SECURE OPENID CONNECT IMPLEMENTATION USING AZURE ACTIVE DIRECTORY AND ASP .NET CORE

PETRO KOLOSOV AND DMITRIJ KUDRYASHOV

Abstract. In this manuscript we discuss the problem of secure storage and transfer of access tokens between microservices. Web browser may store access tokens both, in local storage or in cookie files. We propose a secure implementation to store and transfer auth cookies between microservices using Azure Active Directory, OpenID Connect and ASP .NET Core.

Contents

1. Definitions

- **Access Token** is credential used to access protected resources. An access token is an opaque string representing an authorization issued to the client. The string is usually opaque to the client. Tokens represent specific scopes and durations of access,

Date: July 9, 2023.

Key words and phrases. OpenID Connect, OIDC, Azure Active Directory, PKCE, OAuth 2.0, XSS, CSRF, ASP .NET Core .

granted by the resource owner, and enforced by the resource server and authorization server.

- **Refresh Token** is credential used to obtain a new pair of access and refresh tokens when the becomes invalid or expires, or to obtain additional access tokens with identical or narrower scope (access tokens may have a shorter lifetime and fewer permissions than authorized by the resource owner). Refresh tokens are issued to the client by the authorization server.
- **Resource Owner** is an entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.
- **Resource Server** is the server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.
- **Client** is an application making protected resource requests on behalf of the resource owner and with its authorization. This term does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices).
- **Authorization Server** is the server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

2. Statement of the problem

In this manuscript we discuss the problem of secure storage and transfer of access tokens between microservices. Web browser may store access tokens both, in local storage or in cookie files. Local storage is a web browser mechanism that allows web applications to store data locally on the user's device. It is important to note that Local storage is vulnerable to Cross Site Scripting (XSS) attacks [1]. Cross Site Scripting (XSS) is a type of attack such that malicious JavaScript code is injected into an html-page to access user's sensitive data, for example access tokens. Cross Site Scripting (XSS) attacks may be divided by following groups:

- **Reflected XSS** is a type of attack in which a malicious the script is passed to the web server via URL or form parameters and then returned back to the page's html code without proper filtering or escaping. If the user opens the page, then the script is executed in the browser, which can lead to the loss of sensitive data, such as access tokens.
- **Stored XSS** is a type of attack in which a malicious script is stored on the server, for example in the database and displayed on web pages. The script is executed in users' browsers, requesting pages with malicious code.
- **XSS in the DOM** is a type of attack such that a malicious script modifies the DOM tree of a web page, running in the user's browser. In most cases, based on modifying the URL string.

Another way to store credentials is to store them in cookie files. Cookies are small pieces of data sent by web server and stored on user's device. Storing access tokens in cookies eliminates potential XSS attacks since that HttpOnly setting makes impossible to read cookies using JavaScript code. Having credentials stored in cookies, HTTP request is performed via JavaScript. If the object { `withCredentials: true` } provided and auth cookies exist, then auth cookies are attached to request, but cannot be accessed from JS code anyway. Example of such request in TypeScript

```
return this.httpClient.post<TokensResponse>(
this.baseUrl + this.sessionsRoute,
command,
{ withCredentials: true });
```

Note that cookie files are vulnerable to Cross-Site Request Forgery (CSRF) attacks [2]. Cross-Site Request Forgery (CSRF) – is an attack such that redirects user to the resource where user has active session. It means that attacker could perform requests to resources on behalf of the user. The main idea of Cross-Site Request Forgery (CSRF) attack is illustrated below

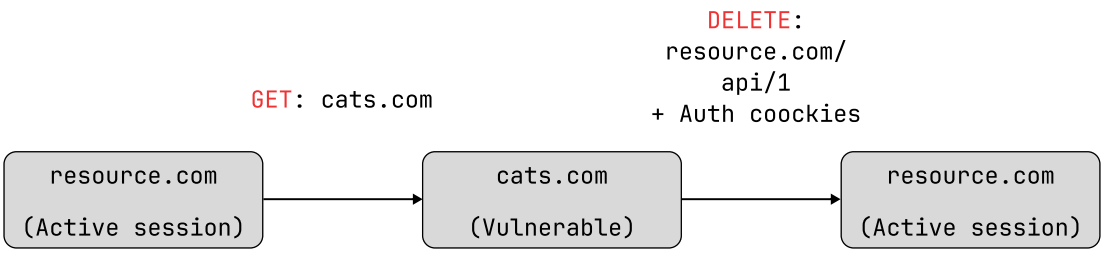

Figure 1. Cross-Site Request Forgery (CSRF) principle diagram.

Cookie files provided with `SameSite` setting that determines whether cookies will be sent along with cross domain requests.

`SameSite` setting has one of the states below:

- **None** – means no restrictions are imposed on the transfer of cookies.
- **Lax** – allows cookie transmission only by secure HTTP methods according to RFC 7231 [3]. These methods are `GET, HEAD, OPTIONS` and `TRACE`.
- **Strict** – blocks cookies from being sent with any requests from third-party resources. Cookies will only be transferred within same domain.

Therefore, `SameSite` setting values such as `Lax` and `Strict` protect user from a CSRF attack blocking submission of cookies using unsecure HTTP methods and cross domain requests. There are more CSRF protection techniques in [4].

3. Introduction to OpenID Connect

OpenID Connect (OIDC) is a simple identity layer [5, 6] on top of the OAuth 2.0 protocol [7]. It enables Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an interoperable and REST-like manner.

OAuth 2.0 is a protocol that enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction

between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.

OAuth 2.0 provides various standardized message flows based on JSON and HTTP; OpenID Connect uses them to provide identity services.

The problem OAuth 2.0 solves is that in the traditional client-server authentication model, the client requests an access-restricted resource (protected resource) on the server by authenticating with the server using the resource owner's credentials. In order to provide third-party applications access to restricted resources, the resource owner shares its credentials with the third party. This creates several problems and limitations [7]:

- Third-party applications are required to store the resource owner's credentials for future use, typically a password in clear-text.
- Servers are required to support password authentication, despite the security weaknesses inherent in passwords.
- Third-party applications gain overly broad access to the resource owner's protected resources, leaving resource owners without any ability to restrict duration or access to a limited subset of resources.
- Resource owners cannot revoke access to an individual third party without revoking access to all third parties, and must do so by changing the third party's password.
- Compromise of any third-party application results in compromise of the end-user's password and all the data protected by that password.

OAuth 2.0 addresses these issues by introducing an authorization layer and separating the role of the client from that of the resource owner. In OAuth, the client requests access to resources controlled by the resource owner and hosted by the resource server. Instead of using the resource owner's credentials to access protected resources, the client obtains an access token. OAuth 2.0 with the Proof of Key Code Exchange [8] flow is displayed on the diagram below

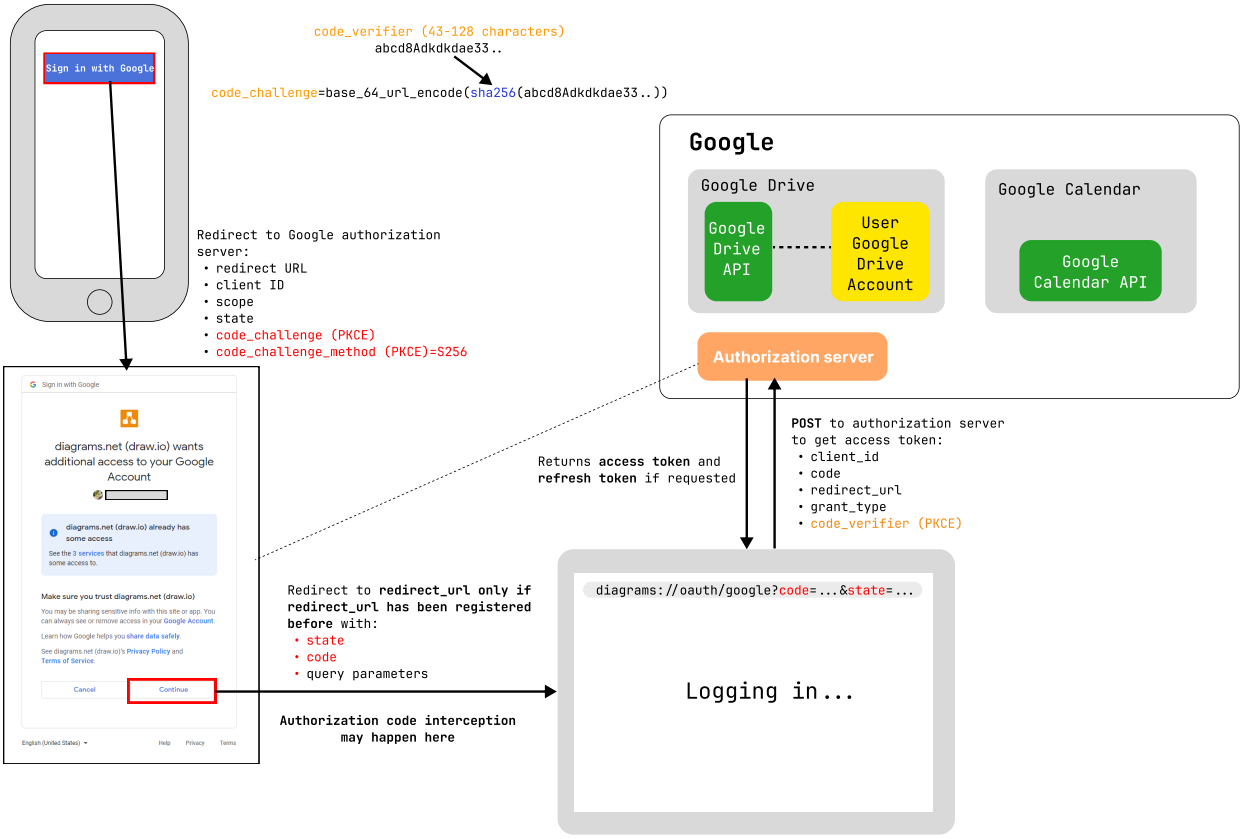

Figure 2. OAuth 2.0 with PKCE flow diagram.

(1) After clicking on the `Sign in with Google` button, the browser is redirected to the authorization endpoint, where the resource owner (user) enters credentials e.g. login and password.

(2) After successful authentication the browser is redirect to `redirect_uri` defined in OAuth provider settings. The `code` parameter is attached to the request parameters.

(3) Having the `code` value application exchanges it to a pair of access and refresh tokens. Provided that we used PKCE with specified `code_challenge` and `code_challenge_method` values exchanging code to a pair of tokens we must pass the value of `code_verifier` to the request.

We mention here such definitions as code and state and they means

- **Code** is an authorization code that is obtained through an authorization server and mediates between clients and resource owners. Before the authorization server

redirects the resource owner back to the client, the authorization server verifies the authenticity of the resource owner. So because the resource owner only authenticates with the authorization server, their credentials are never sent to the client.

- **State** is the value used by the client to store the state between the authorization request and the callback. The authorization server enables this value when redirecting the user agent back to the client. This parameter is used to prevent Cross-Site Request Forgery (CSRF) attacks.

Adding the Proof of Key Code Exchange (PKCE) to the OIDC flow improves protocol security so that the code cannot be exchanged by third-party applications by-passing the original application. Authorization code flow with PKCE is a protocol that represents a client generated secret that can be verified by an authorization server. This secret is called `code_verifier`. The client hashes the `code_verifier` value and writes it to the `code_challenge` parameter of an HTTP request. Proof of Key Code Exchange (PKCE) solves the problem of secure code exchange. If an attacker manages to get an authorization code, then he will not be able to exchange it for access and refresh tokens. Therefore, we ensure that the exchange of a code for tokens is produced by the same application that performed the authentication. Proof of Key Code Exchange can be compared to the digital signature of an authentication process. To exchange the `code` for a pair of access and refresh tokens it is necessary to specify a valid `code_verifier`.

4. Authentication flow

Consider a more practical approach that takes all the previously discussed aspects. Applying modern frameworks like ASP .NET Core, Angular etc. let be the following authentication flow as per diagram below

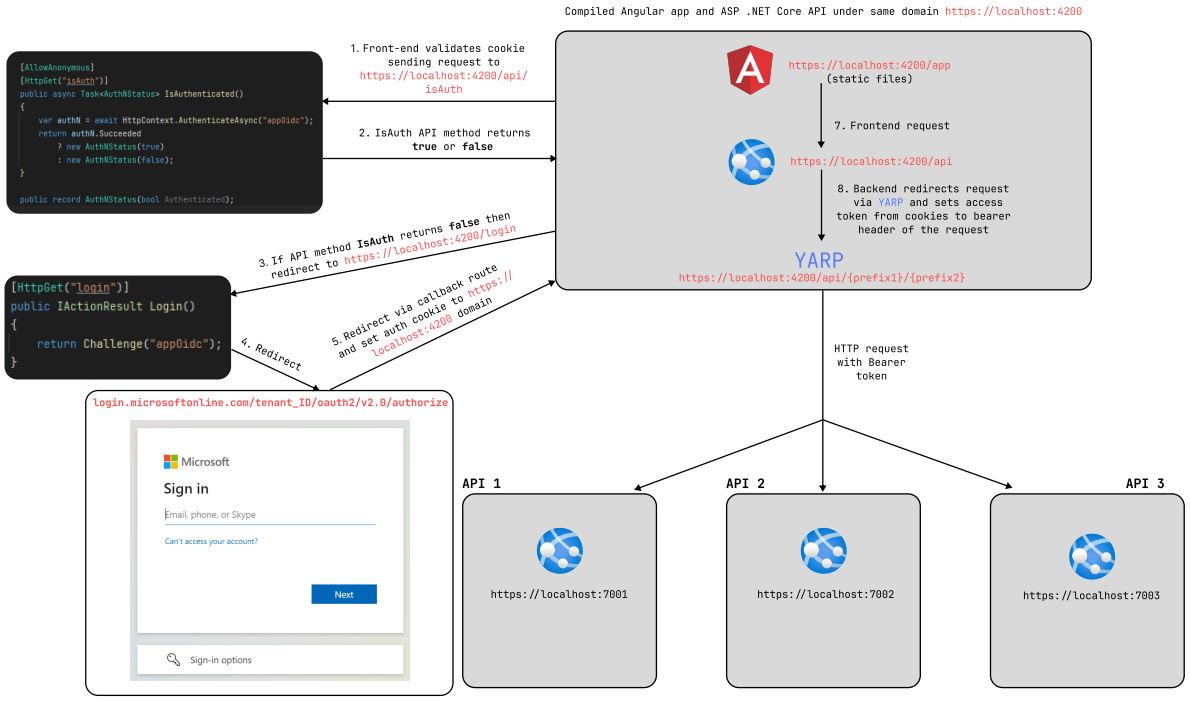

Figure 3. Authentication flow diagram.

Therefore, the whole authentication process can be described as eight steps such that

(1) Compiled Angular frontend application sends request to the authentication endpoint of the ASP .NET Core API to verify current authentication state. Angular application is a set of precompiled bundles that are exposed via same ASP .NET Core API at the `/app` endpoint so that cross-origin requests are not necessary and tokens can be stored in cookie files securely

(2) Authentication endpoint of the ASP .NET Core API responses either with HTTP status code `200 (OK)` or `401 (Unauthorized)`

(3) If `401 (Unauthorized)` status code received from previous step, then browser is redirected to the `login` endpoint of the ASP .NET Core API, otherwise user gets access to the protected resources

(4) Login method of the ASP .NET Core API redirects browser to the Azure AD authorize url `login.microsoftonline.com/tenant/oauth2/v2.0/authorize` where user

enters his credentials. It is important to clarify that in order to get ID token we have to put parameter `openid` to the scope

```
serviceCollection
.AddAuthentication(options => {...})
.AddCookie(CookieAuthenticationDefaults.AuthScheme,
    options => {...})
.AddOpenIdConnect(AuthConstants.AppOidc, options =>
{
    ...
    options.Scope.Add("openid");
});
```

(5) After successful authentication on the Azure AD side, the browser is redirected to the `fallback_url` that is defined in Azure AD application registration. This `fallback_url` is an active endpoint of the ASP .NET Core API. At this point, the `TickerStore` [9, 10] comes into the flow to manage user sessions. Each session is stored as a `UserSessionEntity` entity in the database.

```
public class UserSessionEntity
{
    public Guid Id { get; set; }
    public DateTimeOffset CreatedAt { get; set; }
    public DateTimeOffset ExpiresAt { get; set; }
    public DateTimeOffset UpdatedAt { get; set; }
    public DateTimeOffset DateOfLastAccess { get; set; }
    public byte[] Value { get; set; }
}
```

The Value property of type `byte[]` contains serialized `AuthenticationTicket` [11] object such that contains all required information like access, ID and refresh tokens.

The class `TickerStore` implements `ITickerStore` interface that offers 4 methods: `StoreAsync, RenewAsync, RetrieveAsync, RemoveAsync`.

- The `StoreAsync` method is executed immediately after authentication on the authentication server, it saves the user session to the database.
- The `RenewAsync` method in our case is used by the background service to update user sessions.
- The `RetrieveAsync` method is executed every time a request is sent to the endpoint marked with the `[Authorize]` attribute.
- The `RemoveAsync` method is executed when the browser cookie has expired, as well as is used by the same `RefreshBackgroundService` to remove sessions which have not been used for a long time.

Example `TicketStore` implementation can be found at [10]. Example of `TicketStore` dependency injection can be found at [12]. Authentication cookies are being setup at this step.

(6) Step 1 is repeated here, but now the HTTP request is for sure to be with `200 (OK)` status code.

(7) Precompiled Angular frontend application now sends request to the another microservice with authentication cookies attached to the request's `Bearer` header using `YARP` library [13, 14], so that microservice is accessible. The `YARP` is configured according to [14]

(8) If previous step returns `401 (Unauthorized)` status code, then Step 1 is repeated

5. Refresh token flow

6. Conclusions

In this manuscript we explore the problem of secure storage and transfer of access tokens between microservices. Particular attention was paid to possible vulnerabilities during

transfer of access tokens such as Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF).

To eliminate these vulnerabilities, it is necessary to store authorization tokens in cookies with mandatory `HttpOnly` and `SameSite` settings such that `SameSite` values should be `Lax` or `Strict`. Therefore, cookies are either transmitted via secure `HTTP` methods or not transmitted at all.

Authentication to be implemented using the OIDC protocol [5, 6] and Authorization code flow with PKCE [8]. The main principle of the OIDC protocol is described more detailed in Chapter 2.

Also, we provide an authentication / authorization implementation based on the ASP.NET Core Web API backend and Angular frontend application. These applications are stored under the single domain to eliminate necessity to transfer authorization cookies cross domain way. Transfer of access tokens between microservices is implemented using Reverse Proxy YARP [13] so that the access token is automatically substituted in the request header.

In addition, we proposed a mechanism to refresh an access token through the `TicketStore` [9] entity and `HostedService` class [15]. Therefore, the `TicketStore` checks each request for access token expiration. In case of expiration of the access token, the access token is refreshed by means of authorization microservice. The `TicketStore` also stores pairs of the access and refresh token inside `AuthenticationTicket` entity [11].

Finally, in this manuscript we proposed a solution to the problem of securely storing an access token and passing it between microservices, eliminating the Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) vulnerabilities.

7. Acknowledgements

Thanks someone and somebody for help and useful comments.

References

[1] Kevin Spett. Cross-site scripting. *SPI Labs*, 1(1):20, 2005.

[2] Mohd Shadab Siddiqui and Deepanker Verma. Cross site request forgery: A common web application weakness. In *2011 IEEE 3rd International Conference on Communication Software and Networks*, pages 538–543. IEEE, 2011.

[3] Roy Fielding and Julian Reschke. RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): semantics and content. https://www.rfc-editor.org/rfc/rfc7231, 2014.

[4] OWASP Cheat Sheet Series. Cross-Site Request Forgery Prevention Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html, 2023.

[5] Prabath Siriwardena and Prabath Siriwardena. OpenID Connect (OIDC). https://link.springer.com/chapter/10.1007/978-1-4842-2050-4_6, 2020.

[6] Nat Sakimura, John Bradley, Mike Jones, and E Jay. OpenID connect dynamic client registration 1.0. https://openid.net/specs/openid-connect-registration-1_0-final.html, 2014.

[7] Dick Hardt. The OAuth 2.0 authorization framework. https://www.rfc-editor.org/rfc/rfc6749, 2012.

[8] J Bradley and N Agarwal. RFC 7636: Proof Key for Code Exchange by OAuth Public Clients. https://www.rfc-editor.org/rfc/rfc7636, 2015.

[9] Microsoft. ITicketStore Interface. https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.authentication.cookies.iticketstore?view=aspnetcore-7.0, 2023.

[10] Dmitrij Kudryashov. Ticket store implementation. https://gist.github.com/Ketteiteki/7eff8e3bb35d8d2877bd74404dcef129, 2023.

[11] Microsoft. AuthenticationTicket Class. https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.authentication.authenticationticket?view=aspnetcore-7.0, 2023.

[12] Dmitrij Kudryashov. Ticket store dependency injection. https://gist.github.com/Ketteiteki/3046261b345955f6e3d4e164774bfed1, 2023.

[13] Microsoft. YARP: Yet Another Reverse Proxy. https://microsoft.github.io/reverse-proxy, 2021.

[14] Dmitrij Kudryashov. Yarp section app settings. https://gist.github.com/Ketteiteki/45d76a8409b3bb2fd9030fc8f117ca38, 2023.

[15] Microsoft. Background tasks with hosted services in ASP.NET Core. https://learn.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-7.0&tabs=visual-studio, 2023.

Email address: kolosovp94@gmail.com

URL: https://kolosovpetro.github.io