Integrating SRAM OIDC Authentication in a React JavaScript Application

Introduction

This guide aims to provide a detailed walk-through for integrating OIDC (OpenID Connect) authentication with SRAM (Surf Research Access Management) in a React JavaScript application. By the end of the guide, you'll achieve the following:

- Prepare your SRAM account, create a new collaboration (read more here), and register your service.
- Understand the conceptual use of authentication and authorization in JavaScript using React and OIDC.
- Implement OIDC authentication in a React JS application.
- Enable seamless user login through SRAM using OIDC.

Prerequisites

Before you begin, make sure you have:

- Node.js and npm installed on your machine.
- An account and a collaboration on SRAM.
- A SRAM service (application/platform) registered with your collaboration as a public client.
- Client ID (APP-ID) from your SRAM service registration.
- Callback URL registered in your SRAM service.

Setting up SRAM

Before diving into the integration process, ensure you have properly set up SRAM:

- **Register with SRAM**: Create an account on SRAM for necessary credentials and management tools.
- **Register your service**: Register your service with SRAM and specify callback URLs, desired scopes, and other configurations.
- **Configure callback URL with SRAM**: Submit your callback URL to SRAM for user redirection after authentication.
- **Integrate with SRAM**: Use provided credentials (App-Id) and endpoint details to prepare for your service's authentication flow with SRAM.

Step-by-step Implementation

Create a New React JS Project

Run the command:

```
npx create-react-app my-sram-app
```

Instal OIDC Client Library

Run the command:

```
npm install oidc-client-ts
```

Create OIDC Configuration File

Create a file named ‘OIDC-config.js’ in your project’s ‘**src**’ directory. Populate it with the OIDC configuration settings.

```
import { WebStorageStateStore } from "oidc-client-ts";
export const OidcConfig = {
  authority: "https://proxy.acc.sram.eduteams.org/",
  client_id: "Your-EntityId-Goes-Here",
  redirect_uri: `${window.location.origin}/oidc_callback`,
  scope: "openid profile",
  loadUserInfo: true,
  userStore: new WebStorageStateStore({ store: window.localStorage })
};
```

Here, ‘authority’ is the URL of the SRAM OIDC server, and ‘client_id’ is the App-ID you received from SRAM. The ‘redirect_uri’ is where SRAM will send the user after authentication.

Implement OIDC in App.js

Import the OIDC configuration and UserManager class in your App.js file. Initialize the UserManager with your OIDC configuration.

```
import { UserManager } from 'oidc-client-ts';
import { OidcConfig } from './OIDC-config';
import { useState, useEffect } from 'react';

const client = new UserManager(OidcConfig);
```

Add Login and Logout Functionality

Add functions to handle user login and logout. These functions will interact with the UserManager to perform the respective actions.

```
function signIn() {
  client.signinRedirect().then(()=>{});
}

function signOut() {
  client.removeUser().then(() => {
    setUser(null)
  })
}
```

Handle OIDC Callback

When SRAM redirects the user back to your application, you need to handle the OIDC callback. This function will process the authentication response from SRAM.

```
function handleSigningCallback() {
  client.signinRedirectCallback().then((_user) =>{})
  .catch((err) => {
    console.log(err);
  }).finally(() => {
    window.location.href = window.location.origin;
  })
}
```

Add UI Elements for Login and Logout

Add buttons to your UI to allow the user to log in and log out.

```
return (
  <div className="App">
    <header className="App-header">
      <p>SRAM OIDC Example React App</p>
      {user != null && <>
          <p>Welcome, {user.profile.name}</p>
          <button onClick={signOut}>Log out</button>
      </>}
      {user == null && <>
          <p>You are not logged in</p>
          <button onClick={signIn}>Log in</button>
      </>}
    </header>
  </div>
);
```

Run Your Application

Finally, run your application to see the OIDC authentication in action.

```
npm start
```

Additional Steps

Mapping Attributes from SRAM

SRAM has many attributes that come from the underlying research or educational institute. They can be found here. We can use the ones in the column 'OIDC claim'. To map these attributes, extend your OIDC configuration as follows:

```
// Extend your OIDC-config.js
export const OidcConfig = {
  // ...existing config
  scope: "openid profile eduperson_entitlement",
};
```

Then, in your application, you can access these additional claims from the user object.

```
// Accessing additional claims
function displayUserDetails() {
  client.getUser().then(user => {
    console.log(user.profile.eduperson_entitlement);
  });
}
```

Role-based Authentication with SRAM Roles

You can implement role-based authentication by utilizing the roles provided by SRAM. Here's how you can protect a route in a React application:

```
// Inside your React component
import { Redirect } from 'react-router-dom';

function ProtectedRoute() {
  const user = // get user from state or context

  if (user && user.profile.role === 'Admin') {
    return <AdminComponent />;
  } else {
    return <Redirect to="/login" />;
    }
  }
```

Code-Level Authorization

You can enforce authorization directly in your code by creating a custom hook or function that checks the user's role.

```
// Custom hook for authorization
function useAuthorization(requiredRole) {
  const user = // get user from state or context

  if (user && user.profile.role !== requiredRole) {
    throw new Error('Unauthorized');
  }
}
```

Use this hook in your components to enforce role-based authorization.

```
function AdminComponent() {
  useAuthorization('Admin');
  // ... rest of the component
}
```

Conclusion

By following the above steps, you have successfully set up OIDC authentication with SRAM in a React JS Single Page Application (SPA).

Because this is a SPA, it's important to note that we didn't use a client secret. Instead, we relied on PKCE (Proof Key for Code Exchange) for secure authentication. Make sure to request that SURF marks your client as public to enable CORS requests and PKCE-based authentication.

Note: Remember, the configuration provided here serves as a foundational example. Always tailor the configuration and code to meet your application's specific requirements and security considerations. Keep your APP-ID confidential to ensure the security of your application.