2 COLOR STATE SEAL (blk

Version 1.0

February 11, 2020

by: Dina Heidar

State of Louisiana



IdentityServer4 Technical Design Document

TABLE OF CONTENTS

[Revision History 3](#_Toc32417496)

[Introduction 4](#_Toc32417497)

[Purpose 4](#_Toc32417498)

[Audience 4](#_Toc32417499)

[Acronyms, Abbreviations, Terms and Definitions 4](#_Toc32417500)

[Design Overview 5](#_Toc32417501)

[Approach 5](#_Toc32417502)

[Security Requirements 5](#_Toc32417503)

[Architectural Goals and Constraints 5](#_Toc32417504)

[Technologies Used 5](#_Toc32417505)

[OpenID Connect and OAuth 2.0 Protocols Flows 6](#_Toc32417506)

[What is IdentityServer? 7](#_Toc32417507)

[Common IdentityServer Terminologies 7](#_Toc32417508)

[User 7](#_Toc32417509)

[Client 7](#_Toc32417510)

[Token 7](#_Toc32417511)

[Grant Types (FLOWS) 8](#_Toc32417512)

[Scopes 8](#_Toc32417513)

[Which OpenID Connect/OAuth 2.0 Flow is the right one? 8](#_Toc32417514)

[Response Types 9](#_Toc32417515)

[Un-authenticated user using Authorization Code Flow and Proof Key for Code Exchnage (PKCE) 10](#_Toc32417516)

[UN-AUTHENTICATED User USING HyBrid Flow and EncryPTION 13](#_Toc32417517)

[Topology Diagram 14](#_Toc32417518)

[Features 15](#_Toc32417519)

[Appendix A 16](#_Toc32417520)

[References 18](#_Toc32417521)

**TABLE OF FIGURES**

[Figure 1: OpenId Connect and OAuth 6](#_Toc32414968)

[Figure 2: Authorization Code Flow and PKCE 11](file:///C:\Users\dinah\source\repos\IdentityServer\documents\IdentityServer4%20Technical%20Design%20Document.docx#_Toc32414969)

[Figure 3: Existing User 13](file:///C:\Users\dinah\source\repos\IdentityServer\documents\IdentityServer4%20Technical%20Design%20Document.docx#_Toc32414970)

[Figure 4: System Integration 14](#_Toc32414971)

## Revision History

| Version | Date | Description | Editor |
| --- | --- | --- | --- |
| **1.0** | 2/11/2020 | Initial Draft | Dina Heidar |

## Introduction

### Purpose

The purpose of this project is show how IdentityServer4 can:

1. Authenticate users via the clients to external non oidc/oauth IdentityProviders (e.g. ADFS, CA IAM…. etc.) obtain in return an *access\_token* to communicate with apis(e.g. SAML token to Bearer)
2. Eliminate the need for clients to create jwts for apis thus reducing responsibilities and update certificate maintenance issues when they expire (for both the clients and apis).
3. Create and verify jwts between clients and apis.
4. Obtain *access\_tokens* and *id\_tokens* without using *client\_secrets*.
5. Obtain machine to machine *access\_tokens* that contain the necessary client claims when actions are performed without users.
6. Have *access\_tokens* contain the necessary user claims when api actions are performed on behalf of a user.
7. Ensure that *id\_tokens* are either encrypted (JWE) or not exposed over the browser or both.
8. Add/remove/verify roles, claims and locations for users. (using identity and possibly AdminUI)
9. Utilize customextension/ delegation grants. <https://www.scottbrady91.com/OAuth/Delegation-Patterns-for-OAuth-20>

### Audience

The intended audience for this document is State of Louisiana agency business technical architects and developers.

### Acronyms, Abbreviations, Terms and Definitions

Please refer to Appendix A for the list of all acronyms and abbreviations.

## Design Overview

### Approach

### Security Requirements

### Architectural Goals and Constraints

## Technologies Used

The target platform can be either Windows or Linux. The framework used is .NET Core 3.1 and the development environment is Microsoft Visual Studio.

## OpenID Connect and OAuth 2.0 Protocols Flows

OAuth (1.0/2.0) is a security standard where you sign in to an application using an existing account from another service and give that application permission/authorization to access your data or use features in another application on your behalf without giving it your password. OAuth has a number of “flows” (ways to log in and communicate with the app asking for permissions)

OpenID is another standard protocol for authenticating to external apps from a single account but focused only on authentication. This is the key difference with OAuth, where it was concerned on authorization, but it lacked the ability to authenticate users.

The OpenID Connect flow is an extension over OAuth with adds an additional layer to provide a mechanism to authenticate. It merges the latest OAuth and OpenID into a single system. It is basically a thin layer that sits on top of OAuth that adds login and profile information about the person who is logged in. ... The OpenID Connect flow looks the same as OAuth.

More info at <https://developer.okta.com/blog/2019/10/21/illustrated-guide-to-oauth-and-oidc>

**HTTP**

**OAuth 2.0**

**OpenID Connect**

Authorization

Authentication

Figure : OpenId Connect and OAuth

## What is IdentityServer?

IdentityServer is an OpenID Connect provider - it implements the OpenID Connect and OAuth 2.0 protocols. The protocol flows are defined by setting a grant type(s) to clients. These grant types specify how a client can interact with the token service.

IdentityServer has a number of jobs and features - including:

* protect your resources
* authenticate users using a local account store or via an external identity provider
* provide session management and single sign-on
* manage and authenticate clients
* issue identity and access tokens to clients
* validate tokens

## Common IdentityServer Terminologies

### User

A human that is using a registered client to access a resource

### Client

A client is a piece of software requesting a token from IdentityServer e.g. Mobile app, web app, SPAs etc. They request for tokens

### Token

There are different types of tokens. Each token type has its own use. These token are requested from IdentityServer by the client to use to access an API and/or to identity the user.

* 1. **Identity Token** *(id\_token)* - for authenticating the user
     + identity token is the outcome of the authentication process
     + contains an identifier for the user (a sub/subject claim)
     + also contains info on how and when the user authenticated, but it can contain additional identity data which may represent user claims and more.
     + many OIDC implementers will use JSON Web Token as the format for this token but this is not part of the OIDC spec. JWT is also commonly used for access\_token and refresh\_token.
  2. **Access Token** *(access\_token)* - for accessing an API resource (authorize access to data). These are used as bearer token which means that the bearer can access the authorized resource without further authentication. They generally have a short lifespan.
     + clients request access tokens and forward them to APIs
     + contain information about the client and (optionally) the user which can be used to figure out how much they can do. This is how the APIs authorize.
  3. **Refresh Token** (*refresh\_token*) – these are used to obtain a new access\_token. These are typically long lived.
  4. IdentityServer wants to know which clients can request tokens - it’s a whitelist.

### Grant Types (FLOWS)

How a client wants to interact with IdentityServer to retrieve an *access\_token*. OpenID Connect and OAuth 2 defines a number of grant types:

1. ~~Implicit~~ - will not be covered here
2. Authorization Code
3. Hybrid
4. ~~Resource Owner Password~~ - will not be covered here since it not recommended
5. Client Credentials

### Scopes

The access privileges that are being requested by the Client. There are a number of built-in ones including:

1. **OpenID** – Since OpenID is an extension on OAuth this includes *sub, name, given\_name, family\_name, middle\_name, nickname, preferred\_username, profile, picture, website, email, email\_verified, gender, birthdate, zoneinfo, locale, phone\_number, phone\_number\_verified, address, updated\_a*t.. The magic is here, by using this scope, we extend OAuth to support Authentication
2. **Standard OAuth scopes** which are: *profile, email, address* etc. - your standard OAuth scopes, what should the client have access to?
3. **Custom scopes** that you can define.

## Which OpenID Connect/OAuth 2.0 Flow is the right one?

**Machine to Machine Communication**: Client Credentials Flow

**Browser-based Applications (SPA)**: OAuth 2.0 working group now recommends Authorization Code with PKCE vs previously used Implicit.

**Native Applications**:  Authorization Code with PKCE (with no secret since it is useless) 1 OR Hybrid with encrypted JWT 2 .

## Response Types

The different response types along with scopes tell Identity Server what information (types of tokens) to send to the client.3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Flow** | **Response Types** | **Scope** | **Authorization Code** | **Access Token** | **ID Token** |
| Authorization Code | code | openid | **🗸** | **🗸** | **🗸** |
|  | **🗸** | **🗸** |  |
|  | token | openid |  | **🗸** |  |
|  |  | **🗸** |  |
| Implicit | id\_token |  |  |  | **🗸** |
| Implicit | id\_token token |  |  | **🗸** | **🗸** |
| Hybrid | code id\_token |  | **🗸** | **🗸** | **🗸** |
| Hybrid | code token | openid | **🗸** | **🗸** | **🗸** |
|  | **🗸** | **🗸** |  |
| Hybrid | code id\_token token |  | **🗸** | **🗸** | **🗸** |

### Un-authenticated user using Authorization Code Flow and Proof Key for Code Exchnage (PKCE)

During authentication, mobile/native applications can use the OAuth 2.0 Authorization Code Flow, but they require additional security because they:

Cannot securely store a Client Secret. Decompiling the app will reveal the Client Secret. The Client Secret is bound to the app and is the same for all users and devices.

May make use of a custom URL scheme to capture redirects (e.g., MyApp://) potentially allowing malicious applications to receive an Authorization Code from your Authorization Server.

To mitigate this, OAuth 2.0 provides a version of the Authorization Code Flow which makes use of a Proof Key for Code Exchange (PKCE) (defined in [OAuth 2.0 RFC 7636](https://tools.ietf.org/html/rfc7636)).

The PKCE-enhanced Authorization Code Flow introduces a secret created by the calling application that can be verified by the authorization server; this secret is called the Code Verifier. Additionally, the calling app creates a transform value of the Code Verifier called the Code Challenge and sends this value over HTTPS to retrieve an Authorization Code. This way, a malicious attacker can only intercept the Authorization Code, and they cannot exchange it for a token without the Code Verifier

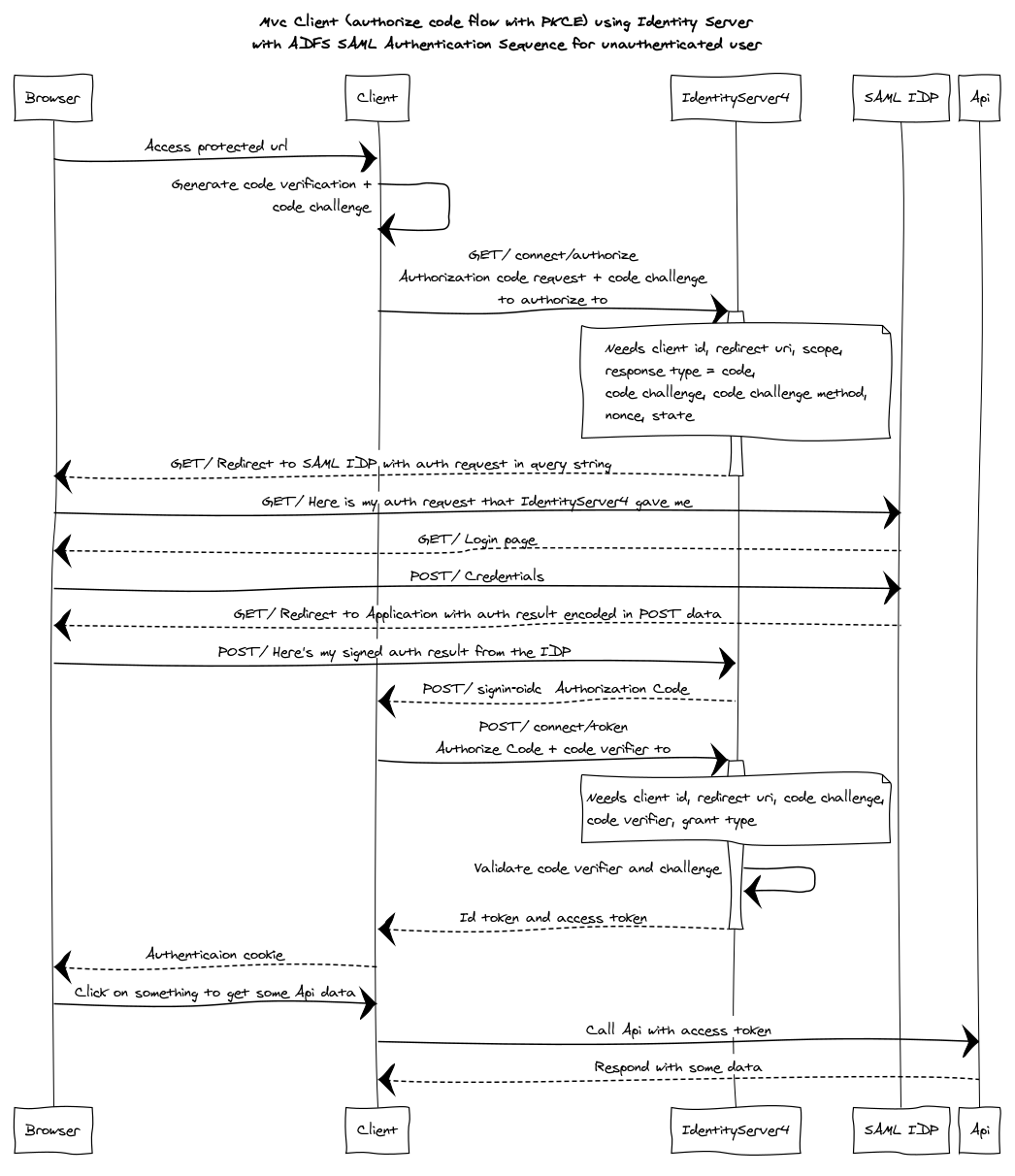


Figure 2: Authorization Code Flow and PKCE

### UN-AUTHENTICATED User USING HyBrid Flow and EncryPTION

Figure 2: Hybrid Flow and Encryption

Figure 3: Existing User

## Topology Diagram

Figure 4: System Integration

## Features

## Appendix A

C

CEK: Content Encryption Key, 7

E

EA: Enterprise Architecture, 6

ESB: Enterprise Service Bus, 6

I

IAM: Identity Access Management, 6

IDP: Identity Provider, 6

J

JWE: JSON Web Encryption, 6

JWS: JSON Web Signature, 6

JWT: JSON Web Token, 6

O

OTS: Office of Technology Services, 7

S

SAML: Security Assertion Markup Language, 6

SOAP: Simple Object Access Protocol, 6

SSO: Single Sign-On, 6

*This page intentionally left blank*

## References

1. Brady S. ASP.NET Core using Proof Key for Code Exchange (PKCE). <https://www.scottbrady91.com/OpenID-Connect/ASPNET-Core-using-Proof-Key-for-Code-Exchange-PKCE>. Published 2019. Accessed2020.

2. Brady S. Encrypting Identity Tokens in IdentityServer4. <https://www.scottbrady91.com/Identity-Server/Encrypting-Identity-Tokens-in-IdentityServer4>. Published 2019. Accessed 2020.

3. Kawasaki T. Diagrams of All The OpenID Connect Flows. <https://medium.com/@darutk/diagrams-of-all-the-openid-connect-flows-6968e3990660>. Published 2017. Accessed 2/12/2020, 2020.