server
Resource Server

Open redirect URL with client identifier
Present authorization UI
Present credentials
Authorization code
RFC 8252: OAuth 2.0 for Native Apps

- External User Agent:
  - External browser/app
  - In-App browser tab
Authorization Code Grant Flow: Native App

1. Generate random `code_verifyer`
2. `code_challange=sha256(code_verifyer)`
3. Open redirect URL with client identifier & `code_challange`
4. Validate request & store `code_challenge`
5. Present authorization UI
6. Present submitted credentials
7. Authorization code
8. Authorization code & `code_verifyer`?
9. Verify `sha256(code_verifyer) == code_challenge`
10. Deny access
Further Security Considerations

- **URI-Schema:**
  - Domain-Related, e.g. com.fhunii.eventmarketing
  - Prevent DNS-Spoofing: Use 127.0.0.1 instead of localhost by performing redirection on localhost (Desktop)

- **Defence against cross-app request forgery:**
  - Usage of the *state* parameter with a random

- **Embedded User Agent (Web-View):**
  - Must open an external browser as the embedded user agent has full access to authorization grant
Agenda

- Introduction
- Flows
- Conclusion
Conclusion

● Choose the flow based on the use case
  ● App: Auth. Code Grant + Native Apps
  ● Web: Implicit Flow
Questions?

oauth2019@pagel.pro
Implementation Flaws

Store username and generate password in the client after authentication
Implementation Flaws

Storing the username/password locally
Choose an account
to continue to OWASP Juice Shop

T
Timo Pagel
timo.pagel@owasp.org

T
Timo Pagel
google@timo-pagel.de

Use another account
<table>
<thead>
<tr>
<th>Product</th>
<th>Description</th>
<th>Pri</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="mailto:admin@juice-sh.op">admin@juice-sh.op</a></td>
<td>0192023a7b6d73250516f6918b500</td>
<td>1</td>
</tr>
<tr>
<td><a href="mailto:bender@juice-sh.op">bender@juice-sh.op</a></td>
<td>0c30e517e3a95aabf1bbfc6744a4ef</td>
<td>3</td>
</tr>
<tr>
<td><a href="mailto:bjoern.kimminich@googlemail.com">bjoern.kimminich@googlemail.com</a></td>
<td>448af65cf28e8adeab7ebb1ecff65f15</td>
<td>4</td>
</tr>
<tr>
<td><a href="mailto:ciso@juice-sh.op">ciso@juice-sh.op</a></td>
<td>861917d5fa5f1172f931dc700d81a8fb</td>
<td>5</td>
</tr>
<tr>
<td><a href="mailto:google@timo-pagel.de">google@timo-pagel.de</a></td>
<td>421a487b68a3a4f057d34996968c5e2a</td>
<td>7</td>
</tr>
<tr>
<td><a href="mailto:jim@juice-sh.op">jim@juice-sh.op</a></td>
<td>e541ca7e6c7208d1280474f613e5e45</td>
<td>2</td>
</tr>
<tr>
<td><a href="mailto:support@juice-sh.op">support@juice-sh.op</a></td>
<td>d57386e76107100a7d6c2782978b2e7b</td>
<td>6</td>
</tr>
</tbody>
</table>
Implications

- Endless Refresh?
- No Caching for shared proxies with Authentication-Header
- Logout -> Invalidation of Refresh/Access-Tokens
- Monitoring of unauthorized invalid Tokens usage attempts
- No-Algo Attack
Agenda

- Introduction
- Flows
- Implementation Flaws
- Conclusion
Conclusion

● OAuth2 is used to delegate access
● Choose the right flow for your use case
● OAuth2 does not prevent from thinking on your own! -> harden endpoints/processes
## Risk Overview

- **Resource Owner Password Credentials Flow**: Browser / Mobile App - Critical (with public clients)
- **Authorization Code Flow**: Confidential Client - Medium-High
- **Implicit Flow**: Browser (JavaScript) - Medium-High
- **Authorization Code Flow (PKCE)**: Mobile App - Medium

[https://auth0.com/docs/api-auth/which-oauth-flow-to-use](https://auth0.com/docs/api-auth/which-oauth-flow-to-use)
OAuth ROPC-Specification

It is also used to migrate existing clients using direct authentication schemes such as HTTP Basic or Digest authentication to OAuth by converting the stored credentials to an access token.

Source: RFC 6749 The OAuth 2.0 Authorization Framework - Section 4.3
Hardening Resource Owner Password Credentials Flow (not recommended) 1/2

● Harden Token Endpoint:
  ● Do not allow cross-domain requests
  ● Brute Force / “Token Brute Force”
  ● Timing Attacks
  ● Lack of security sensitive information
  ● Throttling Policy
  ● ...

● Reduce Risk of Stolen Tokens:
  ● TLS
  ● Disable refresh tokens and use short lived access tokens
  ● Reconsider lifetime of tokens
Hardening Resource Owner Password Credentials Flow (not recommended) 2/2

- Inform resource owners about password reuse
- Limit usage to org. where client/application and authorizing service are from the same org.
- The authorization server may generally restrict the scope of access tokens issued by this flow