OpenID Connect & OAuth2 Protocols

The Right and Left Hand of Software Security

G. James McKenna

8/2/2021

Security is an essential component for enterprise software and there is more to software security than preventing SQL Injections and Cross Site Scripting. In fact, it could be argued that SQL Injections and Cross Site Scripting are lesser security concerns if any User using the software can view, copy, alter or delete data - or worse yet view, copy, alter or delete confidential data. This line of thought leads to a broader and more important part of software security: Authentication and Authorization. In this paper, I will take a look at the protocols used to authenticate a client application against an API, authenticate the User using a client application, and discuss how authorization is a natural extension of authentication.

Contents

Authentication and Authorization are the right and left hand of software security, and in today's world, almost every application reaches out to a backend of sorts. With experience, software developers have come to the understanding that there is a need to authenticate both the client application and the End User. Over the years, authentication and authorization has continued to evolve as the industry came to understand of the shortcomings of existing auth and auth solutions. Many developers and security experts have debated what is needed, how to implement security in a better way, and the best practices that should be followed. On April 2020, Microsoft announced that they will stop supporting, and retire, Basic Authentication for Exchange Active Sync, the Post Office Protocol, Internet Message Access Protocol and Remote PowerShell in Exchange Online security practices - even going so far as to block applications that use these technologies.(The_Exchange_Team, 2020)

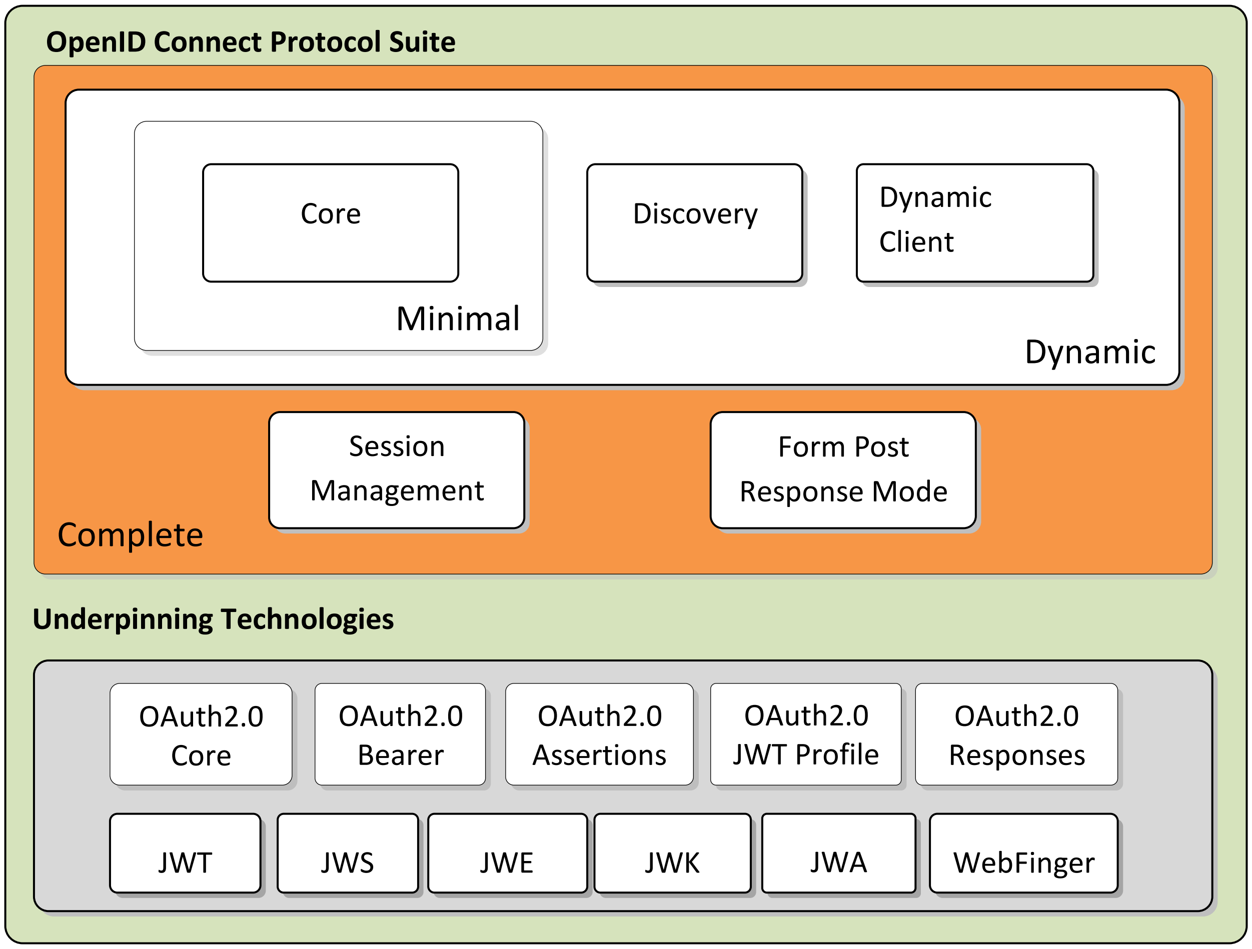
In this paper, I plan to give an overview of the OAuth2 protocol: what it is and how it relates to the OpenID Connect protocol. I will illustrate how they work together and why the two protocols have become the standard (and will be for the foreseeable future). I will introduce the reader to an open source framework that implements the two protocols, and talk about some of the more important features. I will touch on how the protocols can be implemented to provide a Software as a Service (SaaS) service, and list a few major technology companies that have already implemented the two protocols.

The Protocols

OAuth 2.0 Authorization Framework

"*The OAuth 2.0 authorization framework enables a third-part 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*."(Hardt, D Ed. , 2012)

OpenID Connect



Protocol Flows

Implicit Flow

Using the Implicit Flow, all tokens are returned from the Authorization Endpoint and the Token Endpoint is not used. The Implicit Flow is sometimes used by Clients that are implemented using a browser application. The Access Token and ID Token are returned directly to the Client, which may expose them to the User and other applications that have access to the browser. The Authorization Server does not perform Client Authentication. This is the least secure way of implementing the OpenID Connect protocol and can leave the Access Token and ID Token open to Cross Site Scripting and Cross Site Request Forgery attacks. Newer browsers that have implemented such standards as HTTPOnly and Same-Site-Mode offer some level of security, though the developers have to implement the functionality properly.

Hybrid Flow

When using the Hybrid Flow, some tokens are returned from the Authorization Endpoint and others are returned from the Token Endpoint. The way of returning tokens in the Hybrid Flow are laid out in the **OAuth 2.0 Multiple Response Type Encoding Practices**, specification. When making the Authorization request, certain parameters must be included in the query string, and it is the request parameter "response_type" that informs the Authorization Server of the processing flow.

Authorization Code Flow

The Authorization Code Flow is the most secure when using short token lifetimes, and all tokens are returned from the Token Endpoint. The Authorization Code Flow, returns an Authorization Code to the Client Application, which can then exchange it for an ID Token and an Access Token. By returning a code to the browser, no tokens are exposed to the browser or other malicious applications (running in another tab) with access to the browser. The Authorization Server will also authenticate the Client Application before exchanging the Authorization Code for an Access Token. The Authorization Code flow is best for Client Applications that can securely communicate a Client Secret between themselves and the Authorization Server - typically, these types of Clients are server-side rendered, and are able to hold the Client Secret in server-side state.

The Problem OpenID Connect Solves

Securing the Client

OAuth 2.0 has become the standard way to authenticate an application against an API. It is how an API knows whether to process a request for confidential data from a Client Application, or reject the request. The Client will pass a Bearer Token - in the form of a Json Web Token - to an API in the http Authorization Header. In the past, the OAuth 2.0 Access Token was also used to hold and pass User information and access rights to protected resources. This was a breach of the OAuth 2.0 specification but solved a problem that IT security experts couldn’t identify until developers used OAuth 2.0 Access Tokens in such a way.

Authenticating a User

Before the adoption of OpenID Connect, OAuth 2.0 tokens were used in such a way that made it difficult to standardize authentication and authorization across applications within the same domain, or across domains. With the adoption of OpenID Connect, applications and APIs can separate the User from the Client Application. Now, the same application can make a request to an API (protected resource) and the API can determine whether to process the response based on the end user using the client application. For example, a doctor and a nurse using the same application can look up a patient’s medical records. With the OpenID Connect ID Token, an API can differentiate between the requests from the doctor and the nurse, allow the doctor to write to and edit the patient's records, while the nurse may only have read access.

Securing Services

The OAuth 2.0 protocol can be used to authenticate one application calling to another application. A developer can write an application that calls out to APIs. Some APIs are public and open; other APIs are secure and protect both data resources and computing resources. OAuth 2.0 is one way for a service protecting resources to know if the calling application has rights to consume the response data, or computational resources of the service.

Beyond the Protocols

The Big Picture

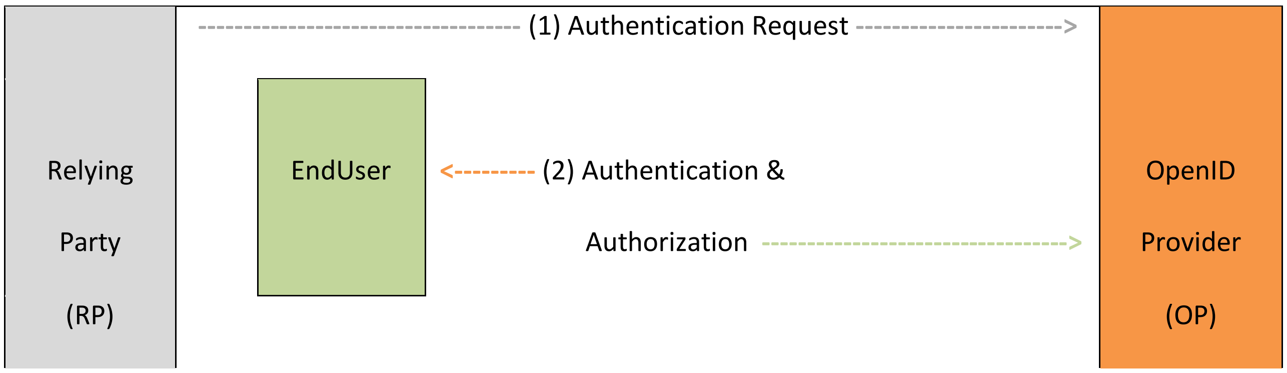
1. The RP (Relying Party/Client) sends a request to the OpenID Provider (OP).

2. The OP authenticates the End-User and obtains authorization.

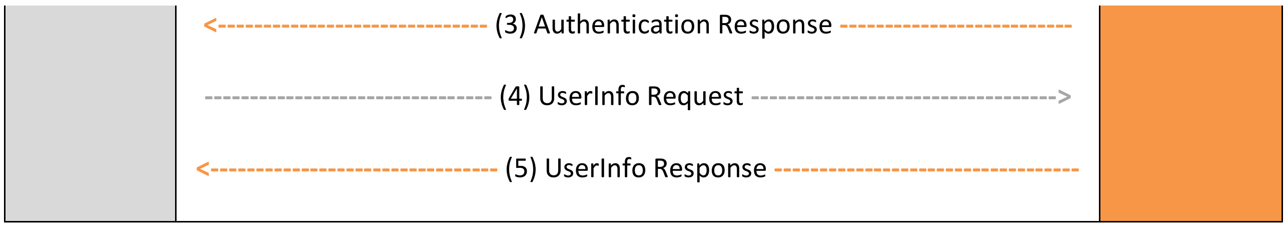
3. The OP responds with an ID Token and usually an Access Token.

4. The RP can send a request with the Access Token to the UserInfo Endpoint.

5. The UserInfo Endpoint returns Claims about the End-User.



(1) Authentication Request
Relying Party (RP)
EndUser
(2) Authentication & Authorization
OpenID Provider (OP)



Terminology

When one first begins their journey of setting up an OpenID Connect implementation, one of the most daunting aspects is all of the terminology: terminology that spans different protocols and disciplines of software security and Information Technology. For this document, I want to introduce some of the necessary terminology for a basic understanding of the OpenID Connect Protocol.

User

A User is the human that is using a client application that is registered with an OAuth 2.0 Authorization Server. The client application requests a token to access resources on behalf of and for the User.

Client

A client is a piece of software that requests an Access Token, or Identity Token, from an OpenID Provider and that same client application must be registered with that OpenID Provider it is requesting a token from. A client application could be a web application, native mobile apps, desktop applications, single page applications or even a server process.

Resources

Resources are something that is to be protected – typically, the identity and data of a User and backend APIs or services. Every resource will have a unique name and the client uses this name to specify which resource they want access to. A User's identity data can contain contact information such as email and phone number, physical address and location, date of birth and even organizational or career titles. API resources are functionality that a client wants to consume or use, and is typically a web API, but not necessarily.

Relying Party

A Relying Party (RP) is an OAuth 2.0 client application requiring End-User Authentication and Claims from an OpenID Provider, and in the OpenID Connect world, Relying Party and Client are sometimes used interchangeably. Please note: some "*client applications*" could be a resource, hence something you would want to protect. In this case, the client application would not be requesting tokens from the OpenID Connect Authorization Server (OP), but verifying access tokens against the OpenID Provider. An example of this would be a client-side application that manages User Accounts. For more details on this type of implementation, see in this document: *Identity Server 4, the Protocols in Practice, API Resource*.

OpenID Provider

An OpenID Provider (OP) is an Authorization Server that builds on top of the OAuth 2.0 Protocol. An OAuth 2.0 Authorization Server provides access tokens, with the OpenID Connect Protocol layering a User's identity data on top. An OP will build, distribute, maintain, validate and invalidate tokens.

Tokens

When implementing an OpenID Connect, Authentication and Authorization Server, there are three types of tokens we have to concern ourselves with: Access Tokens, Identity Tokens and - for the convenience of the user - Refresh Tokens. These tokens are a data structure (JSON Web Tokens are a popular implementation) containing information a resource server needs to verify the right of a client or user, to access protected data and services, and for a client application, or API to have a needed information regarding the user.

Access Tokens

The OAuth 2.0 Protocol defines the Access Token as a string representing authorization issued to a client application from an Authorization Server, and is usually opaque to the client. The tokens are a representation of a specific access scope and the duration of access to a protected resource. The scope is a key/value pair with the value being a list of space delimited string values, with each string value being an authorization scope.

A client application can request a scope from the authentication server, but the Authentication Server may not include the requested scope in the Access Token. The scopes included in an Access Token are dependent on the API requirements, and have been negotiated with the Authentication Server prior to a client requesting authorization. Over and above scopes, Access Tokens also contain information about the client application that intends to make use of the token, the duration of the token and the Authorization Server that granted the token.

The OAuth 2.0 protocol also defines an Introspection Endpoint specification that an Authorization Server adheres to, and that an API is able to differentiated between two types of Access Token. An access token may be an identifier that a resource server can use to retrieve the authorization information from the Authorization Server – known as a Reference Token - or the Access Token may itself be the authorization information in the form of a JSON Web Token (JWT) - known as a Bearer Token - that a client received from the Authorization Server.

ID Token

The main contribution to authentication security that OpenID Connect provides is to build on top the OAuth 2.0 specification. This layering of the protocols enables both client applications and users to be authenticated. The ID Token data structure takes the form of a JSON Web Token (JWT) and is a token that contains Claims about a user. There are many standard OpenID Connect user claims, and custom claims can be defined at the Authorization Server, then requested by a client during the login process.

Once the client has received the ID Token from the Authorization Server, the client application can then customize the user experience in many ways such as theme preference, displaying a user chosen avatar and even allow for authorization by the client rendering links to otherwise protected endpoints. So generally, the access token authorizes a client to make a request to an API, and the ID Token contains information about what endpoints from that API a user is authorized to use.

An example might be a Sales Tracking Application and an API that saves data to persistent storage. Once authenticated by the Authorization Server, the client app receives both an Access Token and ID Token for the logged in user. The Access Token contains a "scope: SalesAPI" property, and the ID Token contains a key/value claim of *"Position": "Junior Sales"*. The client application knows that the user has been authorized to use the *View Customer Purchases,* navigation link, and so renders it for user. The user clicks the link and the client app makes a request to the API for a customer's previous purchases. The API validates the access token and sees that the user has a claim of *"Junior Sales"*, so completes the request by making a call to persistent storage, then returning the data to the client app. The sales person sees that one of the past purchases is incorrect, but the isn't able to edit the entry because the ID Token doesn't contain a claim *"Position: Manager Sales"*. Due to the ID Token not containing the claim *"Position: Manager Sales"*, the client app doesn't render the *Edit Purchase,* navigation link.

Typically, an ID Token will contain claims with personal user information such as name, address, and email. Though as previously stated, custom claims can be defined for such use cases as in the previously stated example - custom claims will usually reflect to a business rule. Ideally, a well-architected API will have authorization privileges over and above simply validating that the request came from an allowed client application, the API will also validate the user's privilege level on an endpoint-by-endpoint basis using claims from the ID Token.

Refresh Token

Depending on the level of security required, a Refresh Token may be functionality to include when configuring an Authorization Server. When looking at the OpenID Connect specs, there is a definition for offline_access. *"OPTIONAL. This scope value requests that an OAuth 2.0 Refresh Token be issued that can be used to obtain an Access Token that grants access to End-User's UserInfo Endpoint even when the End-User is not present (not logged in)."* (Multiple Authors, 2014) At this point, I am not certain why one would design a secure application but still allow a user to access protected resources while not logged in, but the functionality can be configured.

The OpenID Connect Core spec goes on to say, *"The use of Refresh Tokens is not exclusive to the offline_access use case. The Authorization Server MAY grant Refresh Tokens in other contexts that are beyond the scope of this specification."* There is a use case for Refresh Tokens that comes to mind: Inactive Timeout. Implementing a relatively short duration for Access Tokens is a wise security decision, (token expires before a hacker can use it), although a short duration token can be annoying for the user when it comes to inactivity. The Authorization Server can be configured to automatically logout a user when the token expires, but a well designed app can warn a user – with let's say with a Pop Up - that a token is about to expire and provide a choice to stay logged in. If the user is active enough to click the, *stay logged in button*, the app can make a request to the Authorization Server with a refresh token and request a new Access Token to prevent the automatic logout on the current token's expiration.

Integration and Extensibility

Single Sign On

While writing this paper I have attempted to interpret all the research done in my own words, but every so often an idea or concept cannot be expressed better - this is one of those cases. As such, I will quote from the most reputable of IT organizations: *"Single sign-on (SSO) is an identification system that allows* *users to access multiple applications and websites with one set of login credentials. The implementation of SSO within an enterprise helps to make the overall password management easier and improves security as workers access applications that are on-premises as well as in the cloud."*(CISCO)

Multi-Tenant

Once Single Sign-on is implemented an OpenID Connect Authorization and Authentication Server, in essence, becomes a Multi-Tenant Application in the sense that it can authorize some Client Applications to access some resource servers (APIs), and Users to access some resources (data or services). Each Client Application can request different Scopes (see Resources Section below) for the authenticated User. A Multi-Tenant, OpenID Connect Server can be implemented within an organization's IT infrastructure or it can provide identity as a service and can be structured as a Software as a Service (SaaS) business model like in the case of Okta.com and AuthO.com

External Login

When an OpenID Connect server implementation is a SaaS and/or a Multi-Tenant Application, it can be configured as an External Login Provider. A blogging application for example, can off-load the responsibility of Identity Management by setting up and configuring an External Identity Provider such as Microsoft Identity, Google Account or Facebook - to name a few. If a blogger wants to allow comments on blog posts, it makes sense for the blogging application to use an external login provider. In some cases, using an external identity provider such as Twitter or Facebook, can drive traffic and interactions to a blogging site as readers will be more apt to post comments when logging in is a simple as clicking the "*Sign-In with Facebook*", button. If the user is already signed into Facebook - has tokens from Facebook on the device - then Single Sign-On is implemented. The tokens are refreshed and new or different scopes are granted allowing the User to comment on a blog post; the User doesn't have to register with the blogging site and then log in. This makes for a good user experience (UX) and can help drive traffic to smaller online communities that do not have the resources to implement their own User Identity Management systems.

IdentityServer4, the Protocols in Practice

IdentityServer4 is an open-sourced, middleware framework that implements, is certified for, and spec compliant with the OpenID Connect protocol. It is to be used with an arbitrary ASP.NET Core application to solve the typical security problems of today's mobile, native and web applications. It works hand and glove with ASP.NET Core Identity and Entity Framework Core to provide the whole Authentication and Authorization solution that today's applications require. IdentityServer4 supports Single Sign-On, Access Control for APIs, Federation Gateways and is highly customizable to fit the needs of your use case. IdentityServer4 is a server-side framework that has complementary libraries for server-side rendered, client application such as .NET MVC and Razor Pages, native mobile and desktop apps, as well as a JavaScript library for modern SPA applications.

As of October 1 2020, the IdentityServer4 development team announced that they started a new company specializing in Auth and Auth solutions for the .NET environment. On the IdentityServer4 Documents website, the development team states, "IdentityServer4 will be maintained with bug fixes and security updates until November 2022". Considering IdentityServer4 uses the Apache 2 license and is part of the .NET Foundation, I suspect that IdentityServer4 will be around past November 2022.

Resources

Giving an overview of OpenID Connect in this paper is an attempt to show how the OpenID Connect protocol, layered on top of the OAuth protocol, has a main purpose of securing protected resources - whether those resource are a user's identity and personal data, or an organizations confidential data. IdentityServer4 differentiates between a user using the system and a client application attempting to access functionality such as an API (Http-based endpoints), or server-based application. Below one can read about an API Resource and the Identity Resource in the context of an IdentityServer4 implementation. I will give examples similar to, but not exactly like the IdentityServer4 documents to relate this paper to another learning resource in attempt to make clear the topics discussed.

API Resource and API Scope Classes

The API Resource Class in IdentityServer4 is how a protected resource is registered with the Authorization Server implementing the IdentityServer4 Framework. To begin, we start by defining scopes. As stated above, it is the scopes contained in an Access Token that an API validates to let a request through to a protected resource and then to process a response. When a User and/or Client Application authenticates with an IdentityServer4 Authorization Server, the Client requests scopes be added to the returned Access Token. If the scopes have been registered with the Authorization Server, the Client will be allowed access protected resources in the form of http endpoints, and the API will process the request and serve the response. Below there are code examples of registering ApiScopes with Identity Server4, how protected API and Identity Resources are defined within Identity Server4 and an example Client registration that can request access at authentication time. These scopes will be included in Access and Id Tokens in the form of claims (key value pairs in a JWT).

```
public static IEnumerable<ApiScope> GetApiScopes()
{
   return new List<ApiScope>
   {
      // invoice API specific scopes
      new ApiScope(name: "invoice.read",   displayName: "Reads your invoices."),

      new ApiScope(name: "invoice.pay",    displayName: "Pays your invoices."),

      // customer API specific scopes
      new ApiScope(name: "customer.read",
                          displayName: "Reads you customers information."),

      new ApiScope(name: "customer.contact",
                          displayName: "Allows contacting one of your customers."),

      // shared scope
      new ApiScope(name: "manage",
                          displayName: "Provides administrative access to invoice and customer data.")
   };
}
```

ApiResource class with two logical APIs created and registered with their respective scopes:

```
public static readonly IEnumerable<ApiResource> GetApiResources()
{
   return new List<ApiResource>
   {
      // A registered API resource
      new ApiResource("invoice", "Invoice API")
      {
         Scopes = { "invoice.read", "invoice.pay", "manage" }
      },
     // A registered API resource
      new ApiResource("customer", "Customer API")
      {
         Scopes = { "customer.read", "customer.contact", "manage" }
      }
   };
}
```

Identity Resource

Only one Identity Resource is mandatory to be OpenID Connect compliant, and that is the “*openid”* scope. This identity resource tells the identity provider to return the *sub* (subject id) claim in the identity token. An Identity Resource is a group of claims that can be requested by a Client during the authentication workflow. The OpenID Connect specification suggests a couple of standard scope/claim names that might be useful, though other than the openid scope, others can be freely defined at design time by the development team.

OpenID scope defined and registered in code:

```
public static IEnumerable<IdentityResource> GetIdentityResources()
{
   return new List<IdentityResource>
   {
      // Required to be OpenID compliant
      new IdentityResource(name: "openid",
                                     userClaims: new[] { "sub" },
                                     displayName: "Your user identifier"),
      // Defining custom claims
      new IdentityResource(name: "profile",
                                     userClaims: new[] { "name", "email", "website" },
                                     displayName: "Your profile data")
   };
}
```

The Client Class

Clients represent applications that can request tokens from the Identity Server4 Authorization Server.

The details vary, but you typically define the following common settings for a client:

- a unique client ID
- a secret if needed (for certain Flows, Grant Types, Work Flows)

- the allowed interactions with the token service (called a grant type, flow or work flow interchangeably)
- a network location where identity and/or access token gets sent to (called a redirect URI, and out of the scope of this paper)
- a list of scopes (aka resources) the client is allowed to request access to

```
public class Clients
{
   public static IEnumerable<Client> Get()
   {
      return new List<Client>
      {
         new Client
         {
            ClientId = "interactive",
            ClientSecrets = { new Secret("secret".Sha256()) },

            // Grant Type, Flow, Work Flow
            AllowedGrantTypes = GrantTypes.Code,
            AllowOfflineAccess = true,

             // Essential to Client registration, not covered in this paper
            RedirectUris =          { "http://localhost:21402/signin-oidc" },
            PostLogoutRedirectUris = { "http://localhost:21402/" },
            FrontChannelLogoutUri =    "http://localhost:21402/signout-oidc",

            AllowedScopes =
            {
             //Identity resources returned to the Client in the ID Token
             IdentityServerConstants.StandardScopes.OpenId,
             IdentityServerConstants.StandardScopes.Profile,

             //API Resource Scopes this Client can request and will be returned in the Access Token
             "invoice", "customer", ”manage”
            },
         }
      };
   }
}
```

End Points

Discovery Endpoint

The discovery endpoint is used to retrieve metadata about an Identity Server4 implementation. It returns information such as the issuer name (Identity/Authorization Provider), the supported endpoints (some listed below), the configured Grant Types/Flows/Work Flows, the API and identity scopes, and the signing algorithms used for tokens. For an Identity Server 4, OpenID Connect implementation, the discovery endpoint is "*https://exampledomain.com/.well-known/openid-configuration"*. A working example can be found at "*https://demo.identityserver.io/.well-known/openid-configuration"*.

Authorization Endpoint

The authorize endpoint is used at login time to request tokens or authorization codes for a Client Application from the Authorization Server. This process involves authentication of the end-user usually by username and password, but optionally with multi-factor authentication if enabled. Also, the Authorization Server can be configured (on a client by client basis) to optionally allow the User to consent to the adding of identity scopes to the ID Token.

Token Endpoint

The token endpoint is used to programmatically request tokens from the Authorization Server. The Token Endpoint will support the all the Grant Types configured by the registration of the different Client Applications. Typical Grant Types supported are: Authorization Code, Client Credentials, Hybrid, Implicit and Refresh Token.

UserInfo Endpoint

The UserInfo Endpoint can be used to retrieve identity information about a user. All User claims must be preconfigured and registered prior to a Client Application requesting an Identity Resource from the Authorization Server. The calling application needs to send a valid Access Token representing the user as a parameter of the request and depending on the granted scopes, the UserInfo Endpoint will return key/value mapped, User claims.

Device Authorization Endpoint

The device authorization endpoint can be used to request device and user codes. This endpoint is used to start the Device Flow Authorization process (not discussed in the scope of this paper). Typically, this Flow is used when an application interacts with nothing application without the need of a User starting the interaction. The Device Authorization Work Flow is ideal for automation services.

Introspection Endpoint

Rather than the Authorization Server responding with Access and ID Tokens to a login request, the Client Application can be configured to use Reference Tokens. Reference Tokens were not explored in this paper due to requirements of the papers length, but for lack of an explanation, they can be thought of as a more secure, but they impose a two-way communication between the API calling the Introspection Endpoint, and the Authorization Server. Reference Tokens are not self contain tokens, not Bearer Tokens, but a reference to a successful login request that is held on the Authorization Server. The API asks the Authorization Server, “Hey do you recognize this reference code?” and the Authorization Server responses with, “Yes, I know that reference code” or “No that is not a valid reference code” and the API acts accordingly.

Revocation Endpoint

The Revocation Endpoint is called when a systems administrator wants to revoke an Access Token, (Reference Tokens only) and Refresh Token. This endpoint can also be called programmatically or event of some type. However this endpoint is called, a valid token has to be supplied in the query string as a parameter (token_type_hint, access_token, refresh_token).

End Session Endpoint

The End Session Endpoint is used to trigger the single sign-out functionality. Depending on the configuration of both Client Applications and the Authorization Provider, this can immediately sign out the User on both the Client Application(s) as well as the Authorization Server. To use the End Session Endpoint, a Client Application will redirect the User's browser to the End Session URL. All Applications (Single Sign-on) that the User has logged into during the User's session, will participate in the sign-out.

Conclusion

Today's modern authentication and authorization practices have matured to a point where the subject is understood, many of the security holes have been found and sealed, and OpenId Connect will be the standard going forward. It allows for Multi-Factor Authentication, Single Sign-On across application domains and external, or third party authentication providers. As software developers and engineers, we can ensure that one service is allowed to communicate and consume from another, that the end user is allowed access to certain data or processes, and that by following the protocols security becomes agnostic of language, framework and run-time environment. I have discussed the protocols and touched on ways to configure an authentication and authorization service, as well as introducing an open source project that wraps OpenId Connect into a framework ready to adopt.

Go forth and be secure.

References

https://techcommunity.microsoft.com/t5/exchange-team-blog/basic-authentication-and-exchange-online-april-2020-update/ba-p/1275508

http://wiki.openid.net/f/openid-logo-wordmark.png

https://www.oauth.com/

https://datatracker.ietf.org/doc/html/rfc6749

https://openid.net/specs/openid-connect-core-1_0.html

https://openid.net/connect/

https://www.oauth.com/

https://oauth.net/2/

https://openid.net/specs/openid-connect-core-1_0.html

https://identityserver4.readthedocs.io/en/latest/

https://authguidance.com/2019/09/08/ui-token-management/

https://www.cisco.com/c/en/us/products/security/what-is-single-sign-on-sso.html

https://docs.microsoft.com/en-us/azure/active-directory/fundamentals/auth-oidc

Bibliography

CISCO. (n.d.). *What Is Single Sign-On.* Retrieved from CISCO.com: https://www.cisco.com/c/en/us/products/security/what-is-single-sign-on-sso.html

Hardt, D Ed. . (2012, October). *The OAuth 2.0 Authorization Framework, RFC 6749.* Retrieved from Internet Engineering Task Force (IETF) : https://datatracker.ietf.org/doc/html/rfc6749

Multiple Authors. (2014, 11 8). *Final: OpenID Connect Core.* Retrieved from Final: OpenID Connect Core: https://openid.net/specs/openid-connect-core-1_0.html#OfflineAccess

The_Exchange_Team. (2020, April). *Basic Authentication and Exchange Online – April 2020 Update.* Retrieved from TechCommunity.Micosoft.com: https://techcommunity.microsoft.com/t5/exchange-team-blog/basic-authentication-and-exchange-online-april-2020-update/ba-p/1275508

Appendix

From the beginning, AltaML had no interest in participating in choosing the report topic, having me use AltaML time to research the topic, or write the report. As this report topic was not part of the internship with AltaML, there was no one to validate or verify the report. My supervisor at AltaML mentioned that he didn’t know anything about the subject and wouldn’t be much help.

I didn’t have any problems finding information for the report. A simple web search on the topics of OpenID Connect and OAuth 2.0 returns many results. I did have to make some concessions when it came to the layout and the rigid, 5 major points, each with 3 minor points requirement stated in the syllabus. I had a difficult time shoe horning the topic into such a structure: some major points only had 2 minor points, while other major points had more than 3 minor points. Also, the 20 to 25 pages (including or not including reference list and bibliography was never stated) was only scratching the surface of this topic. To do this topic justice, it should have been written freely and not be confined to the 5 major, 3 minor point structure, and to write the report with as many pages/word count as required to fully explore the topic.

Lastly, I can see the topic of Authorization and Authentication, knowledge of both the OpenID Connect and OAuth 2.0 protocols, assisting me in my future career.