**Image for the blog background (choose one that fit well):**

* <https://pixabay.com/photos/chain-stainless-steel-metal-iron-4049725/>
* <https://pixabay.com/photos/keys-open-locks-security-unlock-525732/>
* <https://pixabay.com/photos/padlock-lock-chain-key-security-597495/>

**Title:**

How to test an OAuth2/OpenID Connect based system?

**Abstract:**

Delegation of the authorization/authentication via the leveraging of OAuth2/OpenID Connect (OIDC) is more and more common in modern systems but how ensure that no weaknesses were left during the implementation and the configuration or such systems? Let's we show you some key points to verify…

**GitHub repository associated that will be moved to public when blog post will be released:**

<https://github.com/ExcelliumSA/OAuth2OIDC-Study>

**SEO rules indicated by Mathilde:**

* Paragraphs with fewer than 300 words.
* Keyword used as much as possible: *oauth, oidc, security*
* Presence of sections.

**Author(s):** Dominique Righetto

**Note:** In the content below, all figure captions refer to the image file that Mathilde must be when she creates the blog post. So it's normal if the caption do not describe the figure.

# Introduction

This post present a collection of security oriented validation points that should be verified on a system using **OAuth/OpenID Connect** (OpenID Connect will be called **OIDC** in the rest of the post). Therefore, it assumes you are familiar with all the concepts related to OAuth/OIDC. All references to OAuth refer to **OAuth 2.0**.

If it is not the case then you can refer to this free online course named "*Introduction to OAuth 2.0 and OpenID Connect*" [1] kindly created and provided by Dr. Philippe De Ryck [2] or the several tutorials from ConnectId [3].

Note that this post is mainly a security-oriented feedback following a complete focused training that I recently taken on the OAuth/OIDC topic.

# Context

OAuth and OIDC address respectively the Authorization and Authentication aspects of a system. Therefore, any issues in these areas can have critical consequences from a security point of view like authentication or authorization bypass for example.

One of the challenges is that there are several actors involved as well as well as different communications exchanges.

Below is a **simplified example** of OAuth authorization code flow:

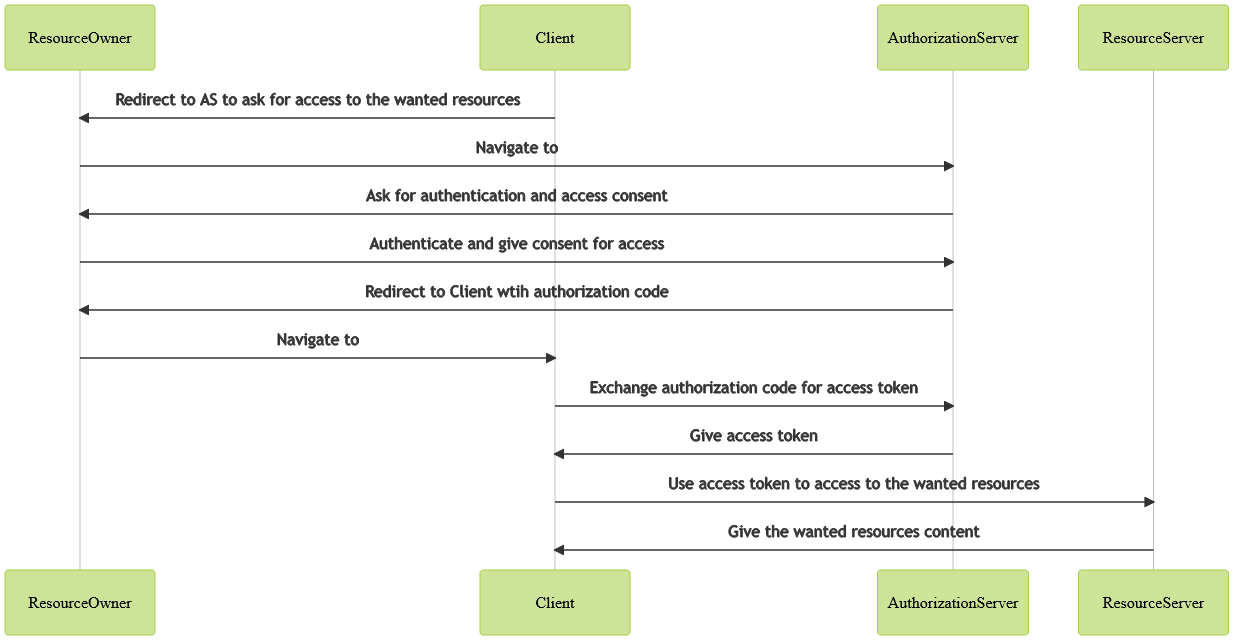


Figure 1: File Figure00.png

This makes the attack surface quite large. To make matters worse, leveraging OAuth/ODIC require to configure an **OpenID Provider** (OIDC) and/or an **Authorization Server** (OAuth). Note that depending on the context, sometimes, you must also provision the entire OpenID Provider/Authorization Server instance. One time again, it's easy to introduce a weakness via insecure settings.

# Find my way in the fog

As I was totally new to the OAuth and OIDC world, I decided to take the course named "*Mastering OAuth 2.0 and OpenID Connect*" [4]. Indeed, OAuth and OIDC are more and more common in modern application architecture and my goal was to understand these new concepts/patterns in order to be able to identity/exploit/prevent security weaknesses.

Following the lessons, I decided to create a list of all pitfalls discovered during the training. The different modules of the course are oriented for developers but I simply converted the "attention points" into "security tests" as well as performing a "mental on-the-fly" penetration test on each feature/flow presented by the instructor to identify potential attack vector/scenario. It is obvious that the list is not exhaustive but it's a good foundation and it will evolve over the time with the growing of my experience in this field.

The list of validation points [5] was organized by actors in order to allow focusing on one actor if the scope of an assessment (code review, configuration review, penetration test, etc.) is targeting only a specific actor. Each validation point has a unique identifier in order to allow referencing it in document, script, report and so on.

A table [6] is provided to indicate if a validation point is manual or can be automated. The automation status is based on the technical capabilities to create code that perform the target test, without human interaction, and give a reliable result with the same level of trust that it was performed manually. Once again this "automation status" can be not accurate for you in you know how to automate it ☺

# Overview of the validation points

In addition to a representation using a "list" approach, a mind map was created in order to provide a higher overview of the collection of validation points.

Below is overview of the counter of tests identified (a validation point 🡨🡪 a test):

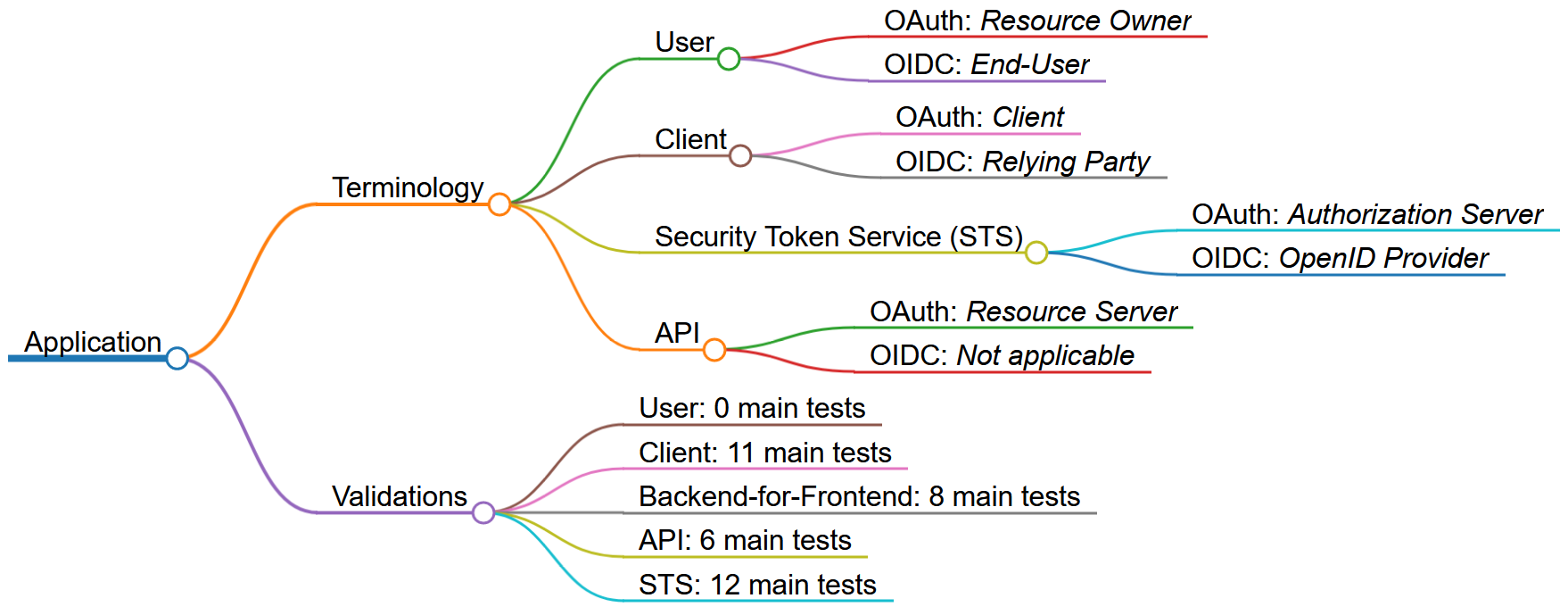


Figure 2: File Figure01.png

A total of 37 main tests were identified. The notion of "main tests" refer to the fact the some test contains "sub tests" but here, for simplicity, only the main tests were included in the count.

Example of main tests (identifier STS04):

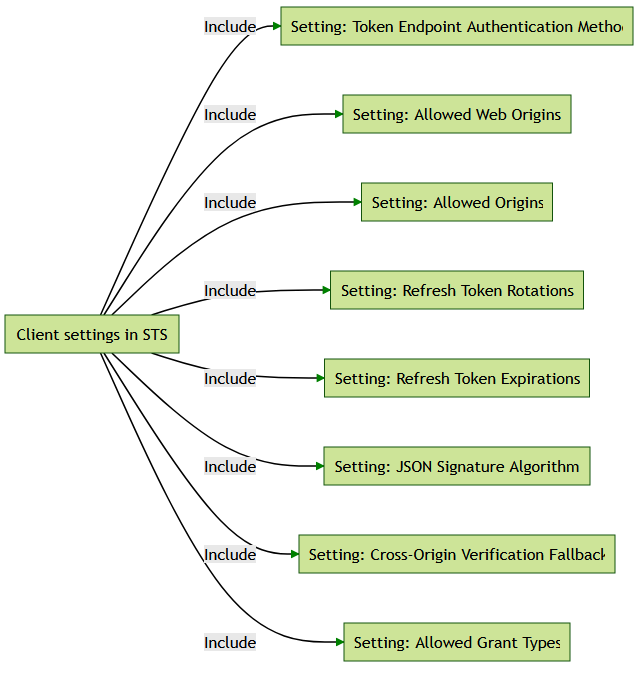


Figure 3: File Figure02.png

The detailed version of mind map is available on the GitHub repository of the blog post [7].

# How to apply control?

In this section, I use a local lab based on Keycloak [8], to show how to perform some of the validation points from the list. A demo configuration is provided to allow you to reproduce the test performed [9].

## CLT01: For SPA, ensure that it uses the Authorization Code Flow with PKCE instead of the Implicit flow

In the demo app, when the login is used, the following request is sent to the "/auth" endpoint:

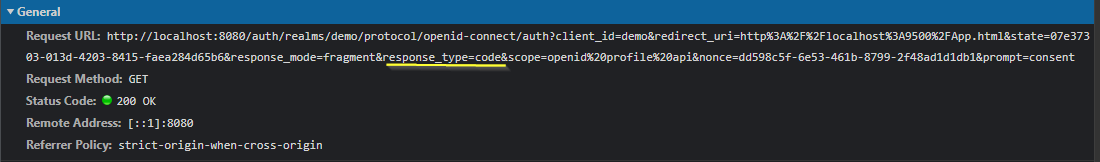


Figure 4: File Figure03.png

The parameter "response\_type" is set to "code". However, there is no parameter named "code\_challenge" so the flow used here is "Authorization Code" and not "Authorization Code with PKCE".

## STS04h: Allowed Grant Types (flow types enabled) and Response Modes for a client should be limited to one used

As seen in the previous test, the "Authorization Code" flow is used, but does the "Implicit" flow it is enabled?

Let's try to start such one…

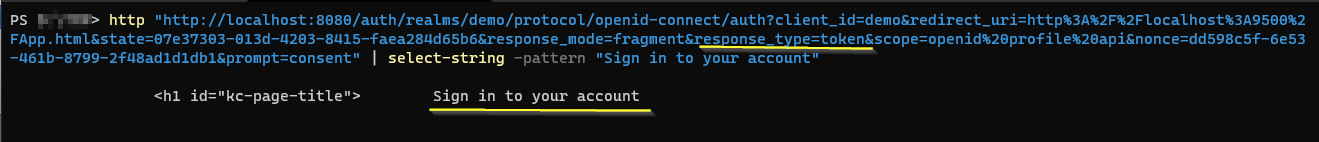


Figure 5: File Figure04.png

Login form is proposed so the flow is allowed. When it is not the case, the following error is received:

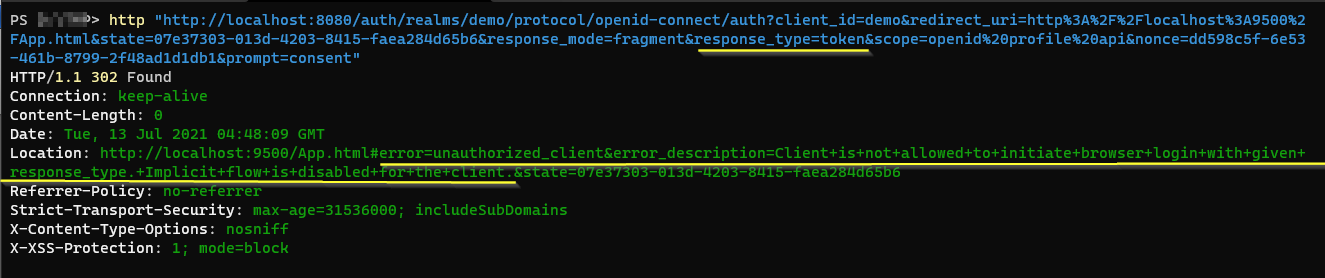


Figure 5: File Figure05.png

## STS12: Ensure that the STS reject any request (flow init) specifying a scope that is not defined for the targeted API and prevent scope enumeration/discovery operation

In the web client app, the following scope are used:

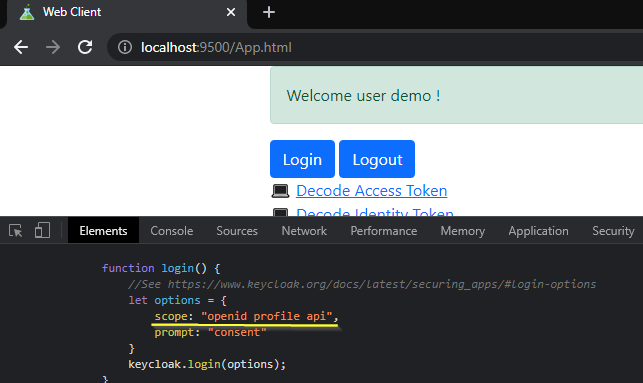


Figure 5: File Figure06.png

Keycloak allow defining optional scope:

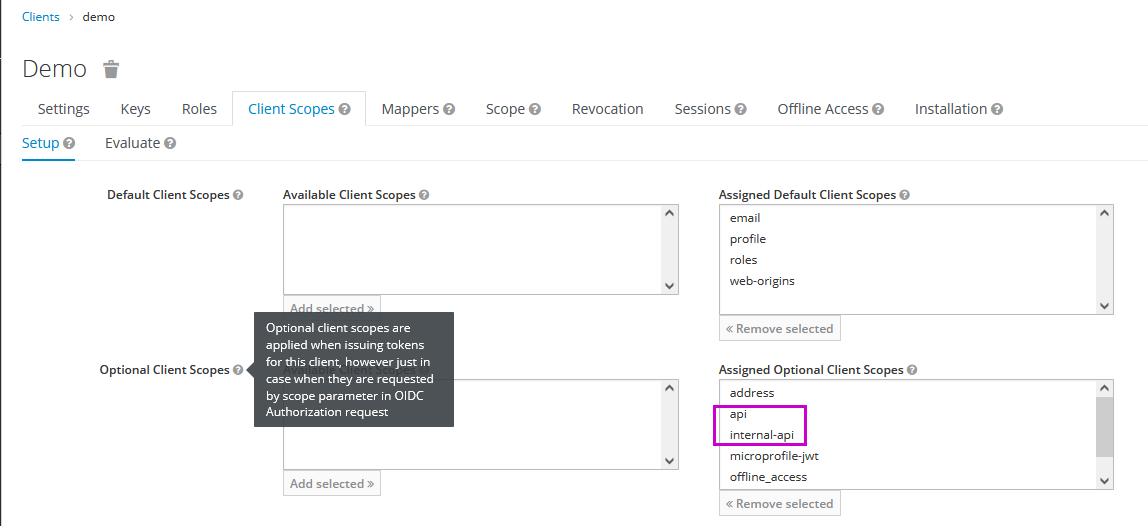


Figure 5: File Figure07.png

When a flow is started with an invalid scope, the following error is received:

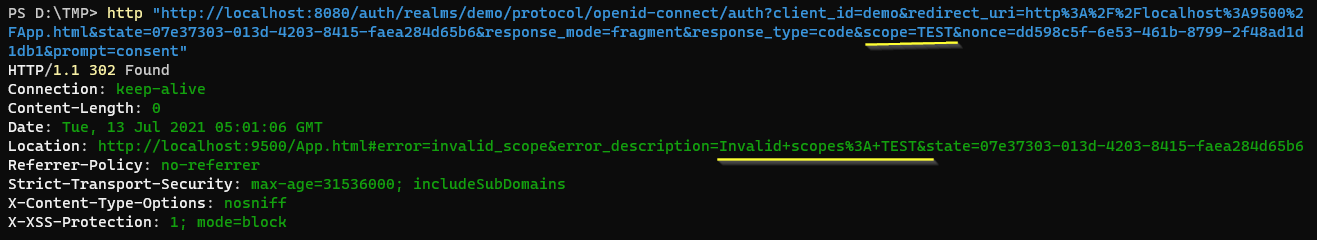


Figure 5: File Figure08.png

When a scope is valid then the login form is received with an HTTP 200.

Based on this discrepancy factor it's possible to try the following scope enumeration using FFUF [10] with a dictionary of English word [11]:

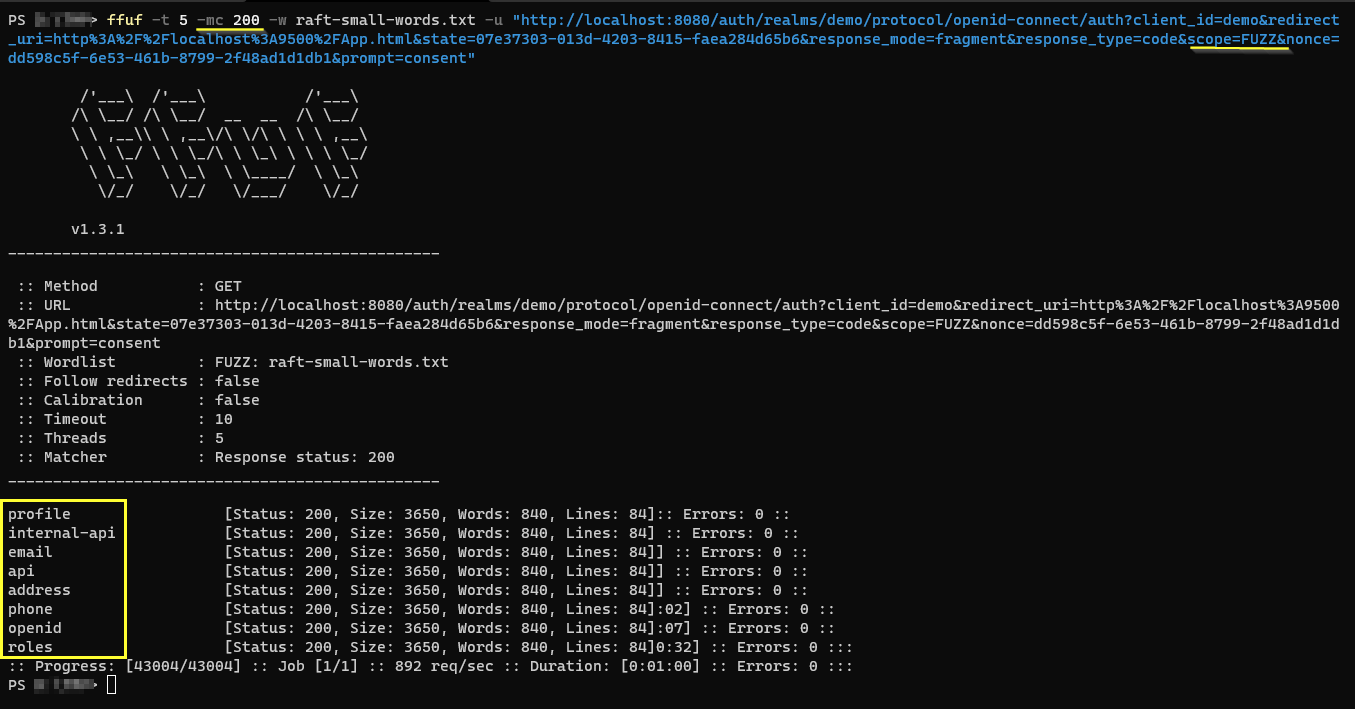


Figure 5: File Figure09.png

8 scopes, not present in the web client, were identified.

# Conclusion

Xxx

# References

1. <https://courses.pragmaticwebsecurity.com/courses/introduction-to-oauth-2-0-and-openid-connect>
2. <https://pragmaticwebsecurity.com/about.html>
3. <https://connect2id.com/learn>
4. <https://courses.pragmaticwebsecurity.com/bundles/mastering-oauth-oidc>
5. <https://github.com/ExcelliumSA/OAuth2OIDC-Study/blob/main/OAauth2_OIDC_Security_Validations.md>
6. <https://github.com/ExcelliumSA/OAuth2OIDC-Study/blob/main/OAauth2_OIDC_Security_Validations.md#validation-automation-status>
7. <https://github.com/ExcelliumSA/OAuth2OIDC-Study/blob/main/OAauth2_OIDC_Security_Validations.png>
8. <https://www.keycloak.org/getting-started/getting-started-docker>
9. <https://github.com/ExcelliumSA/OAuth2OIDC-Study#lab>
10. <https://github.com/ffuf/ffuf>
11. <https://raw.githubusercontent.com/danielmiessler/SecLists/master/Discovery/Web-Content/raft-small-words.txt>