Lab 3

Modern Web Authentication and Authorization

Review Question

- What is OAauth2 ?

- What is OpenID Connect ?

- What are the main vulnerabilities of JWT Tokens ?

Activity 1: OAuth2 and OIDC with Keycloack

KeyCloak is an open-source identity and access management solution by Red Hat which makes it easy to secure modern applications and services with little to no code. Keycloak provides out-of-the-box authentication and authorization services as well as advanced features like User Federation, Identity Brokering, and Social Login.

KeyCloak Instance Configuration

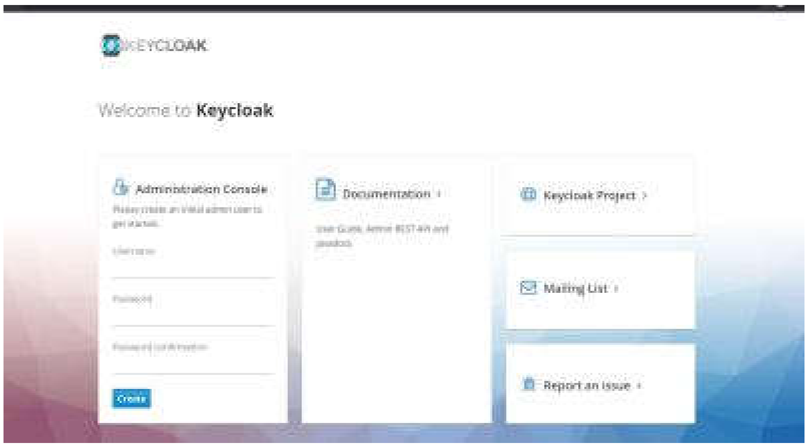
To configure KeyCloak, we need to have the running instance of it. There are many ways to deploy KeyCloak (using Docker, using the provided startup script, …) In this lab, we will start a Keycloack using the startup script.

- Unzip the Keyloack package provided in the training tools.

- From a terminal, cd to the bin directory of Keyloack and start it using the following command:

```
standalone.bat -Djboss.socket.binding.port-offset=100
```

This will start the Wildfly server for your Keycloak on your local machine. We can access the server by executing the URL **`http://localhost:8180`**. If you just use **standalone.bat** to execute without that parameter, the server will run on the port 8080.



Once you start the server, the first thing you will have to do is to create an admin user. We will create, for the purpose learning a user **admin** and password **Pa$$w0rd**

Login to the admin console

Go to the Keycloak Admin Console (http://localhost:8080/auth/admin) and login with the username and password you created earlier.

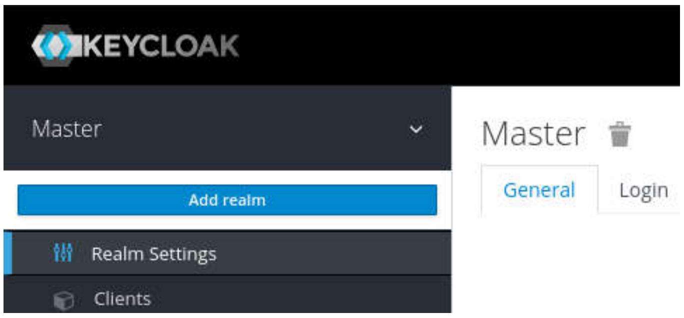
For our demo purposes, we will create a new realm **SpringBootKeycloakApp**. In this realm, we will add our Spring Boot application as a client. Create a new client on Clients tab. We will name our client application as **SpringBootApp**.

- **Create a realm**

  A realm in Keycloak is the equivalent of a tenant. It allows creating isolated groups of applications and users. By default there is a single realm in Keycloak called `master`. This is dedicated to manage Keycloak and should not be used for your own applications.

  Let's create our first realm.

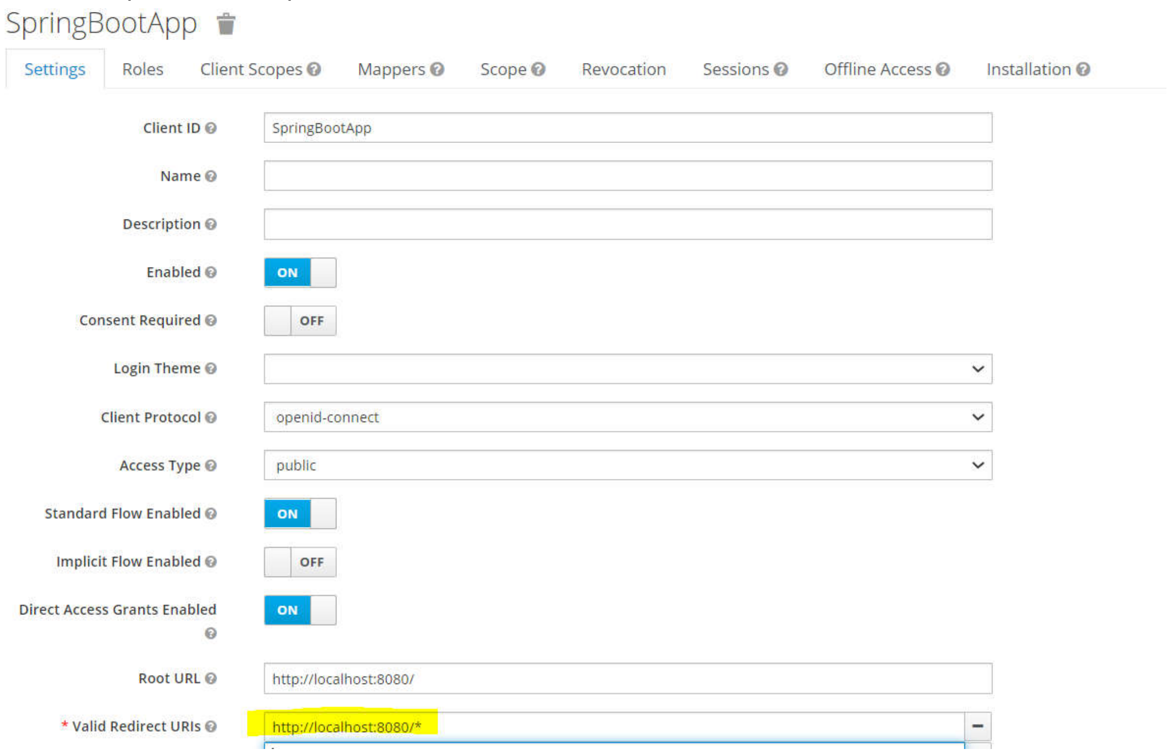
  1. Open the [Keycloak Admin Console]()
  2. Hover the mouse over the dropdown in the top-left corner where it says Master, then click on Add realm
  3. Fill in the form with the following values:
     - Name: **SpringBootKeycloakApp**
  4. Click Create



- **Adding Application (The Client)**

  Create a new client on Clients tab. We will name our client application as **SpringBootApp**.

  Now in settings, we will add redirect URL for our Spring Boot Application. This is the URL where Keycloak will redirect to our app after authentication. Also, we are using **openid** connect as a protocol as part of this implementation.

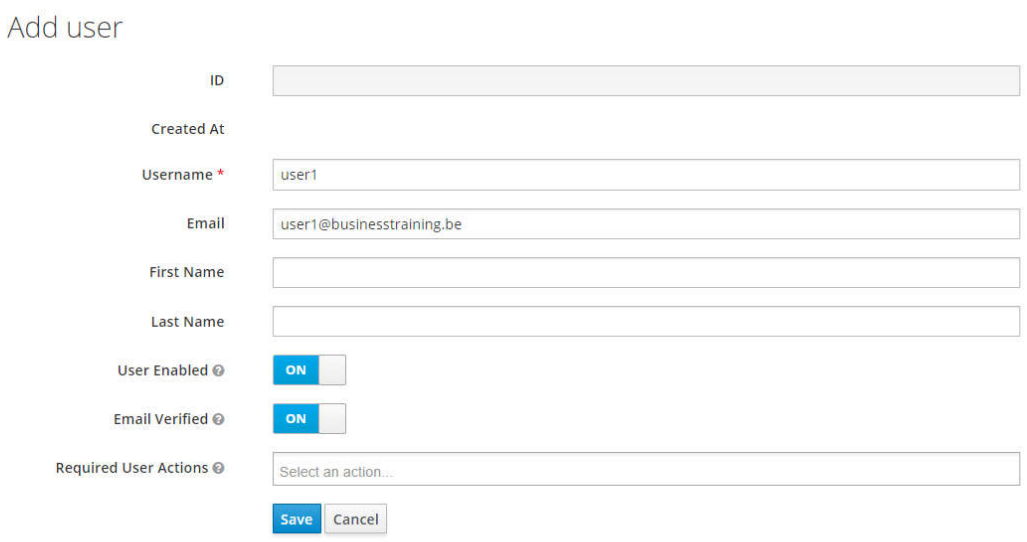


Client ID	SpringBootApp
Name	
Description	
Enabled	ON
Consent Required	OFF
Login Theme	
Client Protocol	openid-connect
Access Type	public
Standard Flow Enabled	ON
Implicit Flow Enabled	OFF
Direct Access Grants Enabled	ON
Root URL	http://localhost:8080/
* Valid Redirect URIs	http://localhost:8080/*

- **Adding User and Role**

Now we will add a user that we will use to authenticate. We will use this user to login to our sample Spring Boot application.

Add a role that you want for this user **ROLE_User** on the roles tab in Keycloak. Once that is done, let's go to the Users tab and add a new user. Name the user **user1** and use the Role Mappings tab, to add it to the group **ROLE_User**. Set a **password** for user user1 using the credentials tab.



Add user

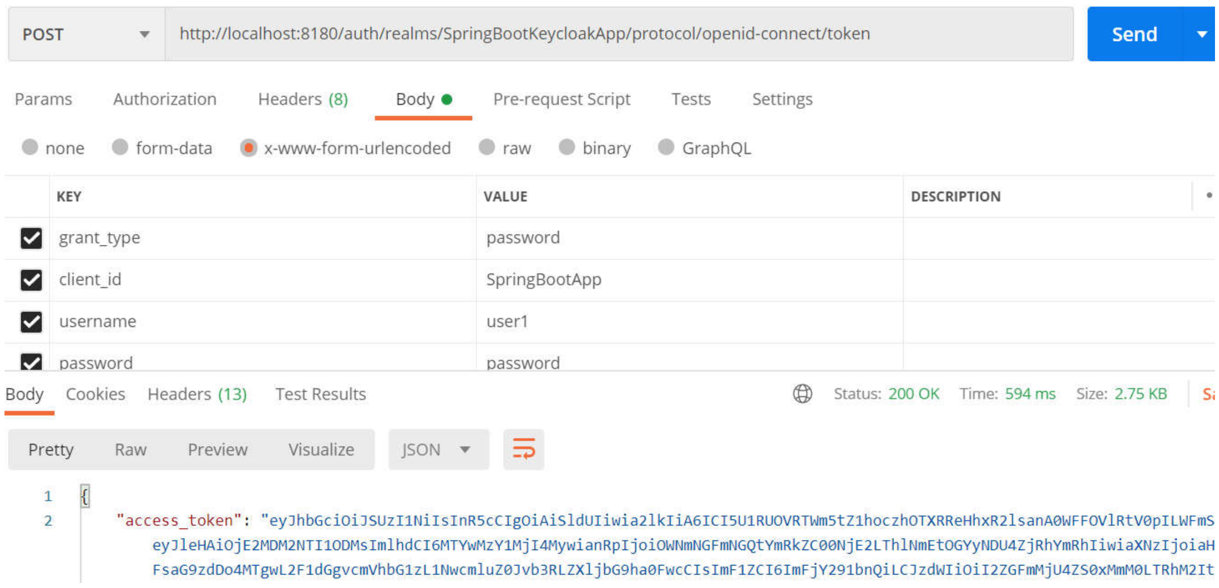
ID	
Created At	
Username *	user1
Email	user1@businesstraining.be
First Name	
Last Name	
User Enabled	ON
Email Verified	ON
Required User Actions	Select an action...

- **Generate Tokens**

This step is optional, and it is just for getting familiar with JWT tokens. We can generate the token by using the below endpoint:

```
http://localhost:8180/auth/realms/SpringBootKeycloakApp/protocol/openid-
connect/token
```

We can use PostMan or Advanced REST Client.



POST http://localhost:8180/auth/realms/SpringBootKeycloakApp/protocol/openid-connect/token

	KEY	VALUE	DESCRIPTION
✓	grant_type	password	
✓	client_id	SpringBootApp	
✓	username	user1	
✓	password	password	

Status: 200 OK Time: 594 ms Size: 2.75 KB

```
{
    "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI5U1RUOVRTWm5tZ1hoczhOTXRReHhxR2lsanA0WFFOVlRtV0pILWFmS
        eyJleHAiOjE2MDM2NTI1ODMsImlhdCI6MTYwMzY1MjI4MywianRpIjoiOWNmNGFmNGQtYmRkZC00NjE2LThlNmEtOGYyNDU4ZjRhYmRhIiwiaXNzIjoiaH
        FsaG9zdDo4MTgwL2F1dGgvcmVhbG1zL1NwcmluZ0Jvb3RLZXljbG9ha0FwcCIsImF1ZCI6ImFjY291bnQiLCJzdWIiOiI2ZGFmMjU4ZS0xMmM0LTRhM2It
```

We can make the request even CURL.

```
Curl.exe -H "Content-Type: application/x-www-form-urlencoded" -d
"username=user1&password=password&grant_type=password&client_id=SpringBootApp"
http://localhost:8180/auth/realms/SpringBootKeycloakApp/protocol/openid-connect/token
```

- **Using Keycloak in a Java Spring Boot application**

  Now, we are going to create a simple Spring Boot application that will use Keycloak for security. As part of this application, we will be showing a list of to-do list tasks for the user who will authenticate with the application. The Keycloak maven dependency includes Keycloak client adapters for authentication purposes. They will replace our standard Spring Security adapters.

  Since we will be authenticating with Keycloak, we will need a session for user's state. We are using **RegisterSessionAuthenticationStrategy** here. HttpSessionManager is a conditional bean because Keycloak already implements that bean.

  Starting the application

  - Clone the Spring application from GitHub

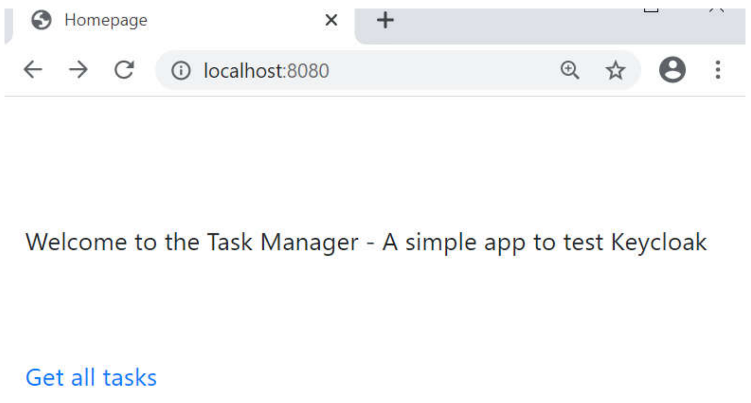
    ```
    git clone https://github.com/mromdhani/MySprinBootKeycloackApp
    ```

  - Change directory into MySprinBootKeycloackApp, and build and lauch the application using Apache Maven wrapper.
    Check that Apache maven is installed and it is the system PATH. Check also that the port 8080 is free before launching the application

    ```
    mvn spring-boot:run
    ```

    - Once the application started, open up your browser and connect to http://localhost:8080. our keycloak application, it will be running on http://localhost:8180. Our Spring Boot application will be running at http://localhost:8080.

      Our first screen of the Spring Boot application will look like below:



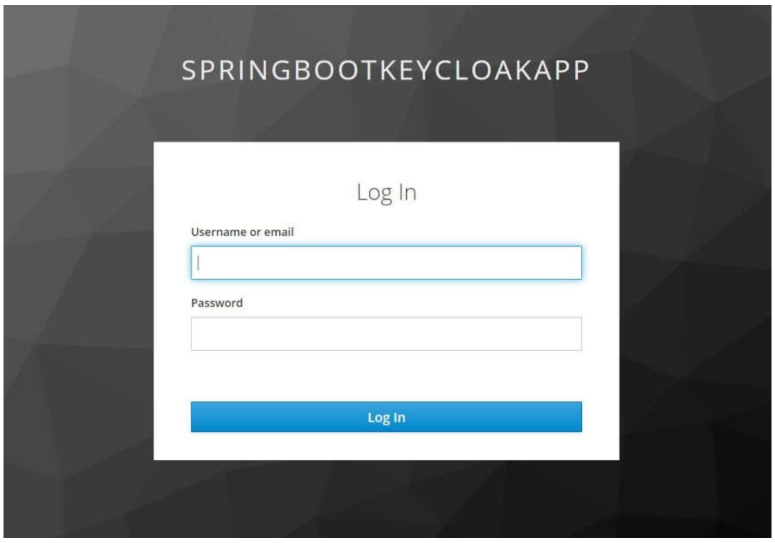
Homepage

localhost:8080

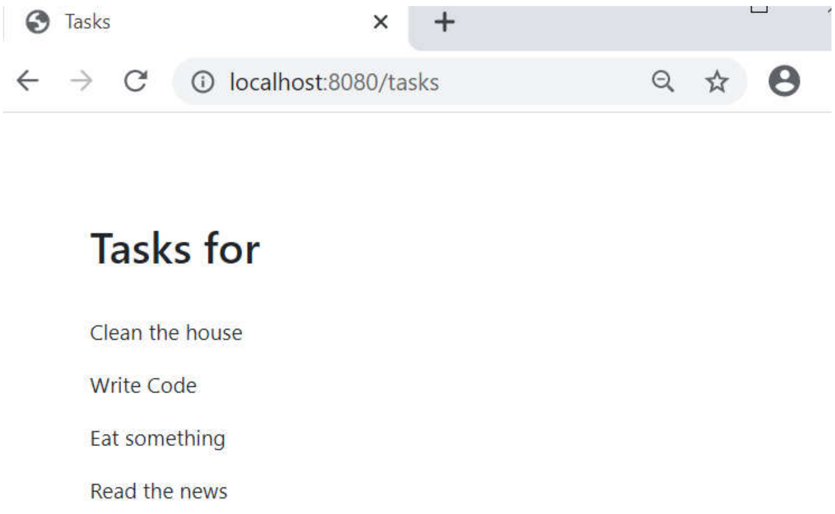
Welcome to the Task Manager - A simple app to test Keycloak

Get all tasks

      Now if a user clicks on Get all tasks, he will be redirected to Keycloak login screen as below:



Now, I will enter my user **user1** username and password and it will show us our list of tasks as follows:



Tasks for

Clean the house

Write Code

Eat something

Read the news

The Authentication Flow explained

When the user clicks on Get all tasks, the user is redirected to Spring Security's sso/login endpoint which KeycloakSpringBootConfigResolver handles and sends an authorization code flow request to Keycloak

```
http://localhost:8180/auth/realms/SpringBootKeycloakApp/protocol/openid-
connect/auth?response_type=code&client_id=SpringBootApp&redirect_uri=http%3A%
2F%2Flocalhost%3A8080%2Fsso%2Flogin&state=70bd4e28-89e6-43b8-8bea-
94c6d057a5cf&login=true&scope=openid
```

Keycloak will process the request to respond with a session code and show the login screen.

Once the user enters credentials and keycloak validates those, it will respond with an authorization code, and this code is exchanged for a token, and the user is logged in.

Activity2: Implementing Authentication using ECAS (EU Login) / Tomcat 8 Client

This lab illustrates the configure ECAS Authentication Services for Web applications developed in Java and deployed on Tomcat 8 Container. The configuration steps are detailed in EU Confluence[https://webgate.ec.europa.eu/CITnet/confluence/display/IAM/ECAS+Tomcat+Client]. Similar guides for PHP, Java EE and other technologies are available on confluence.

Requirements

In order to install the ECAS client for Apache Tomcat, you need:

- JRE/JDK 6.0 or JRE/JDK 7.0 or JRE/JDK 8.0 (for Tomcat 8 only)
- An installation of Apache Tomcat 6.0, 7.0 or 8.0
- A text editor to modify XML configuration files
- The CommisSign PKI certificates, available from the Forge
- The latest ECAS client for Apache Tomcat 6.0, Tomcat 7.0 or Tomcat 8.0 available from the Forge
- The Tomcat 5.5, Tomcat 7.0 or Tomcat 8.0 configuration files, available from the Forge
- The ecas-demo Web application for Apache Tomcat 6.0 or 7.0 available from the Forge.

Installation

Trusting the CommisSign PKI

First, you need to be able to open SSL connections from your application to the ECAS server in order to validate ECAS tickets.

For that purpose, you need to import the CommisSign PKI certificates into your Java trustStore as the SSL certificate of the ECAS server is issued by the CommisSign PKI.

Get the two **CommisSign PKI certificates** (called "European Commission Root CA - 2.cer" and "CommisSign - 2.cer") from the forge, then you have to import them into the Java trustStore of the JVM you will use.

This default trustStore is usually located at ${JRE_HOME}/lib/security/cacerts and its default password is "changeit".

You can do the import with the keytool command-line utility from the JDK or a GUI tool such as portecle, keytoolGUI or KeyMan.

With keytool, it would look like:

```
D:\java\jdk1.8.0_161\jre\lib\security>keytool -import -v -keystore
cacerts -storepass changeit -alias EuropeanCommissionRootCA -file
EuropeanCommission2.cer
D:\java\jdk1.8.0_161\jre\lib\security>keytool -import -v -keystore cacerts
-storepass changeit -alias CommisSign2 -file CommisSign2.cer
```

Copy the client JAR

Get the last version of the ECAS client for Apache Tomcat 8.0 and copy into your installation of Tomcat at ${tomcat.home}/lib (where ${tomcat.home} is the directory where you installed tomcat)

The ECAS client uses Log4J for its logging statements, so you also have to copy log4j jar (for instance log4j-1.2.17.jar) to ${tomcat.home}/lib

Add the ECAS Authenticator for Tomcat

Extract the content of ecas-tomcat-8.0-3.11.2-config.zip into ${tomcat.home}/lib

This way, you added two files in /lib:

in org/apache/catalina/authenticator you now have mbeans-descriptors.xml which is the JMX xml description of the ECAS Authenticator for Tomcat in org/apache/catalina/startup/Authenticators.properties which registers the ECAS auth-method besides the standard authentication methods you can specify in web.xml descriptors

Your ${tomcat.home}/lib folder should look like:

Activate ECAS authentication in your application

In order to activate ECAS authentification in your application, you have to add the following snippet to your web.xml:

```
<login-config>
    <auth-method>ECAS</auth-method>
    <realm-name>ecas-demo Realm</realm-name>
</login-config>
```

Allow authenticated users who have no roles

The meaning of `<role-name>*</role-name>` in the web.xml has changed.
It now means that only authenticated users that possess at least one role from the ones configured in your web.xml are authorized to proceed.

To get back the previous behaviour i.e. granting access to all authenticated users regardless of whether they possess a role or not, you have to edit the file `${tomcat.home}/conf/server.xml`. Locate the Realm element. This should look like:

```
Realm className="org.apache.catalina.realm.LockOutRealm"
allRolesMode="authOnly">
  <Realm className="org.apache.catalina.realm.UserDatabaseRealm"
         resourceName="UserDatabase" />
</Realm>
```

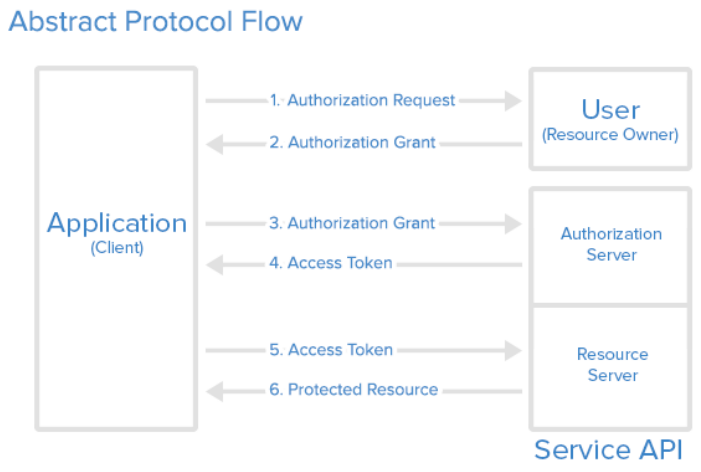
You do not need this if you always use explicit roles instead of `<role-name>*</role-name>`.

Answers to Review Questions

What is OAauth2 ?

OAuth 2 (RFC 6749 created in 2012) is an authorization framework that enables applications to obtain limited access to user accounts on an HTTP service, such as Facebook, Google, GitHub. It works by delegating user authentication to the service that hosts the user account, and authorizing third-party applications to access the user account. OAuth 2 provides authorization flows for web and desktop applications, and mobile device

Here is a more detailed explanation of the abstract flow in the diagram:



1. The *application* requests authorization to access service resources from the *user*
2. If the *user* authorized the request, the *application* receives an authorization grant
3. The *application* requests an access token from the *authorization server* (API) by presenting authentication of its own identity, and the authorization grant
4. If the application identity is authenticated and the authorization grant is valid, the *authorization server* (API) issues an access token to the application. Authorization is complete.
5. The *application* requests the resource from the *resource server* (API) and presents the access token for authentication
6. If the access token is valid, the *resource server* (API) serves the resource to the *application*

The actual flow of this process will differ depending on the authorization grant type in use, but this is the general idea.

What is OpenID Connect ?

OpenID Connect (created in early 2014) is a simple identity layer on top of the OAuth 2.0 protocol. It allows 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.

OpenID Connect allows clients of all types, including Web-based, mobile, and JavaScript clients, to request and receive information about authenticated sessions and end-users. The specification suite is extensible, allowing participants to use optional features such as encryption of identity data, discovery of OpenID Providers, and session management, when it makes sense for them. The structure of the Access Token are undefined by default. This ambiguity guarantees that Identity Providers will build incompatible systems.

OIDC brings some “sanity” to OAuth2. It adds and strictly defines an ID Token for returning user information. Now when we log in with our Identity Provider, it can return specific fields that our applications can expect and handle. The important thing to remember is that OIDC is just a special, simplified case of OAuth, not a replacement. It uses the same terminology and concepts.

What are the main vulnerabilities of JWT Tokens ?

These are 5 vulnerabilities and other key considerations while working with authentication tokens:

1. You are not using strong tokens

You should always keep track of how strong and un-guessable your session tokens are. They should be generated in a manner that any attacker who obtained a large sample of session ID’s from the application could never predict or extrapolate the tokens issued to other users.

Some general rules for generating strong tokens:

- Use an extremely large set of possible values.
- Work with a strong source of pseudo randomness, ensuring an even and unpredictable spread of tokens across the range of possible values.

- Make the tokens long enough (at least 16 bytes).

2. You transport on an open connection

Suppose all your web pages are under HTTPS protocol, but for some reason, you thought that loading that special image (or style file) would be a lot better on the open channel. In the end, if token storages are not secured properly, an attacker could steal your authentication secrets just by looking at those extra file requests without needing to break the username-password lock.

3.Your token storage is not bulletproof

When application security is at the top of the list, you should store tokens in cookie storage. Two special cookie attributes in mind are:

- **Secure cookies** are important in case you forget to transport all your applications under HTTPS.
- **HTTP only** helps so that client-side JavaScript won't reach any precious information. In the case of XSS, this might lower the probability of doing harm on behalf of power users of the application.

Establish path rules for cookies. Adding the stricter path (**same-site**) policy could help, too. If your main website (e.g. example.com) went through pre-production audit and someone forgot to put the correct domain attribute for the cookie, hackers could steal the token if you have a not-so-secure subdomain website (e.g. help.example.com) being hosted on the side.

4.You don't manage token expiration time

Renewal time is just a variable which defines in minutes or seconds how often the renewal of token will happen. The general recommendation here is to refresh the token as often as possible. It's best not to allow one token to be valid for a long time.

5. You allow the user to have multiple logins at the same time.

When thinking about secure authentication management, it can mean having less comfortable user experience at times.

For robust systems, consider outlawing parallel authentications for the same user. Overall, before implementing authentication restrictions, think about what is more important for your application clients—convenience or security. And if it's the latter, don't hesitate to take drastic measures on parallel authentication handling. If there is a requirement to keep the same user session between web and mobile apps, the implementation will not be straightforward.