

Exercise 11b

Using OAuth introspection with a third-party identity server to replace client-certificates

Prior Knowledge

Previous exercises

Objectives

Replace SSL client authentication with the use of an OAuth2 token

Software Requirements

(see separate document for installation of these)

- Docker (and thereby WS02 IS 5.1.0)
- Node.js
- Python

Overview

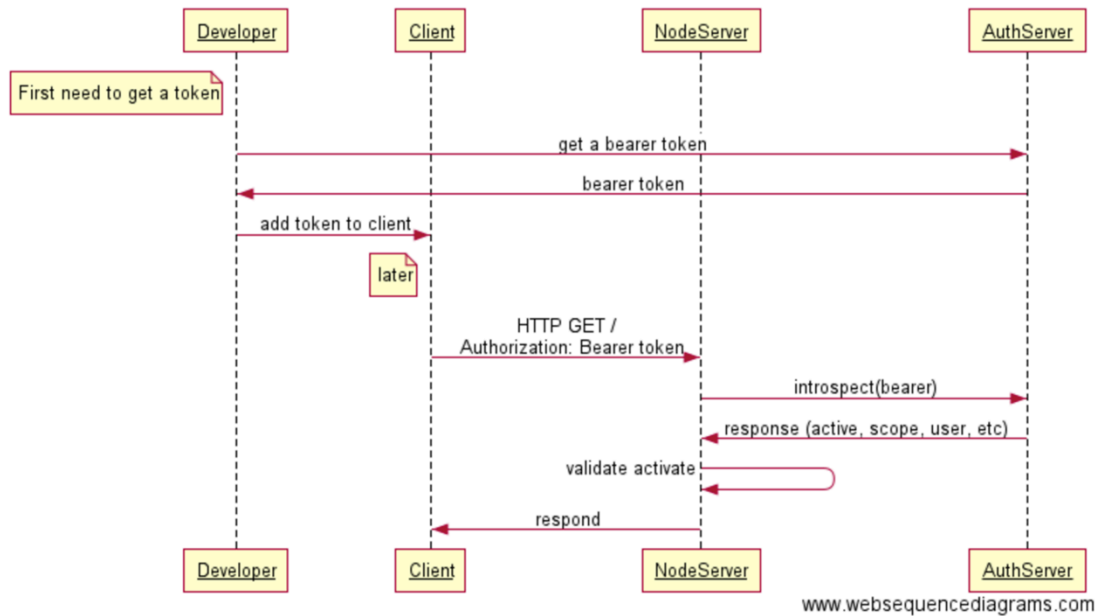
In this lab, we will use server-side TLS to validate the server to the client, and use an OAuth2 token to validate the client to the server.

The lab will follow the following overall approach:

1. *Run a docker image of the WS02 Identity Server to provide an OAuth2 server.*
2. *Create the OAuth2 client definition inside the Identity Server.*
3. *Issue the token and get approval to access a given scope using client credentials.*
4. *Enhance it to use the authorization grant.*
5. *Enhance the node.js server to validate the token and the scope using the OAuth2 Introspection API*

Here is a picture of what we are looking at:

OAuth2 authentication flow



Steps

1. I have created a docker image of the WSO2 Identity Server 5.1.0, and this also includes an add-in that supports OAuth2 Introspection.

```
sudo docker run -d -p 9443:9443 -p 9763:9763  
pizak/wso2is:5.1.0-introspection
```

2. I have also enhanced the code of the SSL exercise to support OAuth2 tokens. Copy this code to your main directory:

```
cp -r ~/repos/ox-soa2/code/sec_oauth/ sec_oauth  
cd ~/sec_oauth
```

3. Ensure you have the right Node dependencies for this:
`npm install querystring`

4. Now look at the file server/server.js

The main changes are in bold.:

```
var Introspect = require('./introspect.js');

introspect = new Introspect("localhost", 9763, "/introspect");

app.get("/", function(req, res){
  console.log(req.headers);
  auth = req.headers.authorization;

  if (!auth) res.sendStatus(401);

  bearer = introspect.getBearer(auth);
  if (bearer) introspect.introspect(bearer, function (username, scope) {
    if (!username) {
      res.status(401).send();
    }
    else {
      obj = {random : Math.floor(Math.random() * 100) + 1),
        username:username,
        scope:scope};
      res.json(obj);
    }
  });
});
```

This is importing a new module called Introspect. Introspect is configured with a introspection endpoint (localhost:9763/introspect).

We extract the bearer token from the headers and if it exists we call introspect. Notice that since this is node.js, we need to preserve the non-blocking nature of the system, so we pass a callback to Introspect that will actually do our "GET" logic. This is because Introspect is going to call out to another HTTP endpoint and we don't want to block node threads while it is waiting for the response.

If introspect succeeds, it passed a username and scope to the callback, otherwise null, null.

5. What we need introspect to do is to call an introspection API defined in <https://tools.ietf.org/html/rfc7662>

Here is a sample mitmdump:

```
oxsoa@oxsoa: ~  
127.0.0.1 POST http://localhost:9763/introspect  
Content-Type: application/x-www-form-urlencoded  
host: localhost:9700  
Connection: close  
Transfer-Encoding: chunked  
  
token: 2a07bc21c725b0610dd274b354c82d6f  
token_type_hint: Bearer  
  
<< 200 OK 161B  
Date: Tue, 07 Jun 2016 18:50:58 GMT  
Content-Type: application/json  
content-length: 161  
Connection: close  
Server: WS02 Carbon Server  
  
{  
  "active": true,  
  "client_id": "lnfwkuzhClQ1xbfaGpfvKe7HowQa",  
  "exp": 1465329032,  
  "iat": 1465325432,  
  "scope": "user",  
  "token_type": "Bearer",  
  "username": "admin@carbon.super"  
}  
  
127.0.0.1:34754: clientdisconnect
```

6. Now lets look at introspect.js
The main logic we care about is here:

```
introspect : function (token, callback) {
  console.log(token);
  data = { token : token,
           token_type_hint : "Bearer"
         }
  encoded = qs.stringify(data);

  var post_options = {
    host: introspect_host,
    port: introspect_port,
    path: introspect_path,
    method: 'POST',
    headers: {
      'Content-Type': 'application/x-www-form-urlencoded'
    }
  };

  var post_req = http.request(post_options, function(r) {
    var body = ""
    r.setEncoding('utf8');
    r.on('data', function (chunk) {
      body += chunk;
    });
    r.on('end', function() {
      try {
        var response = JSON.parse(body);
      } catch (e) {}

      if (response && response.active) {
        callback(response.username, response.scope);
      }
      else
      {
        callback(null,null);
      }
    });
  });

  // post the data
  post_req.write(encoded);
  post_req.end();
}
```

7. The code is actually pretty simple, but the async nature of node.js slightly obfuscates things. Basically we start a post operation, and will get called asynchronously as chunks of response come in. This is slightly overkill as the response to a introspection call will probably always fit in a network buffer, but this is good node.js coding.

8. I have also modified random-client.py to look for a bearer token in the command-line:

```
bearer = ""
headers = dict()
url = "https://localhost:8443"
if (len(sys.argv) > 1):
    bearer = sys.argv[1]
    headers = dict(Authorization="Bearer "+bearer)

h = httpplib2.Http(ca_certs="./keys/ca.cert.pem")
resp, content = h.request(url, "GET", headers = headers)
```

Now if you call:

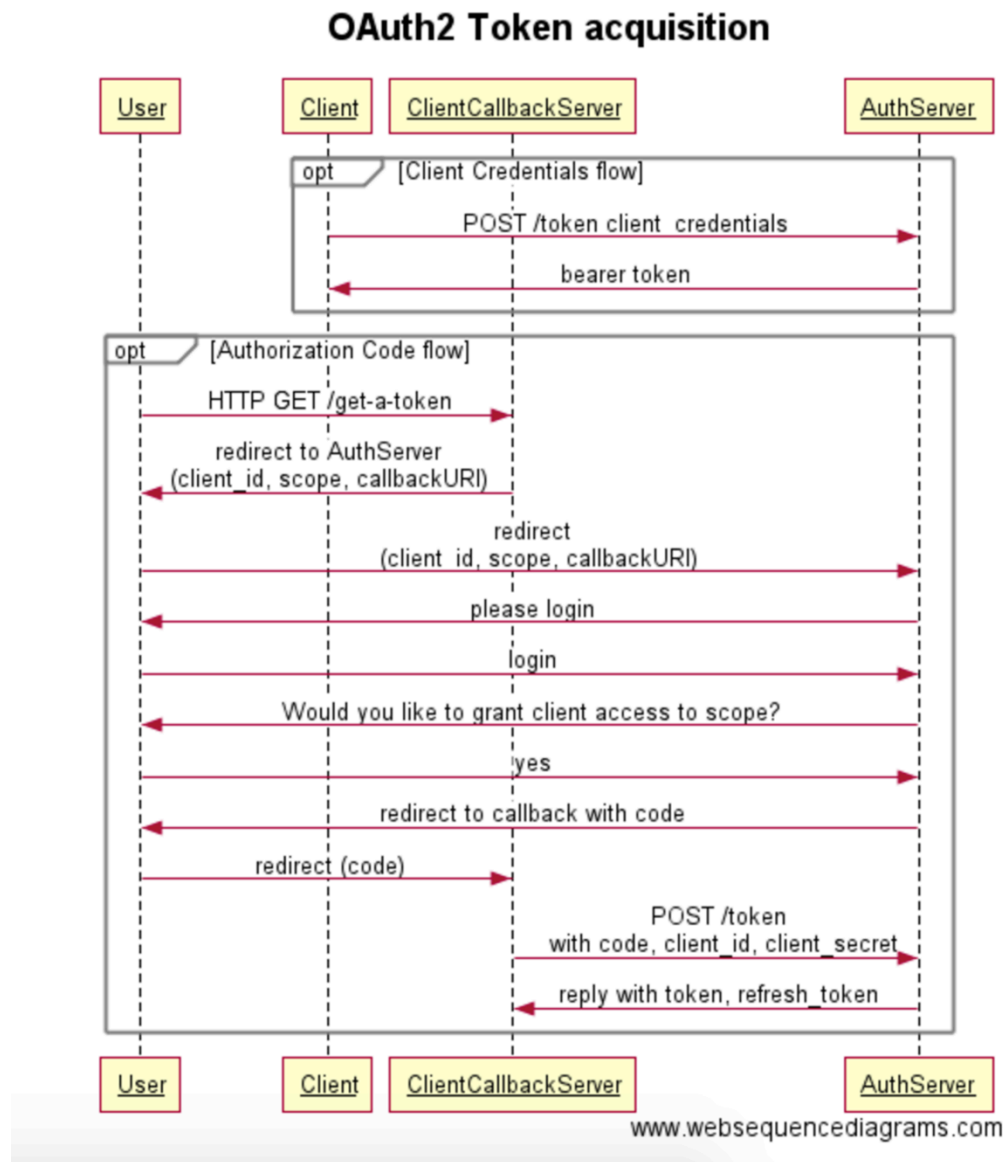
```
python random-client.py fbas78734nerjkka233a1
```

this will add the following HTTP header:

```
Authorization: Bearer fbas78734nerjkka233a1
```

There is a further slight enhancement which is to also print out the full JSON response (so we can see the extra fields we are adding).

9. Ok, we have almost everything in place! We have a OAuth2 server. We have a client that can send a bearer token, and a server that can read the token, pass it over to the OAuth2 server and validate the response.
10. There are multiple ways defined in the OAuth2 specification to get hold of the OAuth2 token. The two ways we are going to look at are documented in this sequence diagram:



11. In both approaches, before we create the token, we need to register our application to the WS02 Identity Server.

12. Browse to:

<https://localhost:9443>

Sort out any SSL issues and then login as **admin/admin**

You should see:

The screenshot shows the WSO2 Identity Server Management Console. The top navigation bar includes 'Home', 'Identity', 'Users and Roles', 'User Stores', 'Claims', 'Service Providers', 'Identity Providers', 'Entitlement', 'Policy Administration', 'Policy Publish', 'PDP', 'Policy View', 'Extension', 'Search', 'Workflow Engagements', 'Workflow Definitions', 'Keystores', and 'Shutdown/Restart'. The main content area displays system information:

Server	
Host	localhost
Server URL	local://services/
Server Start Time	2016-06-07 17:07:13
System Up Time	0 day(s) 15 hr(s) 33 min(s) 53 sec(s)
Version	5.1.0
Repository Location	file:/wso2/wso2is-5.1.0/repository/deployment/server/

Operating System	
OS Name	Linux
OS Version	4.4.0-22-generic

Operating System User	
Country	US
Home	/root
Name	root
Timezone	GMT

Java VM	
Java Home	/usr/lib/jvm/java-1.8-openjdk/jre
Java Runtime Name	OpenJDK Runtime Environment
Java Version	1.8.0_92-internal
Java Vendor	Oracle Corporation
Java VM Version	25.92-b14

Registry	
DBMS	H2
DBMS Version	1.2.140 (2010-07-25)
DBMS Driver	H2 JDBC Driver

13. In the left-hand bar, select **Service Providers -> Add**

Add the Service Provider Name: **oauth2-example**

Click **Register**

[Home](#) > [Identity](#) > [Service Providers](#) > [Add](#)

Add New Service Provider

Basic Information

Service Provider Name:*

? A unique name for the service provider

Description:

? A meaningful description about the service provider

Register

Cancel

14. Expand the Inbound Authentication Configuration

Home > Identity > Service Providers > Add > Service Providers

Service Providers

Basic Information

Service Provider Name*:
A unique name for the service provider

Description:
A meaningful description about the service provider

SaaS Application ☐
Applications are by default restricted for usage by users of the service provider's tenant. If this application is SaaS enabled it is opened up for all the users of all the tenants.

Claim Configuration

Role/Permission Configuration

Inbound Authentication Configuration

- SAML2 Web SSO Configuration
- OAuth/OpenID Connect Configuration
- OpenID Configuration ✓
- WS-Federation (Passive) Configuration ✓
- WS-Trust Security Token Service Configuration
- Kerberos KDC

Local & Outbound Authentication Configuration

Inbound Provisioning Configuration

Outbound Provisioning Configuration

Update Cancel

15. Expand **OAuth/OpenID Connect Configuration**, and then click **Configure**

Enter the callback URL: <https://localhost:8444/callback>

Home > Identity > Service Providers > Add > Register New Application

Register New Application

New Application

OAuth Version* ☐ 1.0a ☒ 2.0

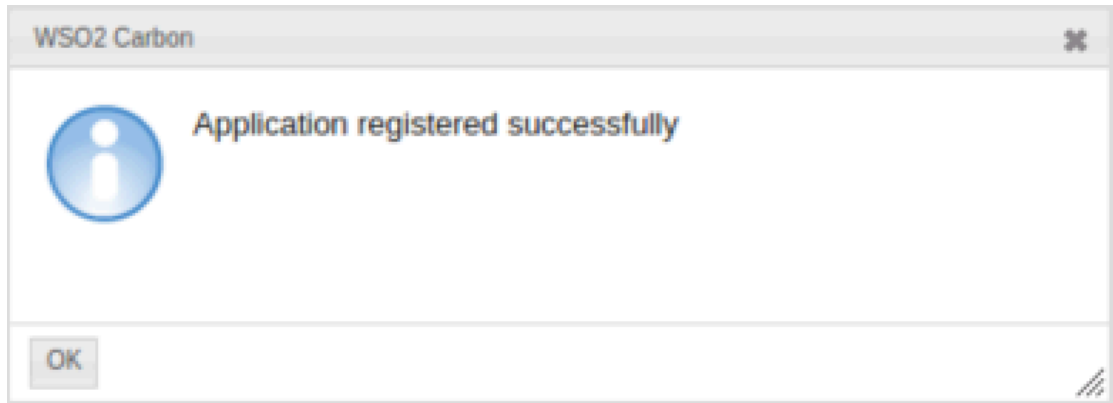
Callback Uri*

Allowed Grant Types ☒ Code ☒ Implicit ☒ Password ☒ Client Credential ☒ Refresh Token ☒ SAML2 ☒ IWA-NTLM

Add Cancel

Click **Add**

16. You should see:



Client credentials flow

17. We now have an OAuth2 Client ID and Client Secret that identifies both the “Client Application” and the user (admin) to the Auth Server.

18. Pull up ARC and create the following request:

URL: <http://localhost:9763/oauth2/token>

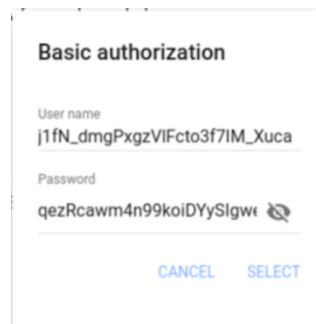
Hint: create the headers with the built in tools (**Headers Form** / then look for the pencil icon)

Headers:

Authorization:

Username: **<your client id copied from the IS web console>**

Password: **<your client secret from the IS web console>**



Now use the ARC **Data Form** to add the following parameter. It will prompt you to add the right Content-Type.

grant_type: client_credentials

scope: test

Your screen should look like:

The screenshot shows the 'Advanced REST client' interface. At the top, the URL bar contains 'http://localhost:9763/oauth2/token'. Below the URL bar, the HTTP method is set to 'POST' (indicated by a blue dot). The 'Content-Type' is set to 'application/x-www-form-urlencoded'. The 'Headers' tab is selected, showing two headers: 'authorization' with value 'Basic ajFmTI9kbWdQeGd6VmxGY3RvM2Y3SU1fWHVjYTpxZXpSY2F3bTRuOTIrb2IEWXITSWd3' and 'Content-Type' with value 'application/x-www-form-urlencoded'. The 'Data form' tab is also visible, showing two parameters: 'grant_type' with value 'client_credentials' and 'scope' with value 'test'. A 'SEND' button is located at the bottom right.

Advanced REST client

Request

Use XHR

> http://localhost:9763/oauth2/token

GET POST PUT DELETE Other methods application/x-www-form-urlencoded

Raw headers Headers form Headers sets

HTTP headers

authorization Basic ajFmTI9kbWdQeGd6VmxGY3RvM2Y3SU1fWHVjYTpxZXpSY2F3bTRuOTIrb2IEWXITSWd3

Content-Type application/x-www-form-urlencoded

ADD

Raw payload Data form Files (0)

ENCODE PAYLOAD DECODE PAYLOAD

Form data for x-www-form-urlencoded parameters

grant_type client_credentials

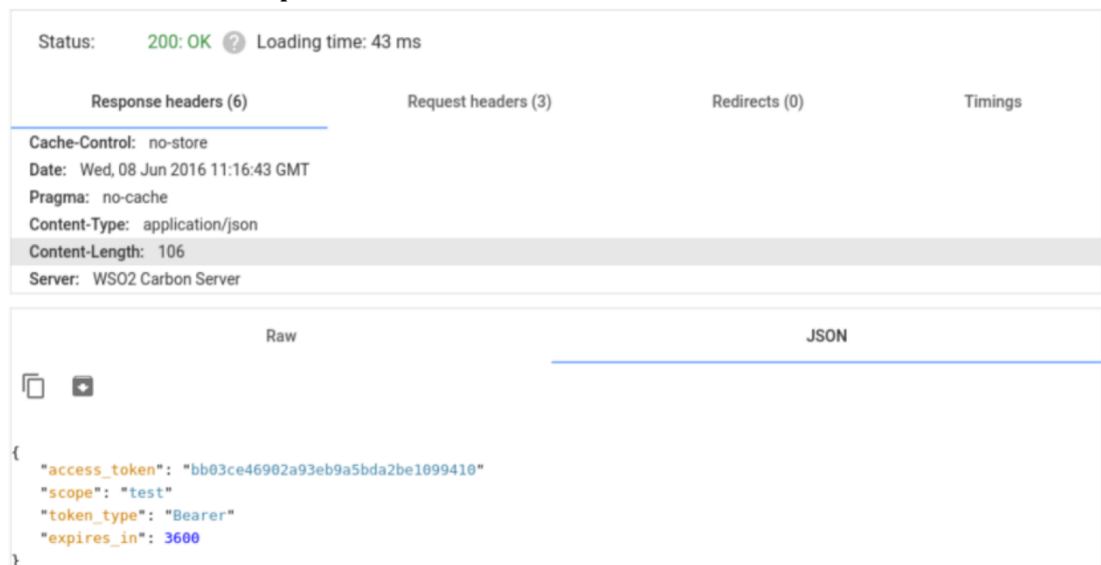
scope test

ADD ANOTHER PARAMETER

SEND

19. Now Send

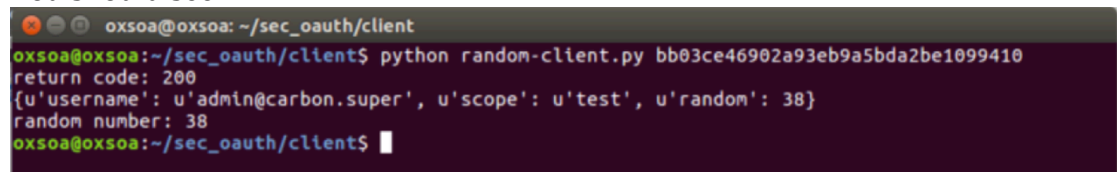
20. You should see a response like:



21. Copy that access token. Now open a new terminal window.

```
cd ~/sec_oauth/client
python random-client.py access_token_here
```

You should see:



22. Congrats, we have made the first type work.

23. This client_credentials model doesn't really implement what we would properly like, which is that a user *delegates authority* to a client to do something with a scope. The reason is that we expected the User to somehow issue this HTTP request and get the credential.

24. The better model is documented in the second flow in the sequence diagram. Please take another look.

25. To support this, we need logic running in a server¹. Normally this would be coded into the same system as the client is running as. In other words, our client is a website so it can host this logic. Since we just have a Python command-line client, I have coded a simple server that does this. You can

¹ This isn't strictly true. You can google urn:ietf:wg:oauth:2.0:oob to find out more. This would have probably been the right way to code the python client, but the server based approach we have implemented is more likely to be of use to you in real life.

look at the code for this in the **callbackserver.js** and **oauth2token.js** code in the **~/sec_oauth/server** directory.

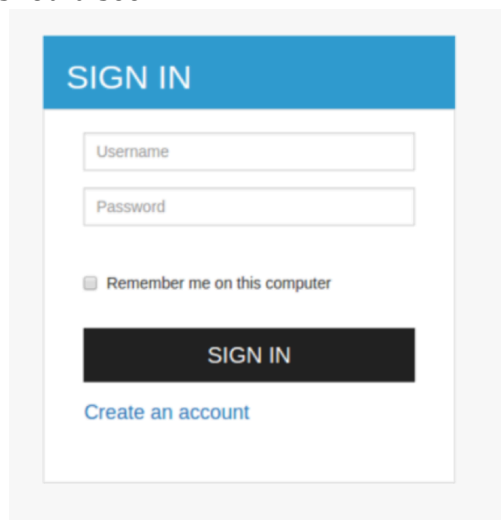
26. Edit the callbackserver.js and replace the existing client_id and client_secret with yours copied from the IS Web Console.

27. Start a new terminal and run the callback server:
cd ~/sec_oauth/server
node callbackserver.js

28. Now let's create a "real" userid instead of admin/admin.

29. Log out of the Admin console and go to
<https://localhost:9443/dashboard>

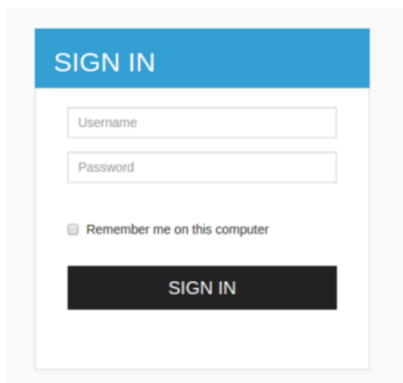
30. You should see:



31. Click Create an account and follow the process.

32. Now browse: <https://localhost:8444/gettoken>

33. You should be redirected to the Auth server to login.

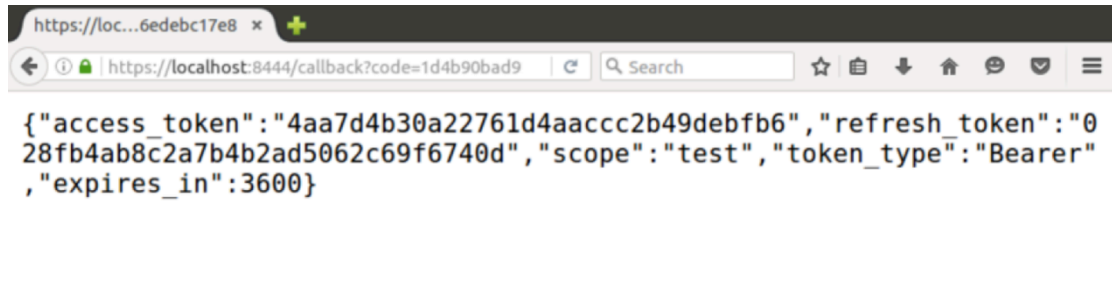


34. Sign in using the ID that you created.

35. You are asked if you want to approve this.

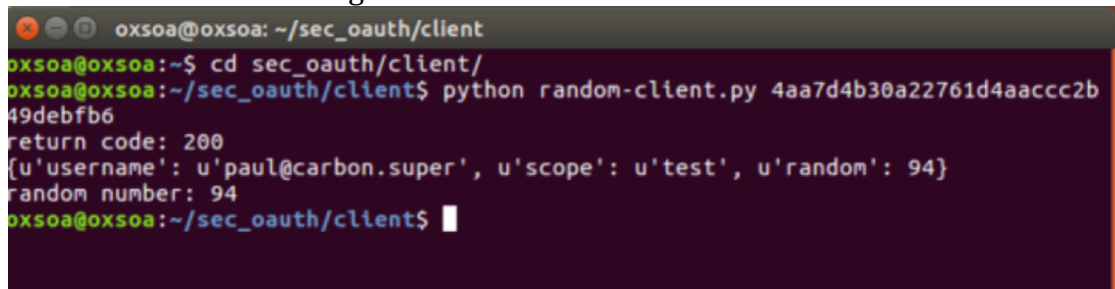
36. Click **Approve**

37. You should see something like:



38. Copy the access token and try your random-client again with this token.

39. You should see something like:



40. Notice that we now have a “real” userid in this response. For example, we could evaluate the scope and the username in our server logic to implement more fine grained logic.

41. Congratulations. The lab is complete!

42. Extension:

Add the refresh client to the python code and code a refresh flow.

Here is the sample refresh flow from the OAuth2 spec:

<https://tools.ietf.org/html/rfc6749#page-47>

```
POST /token HTTP/1.1
Host: server.example.com
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
Content-Type: application/x-www-form-urlencoded
grant_type=refresh_token&refresh_token=tGzv3J0kF0XG5Qx2TlKWIA
```