Secure Authentication Model for Public Clients

Bohdan Bodak[1], Anatoliy Doroshenko[1,2]

[1] *National Technical University of Ukraine "Kyiv Polytechnic Institute", Kyiv, Ukraine*
[2] *Institute of Software Systems of the National Academy of Sciences of Ukraine, Kyiv, Ukraine*

Abstract

The paper focuses on authentication and authorization in public clients and provides a secure model as an alternative to costly Microsoft Duende BFF solution. After giving a brief overview of confidential and public clients in terms of authorization, we have analyzed problems and potential attack vectors associated with the authorization process in public clients due to their inability to hold credentials securely. Confidential clients are implemented on secure servers or able to facilitate secure authentication by other means, while public clients lack this security. Our research discovered algorithms, models, and methods for secure authorization in public clients. As a part of our model, we have implemented high entropy Proof Key for Code Exchange generator in C# .NET 6.0. In addition, we have provided a solution to a problem of storing sensitive information in public clients using the Backend for Frontend concept. This concept leverages a reverse proxy pattern where a backend application acts as a proxy and handles all client requests. Having a proxy backend application significantly tightens security model for public clients, while restricting possible attack vectors. The authorization model being researched was based on Proof Key for Code Exchange and Backend for Frontend approach. During the testing phase of our research, we have confirmed that the model was not vulnerable to Cross-Site-Scripting and Auth Code Interception attacks. A sequence diagram outlining main actors and interactions among them in context of authorization has been designed. The diagram stands as the visual representation of the model that uses proposed methods and algorithms. As a result, we have managed to build an alternative to secure authorization solutions for public clients that do not rely on the client secret. We have summarized our key findings in a Blazor Web Assembly application, which is classified as public and uses the described authentication model via a shared library.

Keywords

Authorization, authentication, PKCE, Proof Key for Code Exchange, Identity Server, public application, OAUTH, XSS, information security.

1. Confidential and public clients

According to OAUTH 2.0 specification [1] clients can be classified as either confidential or public. Confidential clients usually facilitate security due to them being implemented on a protected server or by having other means to secure credentials. Public clients on the other hand are unable to hold user's credentials securely because such clients are downloaded and executed directly in an unsecure environment: a browser or a mobile operating system. Examples of public clients are Single Page Applications (SPA) regardless of a technology being used and native iOS or Android applications. It is a common practice to use a client secret as a method to prove a secure authentication, monitor and

[1]13th International Scientific and Practical Conference from Programming UkrPROGP'2022, October 11-12, 2022, Kyiv, Ukraine
EMAIL: bohdan.bodak@outlook.com (A. 1); doroshenkoanatoliy2@gmail.com (A. 2)
ORCID: 0000-0002-4854-2314 (A. 1); 0000-0002-8435-1451 (A. 2)

© 2022 Copyright for this paper by its authors.
Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)
CEUR-WS.org/Vol-3501/s33.pdf

handle auth requests within OAUTH 2.0 context. A client secret is a unique application identifier generated for a client through a process of registration. Every client that uses some kind of an OAUTH 2.0 compliant auth service has to be registered via that service.

Confidential clients store their secret in code or configuration file. However, public clients can not store a secret without revealing it to unauthorized parties. For example, in mobile applications it is not possible to keep a secret in code because an attacker could retrieve it via a de-compilation process. In addition to that, mobile apps are vulnerable to what is called an Auth Code Interception Attack [2] as displayed on Figure 1.

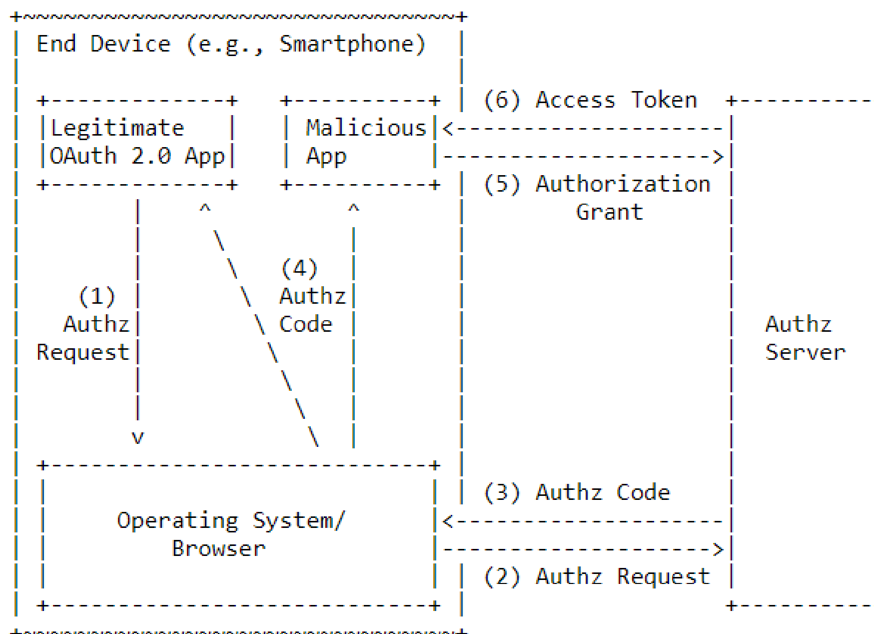

Figure 1: Auth Code Interception Attack

In this example, an attacker registers an additional address via a Malicious App to intercept redirected requests and leverages an ability to intercept an authorization code from the auth server. This example closely resembles a classic Man-In-The-Middle attack and could be viewed as a variation of such. While this particular attack is only feasible for mobile apps, SPA also cannot use a client secret because their entire code is accessible through a browser.

2. Problem with authorization in public clients

From the moment of publication of OAUTH 2.0 specs in 2013, the specification has become widely popular and has been adopted by key industry players such as Facebook, Twitter, and Google, practically becoming a standard [3]. Considering that the standard was first introduced almost a decade ago and technologies are constantly moving forward, there is always a need for new models and methods to protect against security breaches. Formal analysis of popular websites and social networks that use OAUTH 2.0 authentication model has discovered dozens of previously undetected security vulnerabilities [3]. Authors have revealed potential attacks such as Cross-Site Request Forgery (CSRF) and Open Redirectors without going into details about them. Empirical analysis [4] using attack vectors from former work [3] demonstrated a successful interception of an SSO session and evaluated various security concerns associated with it. The article [4] presented a cookie hijacking attack for Facebook, which was used to authenticate in 95 SSO-applications that support OAUTH model. Authors then went ahead and proposed a Single Sign-Off standard to become a part of

OAUTH specification, which mitigates consequences of the problem but does not solve the problem of cookie interception itself.

OAUTH 2.0 security best current practice [5] outlines 9 key attacks together with potential countermeasures. The standard recommends using PKCE [6] for public and confidential clients as an additional level of security against attacks. However, the paper does not state any practical recommendations regarding the implementation, delegating it to application developers. Based on research [3, 4] and practices [5, 7] provided above, developers face a practical problem of implementing a secure authentication mechanism based on PKCE flow, which would not be vulnerable to CSRF, Open Redirectors, Auth Code Interception and other known attacks.

3. Proof Key For Code Exchange (PKCE) algorithm

Considering problems stated above, specification OAUTH 2.0 proposes to use PKCE algorithm [6] instead of client secret, which is basically a variation of proof of possession. Despite our model being developed for public clients, it can be utilized to protect confidential clients as well. Base implementation of this algorithm relies on client providing a proof of code possession to an auth server in exchange for an auth code. PKCE algorithm introduces additional parameters to the auth flow depicted on Pic. 1: code_verifier, code_challenge, and code_challenge method. Code verifier is a randomly generated string used as a form of a secret and generated in accordance with OAUTH 2.0 specs. Code challenge is created on a client by transforming code_verifier value. Code challenge method is optional parameter, which represents the transformation algorithm for code_challenge and can be set to either “S256” (SHA256 algorithm) or “plain”. Specification does not recommend using “plain” as a code challenge method because it can lead to information exposure in case the code challenge is intercepted. In that case, code challenge may be used directly as a code verifier, rendering PKCE flow entirely useless. “Generating codes for PKCE algorithm” chapter provides detailed example of generating code challenge and code verifier according to OAUTH 2.0 specification using C# .NET 6.0. Generic authorization model using PKCE algorithm is depicted on Figure 2.

Figure 2: Authorization model with PKCE algorithm

Basic authorization model with PKCE algorithm can be broken down into the following steps:

a) Client generates code_challenge and code_verifier; sends an authorization request that includes code_challenge and code_challenge_method.

b) Authorization server saves code_challenge together with the method and sends an authorization code back to client.
c) Client sends a request for an access token and includes the auth code with code_verifier.
d) Auth server checks code_verifier against a stored code_challenge and returns an access token in case the verification was successful.

This algorithm works successfully because even if attacker manages to intercept the auth code, it is impossible to exchange the code for an access token without code_verifier, which prevents Auth Code Interception attack displayed on Pic. 1. Worth mentioning that a pair of code_challenge and code_verifier must be unique and valid for one time use only. It is equally important to store code_verifier securely between authorization and access token requests. We dedicated a chapter “Methods of storing code_verifier in public clients” to solving this problem. In addition, a client has to use a TLS secure channel for data transfer.

4. Generating codes for PKCE algorithm

For generation of code_verifier and code_challenge a class named PkceGenerator was created, which has methods GenerateCodeVerifier and GenerateCodeChallenge respectively. A constructor of this class accepts a parameter representing the size of code_verifier. A minimum size is 43 symbols while maximum is 128 according to the OAUTH 2.0 specification. C# .NET 6.0 with System and System.Security.Cryptography assemblies was used for developing this class.

```
namespace DemoApp.Authentication {
  /// <summary>
  /// Provides a randomly generating PKCE code verifier and it's corresponding code challenge.
  /// </summary>
  internal class PkceGenerator {
    /// <summary>
    /// The randomly generated PKCE code verifier.
    /// </summary>
    public string CodeVerifier;

    /// <summary>
    /// Corresponding PKCE code challenge.
    /// </summary>
    public string CodeChallenge;

    /// <summary>
    /// Initializes a new instance of the Pkce class.
    /// </summary>
    /// <param name="size">The size of the code verifier (43 - 128 charters).</param>
    public PkceGenerator(uint size = 128) {
      CodeVerifier = GenerateCodeVerifier(size);
      CodeChallenge = GenerateCodeChallenge(CodeVerifier);
    }
   …
  }
}
```

According to OAUTH 2.0 RFC-7636 specification, code_verifier must be a cryptographically strong high-entropy string with length from 43 to 128 symbols. Accepted symbols are A-Z, a-z, 0-9, -, ., _, ~. Proposed implementation compliant with specs can be found below.

```
/// <summary>
/// Generates a code_verifier according to RFC-7636.
```

```
/// </summary>
/// <param name="size">The size of the code verifier (43 - 128 charters).</param>
/// <returns>A code verifier.</returns>
public static string GenerateCodeVerifier(uint size = 128) {
  if (size < 43 || size > 128)
   size = 128;
   const string unreservedCharacters =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-._~";
      Random random = new Random();
      char[] highEntropyCryptograph = new char[size];

      for (int i = 0; i < highEntropyCryptograph.Length; i++) {
        highEntropyCryptograph[i] = unreservedCharacters[random.Next(unreservedCharacters.Length)];
      }

      return new string(highEntropyCryptograph);
}
```

Code challenge is created by hashing code_verifier using a SHA256 algorithm. A method presented below hashed code_verifier with SHA256 and replaces unacceptable symbols making the value usable in a URL (doing a so-called URL Encoding).

```
/// <summary>
/// Generates a code_challenge according to RFC-7636.
/// </summary>
/// <param name="codeVerifier">The code verifier.</param>
/// <returns>A code challenge.</returns>
public static string GenerateCodeChallenge(string codeVerifier) {
      using var sha256 = SHA256.Create();
      var challengeBytes = sha256.ComputeHash(Encoding.UTF8.GetBytes(codeVerifier));

      return Convert.ToBase64String(challengeBytes)
      .Replace('+', '-')
      .Replace('/', '_')
      .Replace("=", "");
}
```

Using this class in the application code is fairly straightforward: developer has to create an instance of this class and access an appropriate property to get code_verifier or code_challenge value. Simple and lightweight design of the class ensures proper usage. An example is presented below.

```
PkceGenerator pkceGenerator = new ();
string codeVerifier = pkceGenerator.CodeVerifier;
string codeChallenge = pkceGenerator.CodeChallenge;
```

Developers need to keep in mind that each pair of generated code_challenge and code_verifier must be used only once to ensure security. Applications should not attempt to utilize the same pair of codes because it could lead to potential interception and unfair usage of such codes by attacker in a pursue to retrieve an auth code. OAUTH 2.0 specification and RFC-7636 do not provide any information or best practices regarding code_verifier storage, so in the next chapter we are set to outline main methods of storing code_verifier in applications because it is fundamental to PKCE authorization flow.

5. Methods of storing code_verifier in public clients

As it was stated before, PKCE algorithm specification does not propose any methods of storing code_verifier in between requests. Due to the fact that client receives an auth code via a callback, it is not possible to save code verifier in a variable or by other means in code. We are going to examine methods for storing code_verifier value in terms of WEB-applications (React, Angular, Web Assembly). Additional methods of secure data storage for Android [8] and iOS [9] mobile apps are different and lay beyond the scope of this paper.

5.1. Storing a value in browser storage

Browser storage is the most obvious solution when dealing with information storage in WEB-clients. There are two types of storage in a browser: Local Storage and Session Storage [10]. By means of simple JavaScript code accessing a browser's API, developers can save data and user's configuration. A key difference between Local and Session storage is a persistence of data. Session Storage does not keep data between different tabs or windows of the same browser giving developers a false sense of security and confidence to save any value there. Unfortunately, if a website is vulnerable to Cross-Site-Scripting (XSS) attacks, an attacker could read information from browser storage regardless the storage type: Session or Local.

Pros of storing a value in browser storage:

- Quick read/write operations via native browser API.
- Easy implementation with JS.

Cons of such approach:

- Vulnerable to XSS attacks.
- Potential for a data leak.

It is worth mentioning that modern frameworks such as Angular and React do a fair share of work and provide methods for protecting against XSS-attacks. Nevertheless, there is always a possibility of sensitive data leak when this data is being stored in a browser.

5.2. Storing in JavaScript cookies

Another approach to information storage in a browser are cookies [12]. JavaScript code is used in this method as in the previous one, whereby the document.cookie is called to store and retrieve data from cookies. Pros and cons are practically the same as in the approach outlined above. There is a simple fact every WEB developer has to understand: if they can do something in their JavaScript code – the attacker can do it as well. Therefore, instead of investing resources into complicated algorithms and methods to protect information in browser storage or cookies, WEB developers should look into protecting applications from XSS attacks while finding different approaches to keep critical information such as tokens secure.

5.3. Storing a value using Server cookies

Server applications are able to set cookies in response to a client, which cannot be retrieved via JavaScript. Such cookies are added to a response with HttpOnly flag [13]. In addition, it is strongly recommended for a server to set SameSite attribute as Lax or Strict [13] to avoid a situation where cookies are being sent to third-party domains. If the SameSite attribute is not set and browser does not support default setting to SameSite.Lax for SameSite.None cookies, then a so-called Hidden iframe attack could be executed sending cookies to unwanted websites. Assuming that an application is vulnerable to XSS-attack and has HttpOnly cookie without SameSite attribute, an attacker could embed a hidden iframe into website's DOM, which sends requests to a third-party service. All cookies without SameSite or SameSite.None would end up on attacker's website in this case.

Pros of Server cookies approach:

- Makes XSS attack impossible if configured correctly.
- The most secure method for storing sensitive information.

Cons of this approach:

- A backend service hosted on the same domain as the client is mandatory in order to set HttpOnly cookies with SameSite.Strict attribute.
- Deployment process could become complicated because it is required to deploy both a client and a server application.

6. Implementing a Backend For Frontend (BFF) method with PKCE authorization

After applying an approach to store data in Server cookies to the process of PKCE authorization, we will get a method which is called a Backend For Frontend (BFF). This method has gained a great share of popularity and appreciation within development community in past few years for securing public applications with PKCE flow. Some companies already provide ready-to-use BFF solutions for applications. As an example, Identity Service developers – Duende (which is not a part of Microsoft) provides a Duende BFF [14] solution as a module for their Identity Service, which is available to corporations with Enterprise subscription. The main advantage of this solution is rich functionality available to developers and compatibility with existing Identity Service solutions. Among disadvantages are the steep price for the Enterprise subscription and integration with only latest Identity Service 5 providers, which limits the scope of this solution.

Considering an absence of other solutions on the market and the high price of the corporate Microsoft Enterprise subscription, it had been decided to develop our own version of BFF using C# .NET 6.0 which works with OAUTH 2.0 compliant clients. Let us review an example of authorization flow in a system that includes Blazor Web Assembly [15] SPA client and BFF service written using C# .NET 6.0. We are not going to concentrate on a OAUTH 2.0 authorization service because it is out of scope for this article.

A step-by-step authorization process with PKCE algorithm and BFF approach can be described in these steps:

1. A user initiates authorization flow by pressing "Login" button in the application.
2. Application generates code_challenge and code_verifier using available methods from "Generating codes for PKCE algorithm" chapter.
3. Application sends code_challenge, code_verifier and return_url to BFF service via a POST request.
4. BFF service adds code_verifier to Response.Cookies with HttpOnly, Secure, and SameSite.Strict attributes.
5. BFF service redirects a request to authorization server /authorize method with code_challenge and return_url.

6. Authorization server directs a user to a login page where the user enters credentials and provides consent.
7. Authorization handles the login result, saves code_challenge, and generates auth_code.
8. Authorization server redirects request back to return_url adding an auth_code.
9. Client sends a POST request to BFF for authorization token adding an auth_code to request parameters.
10. BFF reads code_verifier from cookies, adds code_verifier and auth_code and calls authorization service to get a token. BFF can delete code_verifier from cookies at this point.
11. Authorization server verifies code_verifier against stored code_challenge, auth_code and other parameters such as requestor's IP address.
12. After a successful verification, auth server issues an auth token and sends it back.
13. BFF receives authorization token, writes it in cookies as HttpOnly, Secure, SameSite.Strict.
14. BFF returns successful result to a client application.
15. Client requests user's information via BFF and restores state.

Authorization flow is depicted in details on Figure 3.

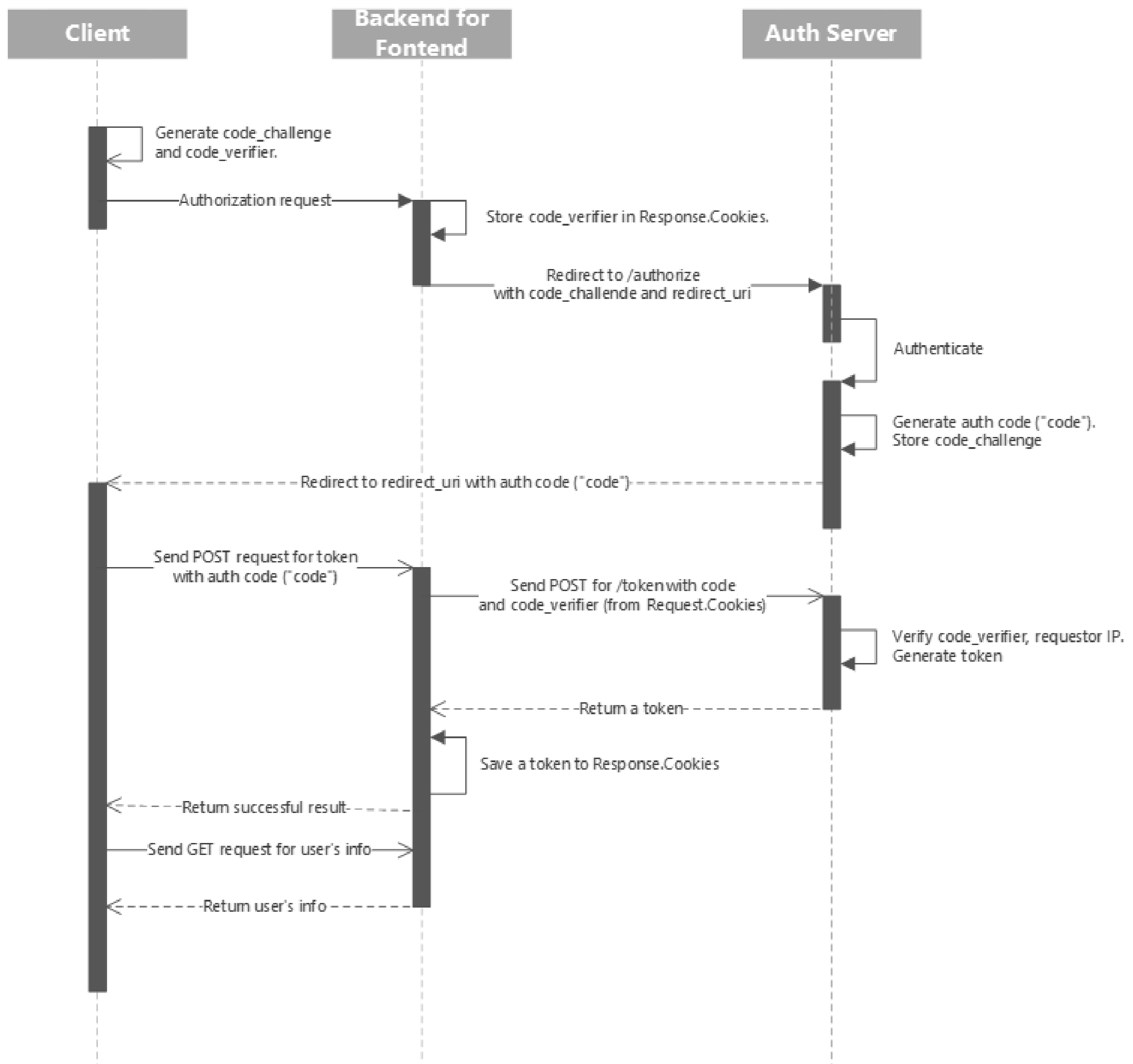

Figure 3: Authorization flow diagram with PKCE algorithm and BFF method

Therefore, after successful authorization, a client is authorized to call protected BFF methods designated for authorized users only. At the same time, a client does not contain any sensitive information which could be vulnerable to XSS-attacks. Secure data such as code_verifier or authorization token is stored in HttpOnly cookies with SameSite.Strict as presented in the chapter above. In this model BFF acts as a reverse proxy pattern [16] that routes requests and handles client authorization.

7. Shared authentication library

In order to ensure code reusability and scalability of this project, we have built a shared library, which uses our proposed model. This common project handles authentication and authorization for .NET Core API and Blazor applications. Our library can be packaged as a NuGet package and distributed via Azure Artifacts or similar package distribution platform. Customers will simply install the necessary packages in their projects, complete the configuration, and be able to facilitate secure authentication in public single-page applications. This paragraph provides an overview of the library as well as configuration that is required for the setup.

7.1. Shared library structure

Our shared library is called BFFLight and it consists of two projects: BFFLight.Wasm and BFFLight.Server.

The first project – BFFLight.Wasm has all necessary classes and extensions to setup secure authentication in Blazor Web Assembly client application. Project's structure is the following:

- Authentication (folder with authentication interfaces, classes, and models)
  - ProfileResponse.cs (model for user's info response)
  - BFFLightPostLogin.razor (post-login component, gets user's info, restores state)
  - PkceGenerator.cs (class that generates code_verifier and code_challenge, described in chapter 4)
  - SsoConnector.cs (a service that handles HTTP requests to a backend API for authentication, user's info, token exchange, etc.)
  - SsoClientOptions.cs (model for client's configuration)
  - TokenResponse.cs (model for the access_token response)
  - IBFFLightSignInManager.cs (interface for SignIn, SignOut, and GetClaimsPrincipalAsync methods)
  - BFFLightSignInManager.cs (implementation of the interface above, uses the SsoConnector, PkceGenerator to handle sign in, sign out, and get user's information)
- Extensions (folder with extension classes and methods)
  - IServiceCollectionExtensions.cs (provides an extension method AddBFFLightWasmAuth that configures options and injects required services during application start)

The second project is BFFLight.Server targets .NET Core 6.0 API and has necessary controllers, classes, and extensions for secure server authentication. The project structure is described below:

- Controllers (folder with API controllers)
  - SsoController.cs (main authentication controller with Token, GetProfile, and Logout methods; uses ISsoService implementation)
- Models (folder with models)
  - SsoServerOptions.cs (model for server's configuration)
  - UserInfoModel.cs (model for user's info)
- Services (folder with services)

  - ISsoService.cs (authentication service interface with methods GetAuthToken, GetProfile, Logout)
  - SsoService.cs (default implementation of the ISsoService)
- Extensions (folder with extension classes and methods)
  - IServiceCollectionExtensions (provides an extension method AddBFFLightServerAuth that configures options and injects required services during application start)

7.2. Shared library configuration

Before developers begin to use this library, they first need to make sure that their application is registered with an authentication provider. As a result of this registration process, they will get a Client ID and Client Secret. In addition, developer also needs to know the URL address of an OpenId SSO provider they use. The setup process begins with the installation of BFFLight.Wasm and BFFLight.Server to Blazor Web Assembly and .NET Core API projects respectively.

Let us review client's configuration first. After installing NuGet packages, a developer should open Program.cs file in their Blazor project to add the following lines before the await builder.Build().RunAsync():

```
builder.Services.AddBFFLightWasmAuth(); // configure BFFLight services
builder.Services.AddAuthorizationCore(); // add default Core authorization if not already added
```

Next, a developer should open wwwroot/appsettings.json file (create a new file if it does not exist) and add the following configuration keys:

```
"SsoClientOptions": {
  "ClientId": "sso_provider_client_id", // id given to you by the SSO provider after registration
}
```

Finally, a developer should open the App.razor file and wrap all content in a <CascadingAuthenticationState></CascadingAuthenticationState> tag. This is a built-in Blazor tag used to propagate an authentication state down components' tree. With the CasadingAuthenticationState, developers will be able to inject a [CascadingParameter] Task<AuthenticationState> parameter to then get the auth state and user's information in their pages.

After completing client's configuration, developers can move on to the server project setup. First, they should introduce a call to initialize BFFLight services in Program.cs file before the var app = builder.Build() line:

```
builder.Services.AddBFFLightServerAuth(); // configure BFFLightServices
```

Moving on from the Program.cs setup, a developer should open appsettings.json file in order to add these configuration keys:

```
"SsoServerOptions": {
  "ClientId": "sso_provider_client_id", // id given to you by the SSO provider after registration
  "ClientSecret": "sso_provider_secret", // secret key provided by SSO after registration
  "SsoServiceUrl": "sso_url", // URL of the SSO provider
}
```

This completes the configuration process. Following these steps and using our BFFLight library will allow developers to implement secure authentication and authorization in Blazor single page public client.

8. Conclusions

In this paper we have outlined key differences between confidential and public clients in terms of secure authorization, provided analysis of problems associated with secure authorization in public clients using OAUTH 2.0 protocol. During the analysis stage of this research, we have managed to discover algorithms, models, and methods for implementing secure authorization in public clients. In addition, we have performed practical modeling and implementation of Proof Key for Code Exchange algorithm with Backend For Frontend method. Finally, we have developed a proof-of-concept public client that uses our shared BFFLight library with the model and algorithm for secure authorization and is not vulnerable to attacks mentioned in the analysis phase: Cross-Site Scripting (XSS), Auth Code Interception, CSRF, etc.

9. References

[1] Microsoft Internet Engineering Task Force (IETF), The OAuth 2.0 Authorization Framework, 2012. URL: https://datatracker.ietf.org/doc/html/rfc6749#section-2.1.
[2] OWASP Project, Testing for OAuth Client Weaknesses, 2022. URL: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/05-Authorization_Testing/05.2-Testing_for_OAuth_Client_Weaknesses.
[3] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, S. Maffeis, Discovering concrete attacks on website authorization by formal analysis, volume 22 of Journal of Computer Security (4th. ed.), 2014, pp. 601-657. doi:10.3233/JCS-140503.
[4] M. Ghasemisharif, A. Ramesh, S. Checkoway, C. Kanich, J. Polakis, O Single Sign-Off, Where Art Thou? An Empirical Analysis of Single Sign-On Account Hijacking and Session Management on the Web, in 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 1475-1492.
[5] T. Lodderstedt, J. Bradley, A. Labunets, D. Fett, OAuth 2.0 Security Best Current Practice (draft-ietf-oauth-security-topics-16), Internet Engineering Task Force (IETF). URL: http://www.watersprings.org/pub/id/draft-ietf-oauth-security-topics-06.html.
[6] Google Internet Engineering Task Force (IETF), Proof Key for Code Exchange by OAuth Public Clients. URL: https://datatracker.ietf.org/doc/html/rfc7636.
[7] T. Lodderstedt, M. McGloin, P. Hunt, RFC 6819: OAuth 2.0 threat model and security considerations, Internet Engineering Task Force (IETF), 2013, pp. 1-71.
[8] Android Developers Documentation, App security best practices. URL: https://developer.android.com/topic/security/best-practices#safe-data.
[9] Apple Developers Documentation, Encrypting Your App's Files. Protect the user's data in iOS by encrypting it on disk. URL: https://developer.apple.com/documentation/uikit/protecting_the_user_s_privacy/encrypting_your_app_s_files.
[10] Mozilla Development Documentation, Window.localStorage. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage.
[11] OWASP Community, Cross Site Scripting (XSS). URL: https://owasp.org/www-community/attacks/xss/.
[12] Mozilla Development Documentation, Using HTTP cookies. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies#security.
[13] Mozilla Development Documentation, Same Site cookies. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite.
[14] Duende Software, BFF Security Framework. URL: https://docs.duendesoftware.com/identityserver/v5/bff/.
[15] Microsoft Documentation, ASP.NET Core Blazor. URL: https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-6.0.
[16] A. Chiarelli, Building a Reverse Proxy in .NET Core, Auth0 Blog. URL: https://auth0.com/blog/building-a-reverse-proxy-in-dot-net-core/.