UNIVERSITY OF CRETE
Computer Science Department

Single Sign-on for Internet Services

Thesis submitted in partial fulfilment of the requirements for the degree of Computer Science

LOUKAS MERKOURIS, University of Crete, Greece

The Internet hosts various amount of web applications, services and even more users. These users, while using an application, often want to access their data that resides in another, third-party application, i.e., when a user uses an email application and wants to access his contacts that exist in another application, how will the email application eventually get his contacts? In the old-fashioned way, the user had to hand over the credentials of his third-party application account to the application he eventually wants to get his data. This was insecure for the user, the user gave access to all of his data in the 3rd party application, including his password. With that method, the application that took his credentials could store them in plain text, the database could be hacked, and to revoke access, there was only one option; to change his password. With Single Sign-on (SSO), this problem no longer exists, as users can authorize the application to access a specific set of data residing elsewhere without disclosing passwords. Today there are many protocols and frameworks to achieve SSO, for users to be able to authorize and authenticate using third-party applications. In this thesis, we examine SSO between Internet services using the OAuth 2.0 [6] and OpenID Connect[7] protocols, by implementing example web-based applications in Python using the Django[4] framework. We also extend one of the example applications to use a remote LDAP[5] server as a user directory instead of a local user database.

1 INTRODUCTION

Single Sign-on (SSO), is an authentication and authorization scheme where users can securely authorize, authenticate and gain access to multiple applications and websites by only logging in with a single username and password. In this project we use the OAuth2.0 (OAuth) and OpenID Connect (OIDC) technologies to enable SSO. With SSO we can authorize and authenticate applications and access resources in a secure and fast way. Users can be in a local or a remote database. To enable authentication with users in a remote database, we use authentication via the Lightweight Directory Access Protocol (LDAP), which is commonly deployed for accessing and maintaining data in a company/organization server.

2 DESIGN

To test SSO with OAuth and OIDC, we implemented 3 applications with Django, which is a high-level Python web framework that encourages rapid development and clean, pragmatic design. Django follows the model–template–views (MTV) architectural pattern. Django's primary goal is to ease the creation of complex, database-driven websites. The framework

emphasizes reusability and "pluggability" of components, less code, low coupling, rapid development, and the principle of don't repeat yourself. Python is used throughout, even for settings, files, and data models. Django also provides an optional administrative create, read, update and delete interface that is generated dynamically through introspection and configured via admin models.

2.1   OAuth 2.0

OAuth 2.0 [6] is an open-standard authorization protocol that provides applications the ability for "secure designated access". For example, you can tell Facebook that it's ok for YouTube to access your profile or post updates to your timeline without having to give YouTube your Facebook password. This minimizes risk in a major way: in the event, YouTube suffers a breach, your Facebook password remains safe.

Glossary of the components are, **a) Resource Owner** is the user, **b) Resource Server** is a service hosting data, **c) Client** is an application that accesses data residing the Resource Server, **d) Authorization Server** authenticates the user and grants Tokens, and **e) Access Token** is a piece of data that represents the authorization to access resources on behalf of the Resource Owner.

To enable the OAuth protocol in our project we created the **Provider**, acting both as the Resource Server and the Authorization Server, (the Resource and Authorization Servers can be separate applications, for this project they are the same), and the **Client**. For implementing the Provider we used the **Django OAuth Toolkit**[2] package, which provides all the endpoints, data, and logic needed to add OAuth 2.0 capabilities to our Django project. With the toolkit, we made our Authorization Server issue access tokens to Client applications for a certain API.

Before our Application (the Client), can use the Authorization Server for authorization or authentication, we must first register the app. Once registered, the Client will be granted access to the API, subject to approval by its users. The registration process will automatically generate a unique Client_id and Client_secret, which are used so the Client and the Authorization Server establish a working relationship. The Authorization Server generates the Client ID and Client Secret (sometimes called the App ID and App Secret) and gives them to the Client to use for all future OAuth exchanges. The Client Secret must be kept secret so that only the Client and Authorization Server know what it is. This is how the Authorization Server can verify the Client. We have to provide the rest of the information:

- **User:** The owner of the Application (e.g., a developer, or the currently logged in user).
- **Redirect URIs:** Applications must register at least one redirection endpoint before using the authorization endpoint. The Authorization Server will deliver the Access Token to the Client only if the Client specifies one of the verified redirection URIs.
- **Client type:** This value affects the security level at which some communications between the Client application and the Authorization Server are performed. For this project we use Confidential.
- **Authorization grant type:** Authorization Code.
- **Name:** This is the name of the Client application on the server, and will be displayed on the authorization request page, where users can allow/deny access to their data.

We used the Authorization Code grant type, because when authorizing an application to access OAuth protected data with this grant type, the flow is always initiated by the user and the application can prompt users to click a special link to start the process. When a user

clicks the link, the user is redirected to the Authorization Server. If the user is not logged in, he will be prompted for a username and password. This is because the authorization page is login-protected by Django-OAuth-Toolkit. After login the user should see be presented with a form to give his authorization to the client application. User will flag the checkbox, click Authorize, and will be redirected again to the client.

At this point the Authorization Server redirects the user to a special page on the Client passing in an Authorization Code and a special token that the Client will use to obtain the final Access Token. This operation is usually done automatically by the Client application during the request/response cycle. Access Tokens have a specific lifetime, given by the Authorization Server. In our case, the Access Token has a Refresh Token as well which is used for a brand new Access Token when necessary, without repeating the authorization process, as it has no expire time.
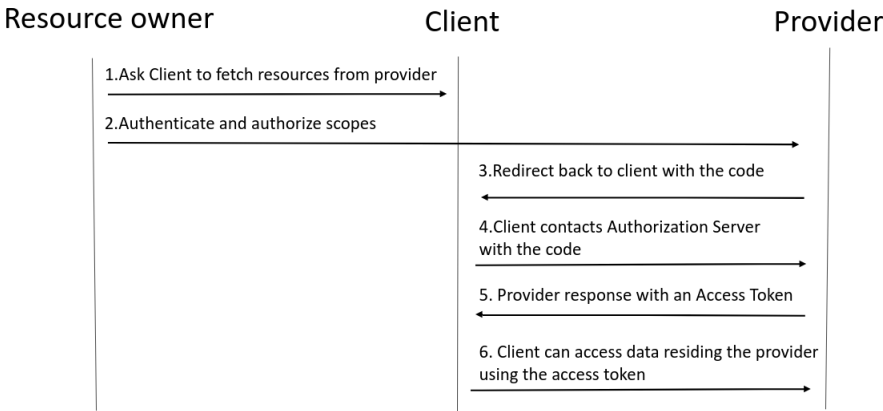

Fig. 1. OAuth flow

The OAuth2.o flow (fig. 1) follows these steps:

(1) the Resource Owner, want to allow the Client the Client, to access data residing the Provider.
(2) The Client redirects browser to the Authorization Server and includes with the request the Client ID, Redirect URI, Response Type, and one or more Scopes it needs. The Authorization Server verifies the user, and if necessary prompts for a login. he Authorization Server presents user with a Consent form based on the Scopes requested by the Client. User grant (or deny) permission.
(3) The Authorization Server redirects back to Client using the Redirect URI along with an Authorization Code.
(4) The Client contacts the Authorization Server directly (does not use the Resource Owner's browser) and securely sends its Client ID, Client Secret, and the Authorization Code.
(5) The Authorization Server verifies the data and responds with an Access Token.

The Client can now use the Access Token to send requests to the Resource Server for accessing data.

2.2 OpenID Connect

OAuth 2.0 is designed only for authorization, for granting access to data and features from one application to another. After the OAuth flow, usually, the first step for the Client is to get information about the Resource Owner (the end-user), by sending a request to the Resource Server. That step can be avoided by using OIDC. OpenID Connect (OIDC) is a thin layer that sits on top of OAuth 2.0 that adds login and profile information about the person who is logged in. Establishing a login session is often referred to as authentication, and information about the person logged in (i.e. the Resource Owner) is called identity. When an Authorization Server supports OIDC, it is sometimes called an identity provider, since it provides information about the Resource Owner back to the Client. OIDC flow looks the same as OAuth, the only differences are that in the initial request, a specific scope of **openid** is used, and the Server responds with an Access Token and A JSON Web Token (JWT) that contains the information about the Resource owner.
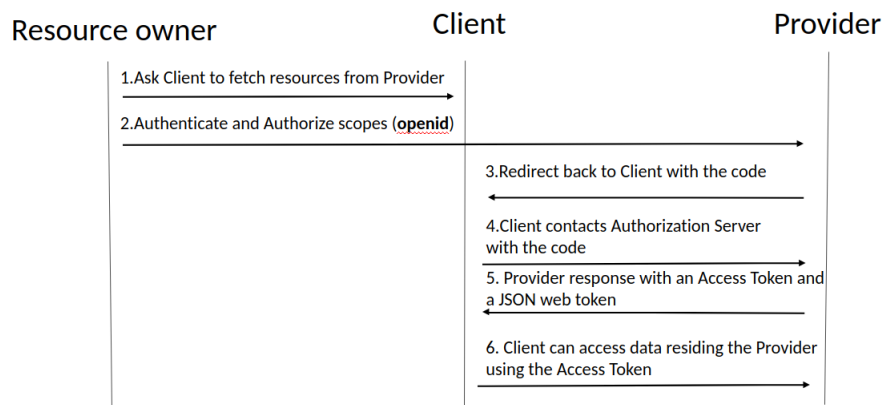

Fig. 2. OIDC flow

OIDC flow (fig. 2) is the same with OAuth2.0 flow with some extra steps.

(1) the Resource Owner, want to allow the Client the Client, to access data residing the Provider.

(2) The Client redirects browser to the Authorization Server and includes with the request the Client ID, Redirect URI, Response Type, and one or more Scopes it needs. The Authorization Server verifies the user, and if necessary prompts for a login. The Authorization Server presents user with a Consent form based on the Scopes (a specific scope **openid** is used) requested by the Client. User grant (or deny) permission.

(3) The Authorization Server redirects back to Client using the Redirect URI along with an Authorization Code.

(4) The Client contacts the Authorization Server directly (does not use the Resource Owner's browser) and securely sends its Client ID, Client Secret, and the Authorization Code.

(5) The Authorization Server verifies the data and responds with an Access Token and a **JSON Web Token**.

The Client can now use the Access Token to send requests to the Resource Server for accessing data and has information about the Resource Owner using the JWT.

2.3 Lightweight Directory Access Protocol

The Lightweight Directory Access Protocol (LDAP) is one of the core authentication protocols that was developed for directory services. LDAP historically has been used as a database of information, primarily storing information like **users**, **attributes about those users**, **group membership privileges** and more. This information was then used to enable authentication to IT resources such as an application or server. They would be pointed to the LDAP database, which would then validate whether that user would have access to it or not. That validation would be done by passing a user's credentials. LDAP authentication follows the client/server model. In this scenario, the client is generally an LDAP-ready system or application that is requesting information from an associated LDAP database and the server is, of course, the LDAP server. The server side of LDAP is a database that has a flexible schema. In other words, not only can LDAP store username and password information, but it can also store a variety of attributes including address, telephone number, group associations, and more. As a result, a common LDAP use case is to store core user identities. In doing so, IT can point LDAP-enabled systems and applications (for example) to an associated LDAP directory database, which acts as the source of truth for authenticating user access.

We add LDAP authentication in our OIDC Provider using the **auth-ldap** [1] package. This package is a Django authentication backend that authenticates against an LDAP service (in our case FORTH'S LDAP Server). Configuration can be as simple as a single **distinguished name** (DN) template, but there are many rich configuration options for working with users, groups, and permissions. For authentication with the LDAP server, we need to add the LDAP backend from the auth-ldap package and point to the LDAP server.

Now that we can talk to the LDAP server, the next step is to authenticate a username and password. There are two ways to do this, called **search/bind** and **direct bind**. The first one involves connecting to the LDAP server either anonymously or with a fixed account, and searching for the distinguished name of the authenticating user. Then we can attempt to bind again with the user's password. The second method is to derive the user's DN from his username and attempt to bind as the user directly.

We used Direct Bind, to skip the search phase. All we need to do was to set `AUTH_LDAP_USER_DN_TEMPLATE` to a template that will produce the authenticating user's DN directly. This template should have one placeholder, %(user)s. By setting `AUTH_LDAP_USER_ATTR_MAP` we are able to match our user's model attributes with the LDAP user. With these settings, we are able to connect to an LDAP server. When the LDAP server authenticates the user's credentials, the auth-ldap backend will return an LDAP_user and will create a Django user on return with the attributes that were given.

The authentication scheme from a LDAP database with OIDC flow (fig. 2), follows these steps:

(1) the Resource Owner, want to allow the Client the Client, to access data residing the Provider.
(2) The Client redirects browser to the Authorization Server and includes with the request the Client ID, Redirect URI, Response Type, and one or more Scopes it needs.
(3) The Authorization Server validates given credentials with the remote LDAP database,
(4) If user exists, returns the user and create a local Django user. The Authorization Server presents user with a Consent form based on the Scopes (a specific scope **openid** is used) requested by the Client. User grant (or deny) permission.
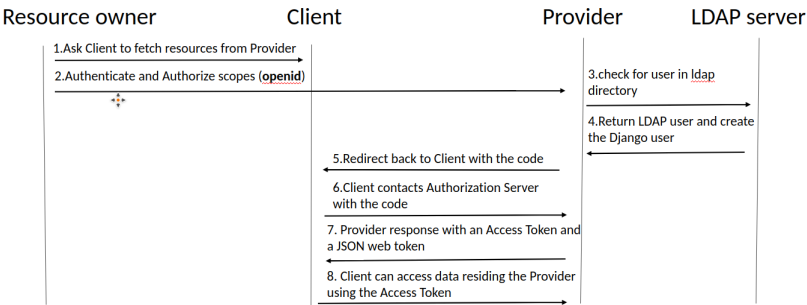
Fig. 3. OIDC with LDAP flow

(5) The Authorization Server redirects back to Client using the Redirect URI along with an Authorization Code.

(6) The Client contacts the Authorization Server directly (does not use the Resource Owner's browser) and securely sends its Client ID, Client Secret, and the Authorization Code.

(7) The Authorization Server verifies the data and responds with an Access Token and a **JSON Web Token**.

3 IMPLEMENTATION

In this section, we will take a closer look at how the implementation works. Starting from the Client and following the Providers, showing how we add the components on each application, what extra code we use to enable different authentication and authorization schemes.

3.1 The Client

The client is a Django application with a custom user model. Django comes with a user authentication system. It handles user accounts, groups, and permissions, using cookie-based user sessions. For authenticating, Django searches for the given user's credentials in its backends, and if the user exists, Django authenticates and redirects the user to the proper URL. We implement 2 custom backends for the Client application, one for searching users in the OAuth Provider and one for OIDC Provider.

We installed the Python **django-social-auth** [3] module, which is a package for authenticating against third-party applications, like GitHub, Google, Facebook, and other websites. It also contains the base class for the OAuth authentication backend, called BaseOAuth2. In our custom OAuth backend, we override some of the BaseOAuth2 components, so the backend knows where to search for the Authorization Server, where to get the token, and other details as we will see in fig. 5.

For the OIDC custom backend, all we need to do is to overwrite the OpenIdConnectAuth class, which django-social-auth also provides, provide the name of our backend and the URL of the Authorization Server. The Client is ready to redirect the Resource User to the Providers and initiate the OAuth2.0/OIDC process.

3.2 The Providers

As mentioned, we used the Django-OAuth-Toolkit for the implementation of the Providers. To enable the Django-OAuth-Toolkit we need to add `"oauth2_provider"` in installed apps and add the middleware `"oauth2_provider.middleware.OAuth2TokenMiddleware"`

```
1   from urllib.parse import urljoin
2   #import jwt
3
4   from requests import HTTPError
5   from django.contrib.auth.models import User
6   from social_core.backends.oauth import BaseOAuth2
7
8
9   class clientappOAuth2(BaseOAuth2):
10      """Custom Django OAuth authentication backend"""
11      name = 'clientapp'
12      AUTHORIZATION_URL = 'http://localhost:8001/oauth/authorize/'
13      ACCESS_TOKEN_URL = 'http://localhost:8001/oauth/token/'
14      ACCESS_TOKEN_METHOD = 'POST'
15      SCOPE_SEPARATOR = ','
16      REDIRECT_STATE = False
17      STATE_PARAMETER = True
18      SEND_USER_AGENT = True
19      ID_KEY="username"
20
21      def get_user_details(self, response):
22          """Return user details from my provider account"""
23          return response
24
25
26      def user_data(self, access_token, *args, **kwargs):
27          return self.get_json('http://127.0.0.1:8001/userinfo', headers={
28              'Authorization': 'Bearer ' + access_token
29          })
30
```

Fig. 4. OAuth custom backend

```
1   from social_core.backends.open_id_connect import OpenIdConnectAuth
2
3   class mycustomOIDC (OpenIdConnectAuth):
4       name = 'mycustomoidc'
5       OIDC_ENDPOINT = 'http://localhost:8002/oauth'
6
```

Fig. 5. OIDC custom backend

in settings.py. The next step is to add the toolkit's backend for OAuth 2.0 requests by adding `"oauth2_provider.backends.OAuth2Backend"` in authentication backends, and specify the scopes as:
`'SCOPES': {'read': 'Read scope', 'write': 'Write scope'}`. In urls.py we add an extra URL: `path('userinfo', user_info_secret)`, which is the endpoint in order for the custom backends know where to search for the user's information after authenticating. We also provide the OAuth Toolkit's endpoints by adding `url(r"^oauth/", include('oauth2_provider.urls', namespace='oauth2_provider'))`.

For the OIDC Provider implementation, we need to do two more steps. Add the specific scope of **openid** in `'SCOPES'`, and create a `CustomOAuth2Validator` method, which is used for mapping the user's info from the OIDC Provider to the Client.

Finally, we have to register the Client in both of the Providers and set the correct Client_id and Client_secret in Client's settings.py. For the OIDC implementation, we need to provide an algorithm, which is used for signing the JWT that the Provider sends back to Client, for which we used the RS256 algorithm. After the registration process, Providers are ready to give Access Tokens to Client.

In the OIDC implementation, we also add the LDAP authentication scheme, by setting:

- `AUTH_LDAP_SERVER_URL`: Points to the LDAP server.

- `AUTH_LDAP_BIND_DN`: Username.
- `AUTH_LDAP_BIND_PASSWORD`: Passwword.
- `AUTH_LDAP_USER_DN_TEMPLATE`: How to find the user.
- `AUTH_LDAP_USER_ATTR_MAP`: What attributes we want to populate the Django user who is gonna be created.

OIDC Provider is ready to authenticate users using the remote database that LDAP server provides.
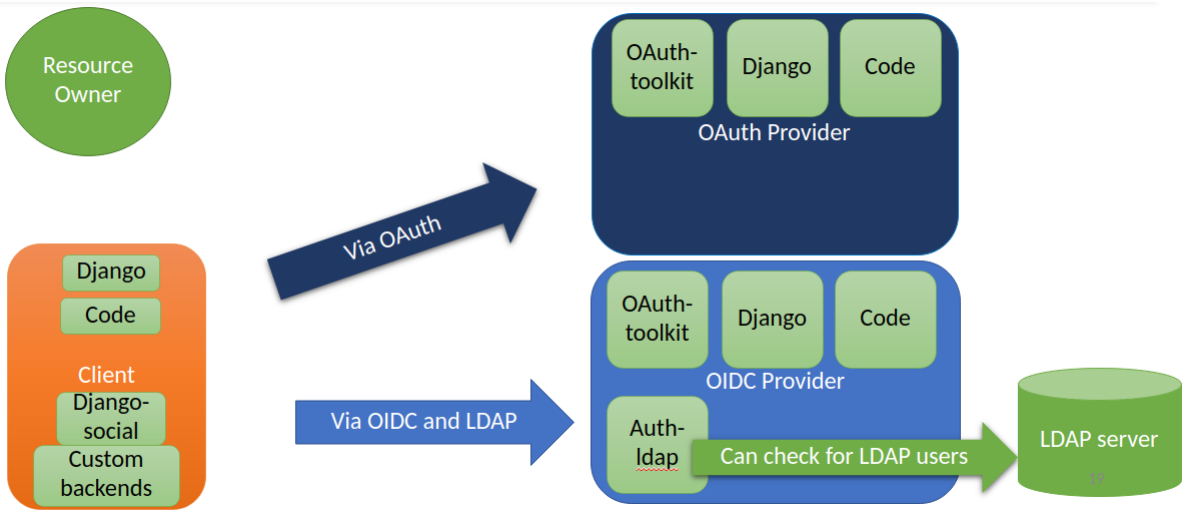

Fig. 6. Implementation

Why not add the overall figure from the presentation and describe it?

4 CONCLUSION

We managed to enable both OAuth 2.0 and OIDC protocols to authorize and authenticate between Django applications. We built two applications with Django, and made them Providers with Django-OAuth-Toolkit, acting as Resource Servers and Authentication Servers, one for OAuth and one for OIDC. With Django we create the Client, an application for users to login via the Providers, made possible by creating 2 custom backends (one for OAuth2.0 and one for OIDC) which override the base OAuth class that the django-social-auth provides and we managed to authenticate and authorize from the Client application. With the auth-ldap package, we added LDAP authentication functionality to the OIDC implementation of the provider using FORTH's LDAP server. Code is available at https://github.com/CARV-ICS-FORTH/django-oauth2-oidc-example.

ACKNOWLEDGMENTS

REFERENCES

[1] Django Authentication Using LDAP. https://django-auth-ldap.readthedocs.io/en/latest/.
[2] Django OAuth Toolkit Documentation. https://django-oauth-toolkit.readthedocs.io/en/latest/.

[3] Django social auth documentation. https://python-social-auth.readthedocs.io/en/latest/configuration/django.html.

[4] Django, the high-level Python web framework. https://www.djangoproject.com/.

[5] LDAP, the Lightweight Directory Access Protocol. https://ldap.com/.

[6] OAuth 2.0, the industry-standard protocol for authorization. https://oauth.net/2/.

[7] OpenID Connect, a simple identity layer on top of the OAuth 2.0 protocol. https://openid.net/connect/.