How to evaluate an "OAuth/OpenID Connect" system from a security point of view?

October 22, 2021

Share

This post presents 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"* kindly created and provided by Dr. Philippe De Ryck or the several tutorials from ConnectId.

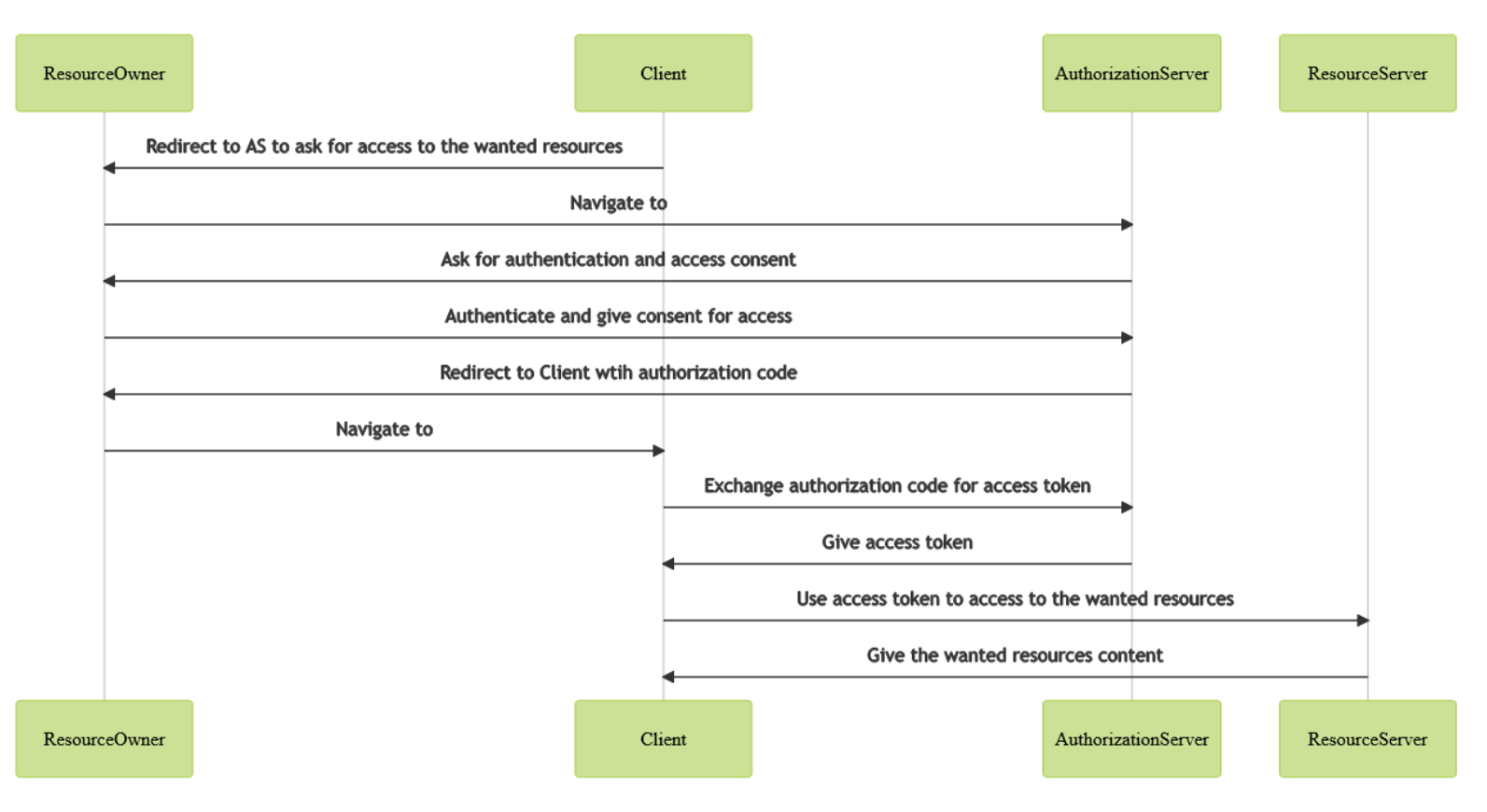
Note that this post is mainly security-oriented feedback following a complete focused training that I have recently taken on the OAuth/OIDC topics.

The importance of OAuth & OIDC

OAuth and OIDC address respectively the Authorization and Authentication aspects. 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 faced is that there are several actors involved as well as different communications exchanges.

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



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

Getting familiar with OAuth & OIDC

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"*. Indeed, OAuth and OIDC are more and more common in modern application architecture and my objective was to understand these new concepts and patterns to be able to identify, exploit and prevent security weaknesses.

After 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, in my head, a penetration test on each feature or flow presented by the instructor to identify potential attack vectors and scenarios. It is obvious that the list is not exhaustive, but it is a good foundation, and it will evolve over time with the growth of my experience in this field.

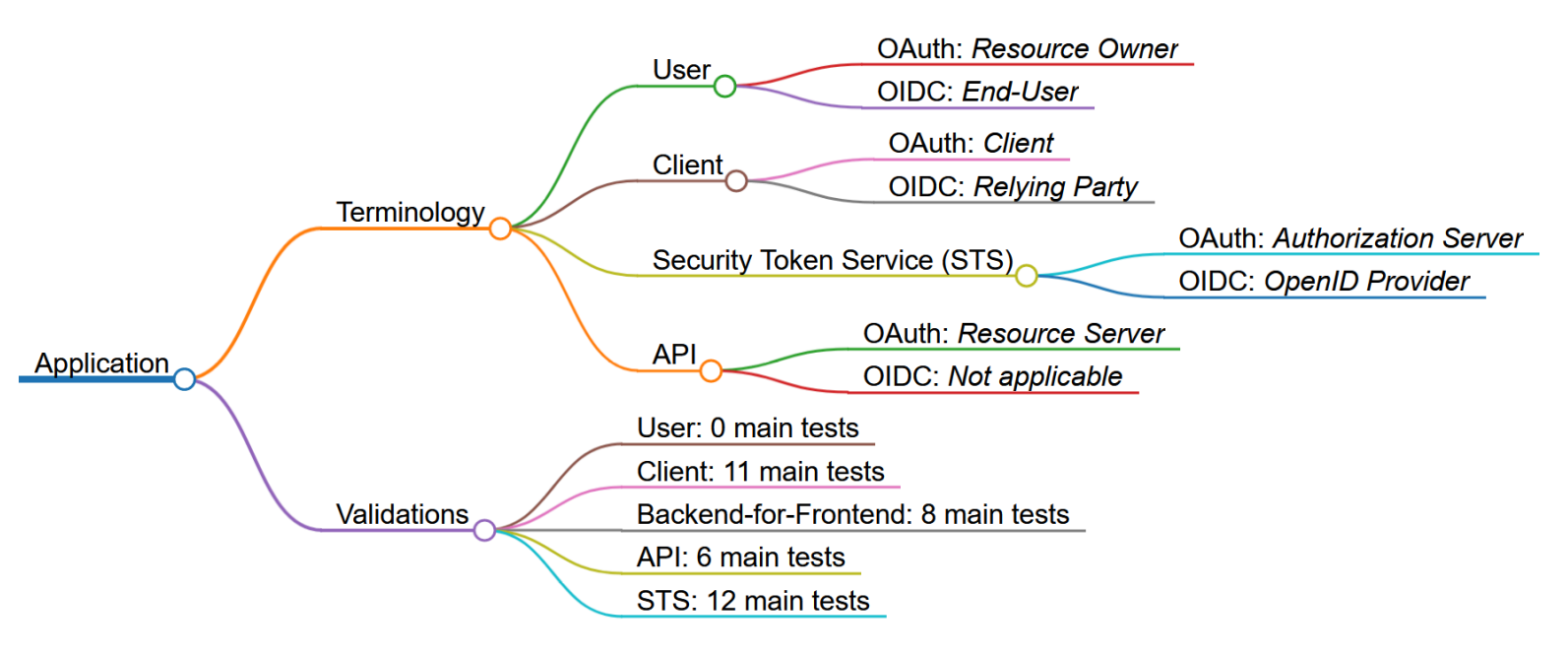
The list of validation points was organized by actors in order to allow focusing on one actor if the scope of an assessment (code review, configuration review, penetration test, etc.) is targeting only a specific actor. Each validation point has a unique identifier in order to allow referencing it in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

A table is provided to indicate if a validation point is manual or automated. The automation status is based on the technical capabilities to create code that performs the target test, without human interaction, while giving a reliable result with the same level of trust as if it was performed manually. Once again, this "automation status" can be inaccurate for you if you know how to automate it 😊.

Overview of the validation points for OAuth & OIDC

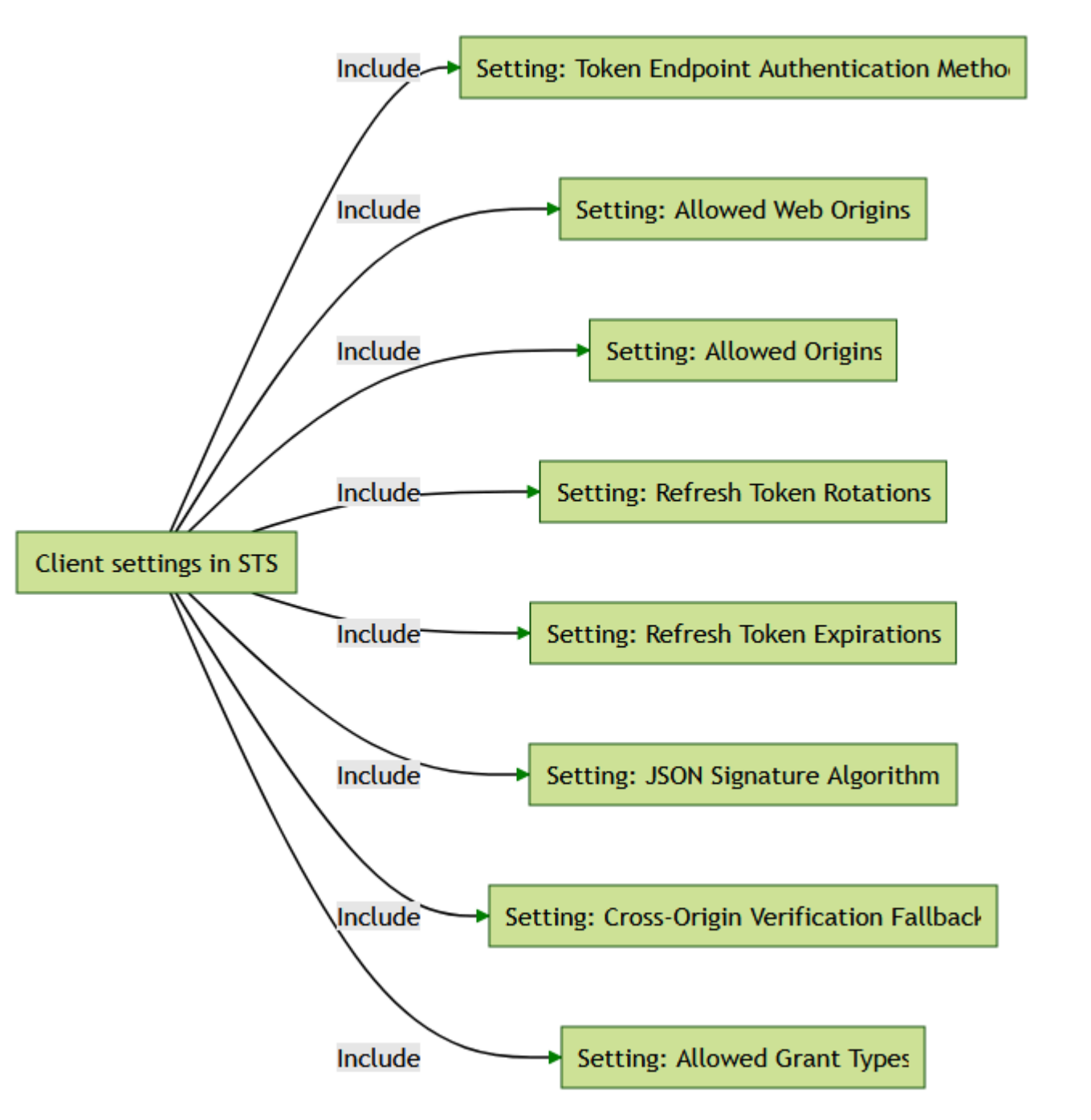
In addition to a representation using a "list" approach, a mind map was created in order to provide a high-level overview of the collection of validation points.

Below is the overview of the number of tests identified (a validation point refers to a test):



A total of 37 main tests were identified. The notion of "main tests" refers to the fact that some tests contain "sub-tests" but here, for simplicity, only the main tests were included.

Example of main tests (identifier STS04):



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

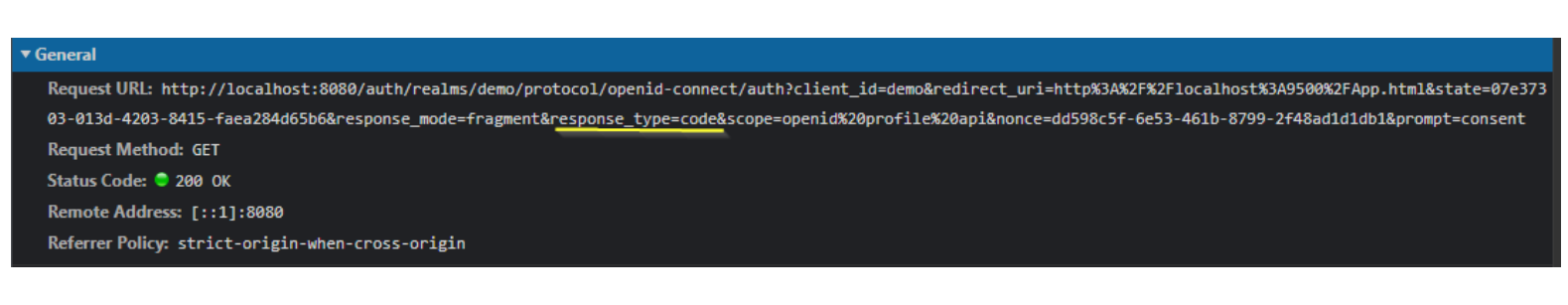
How to apply control on the different OAuth & OIDC areas

In this section, I used a local lab based on Keycloak 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.

Please, feel free to surf here to access the following photos in case of poor quality.

For SPA, ensure that it uses the Authorization Code Flow with PKCE instead of the "Authorization Code" basic flow (reference CLT01)

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



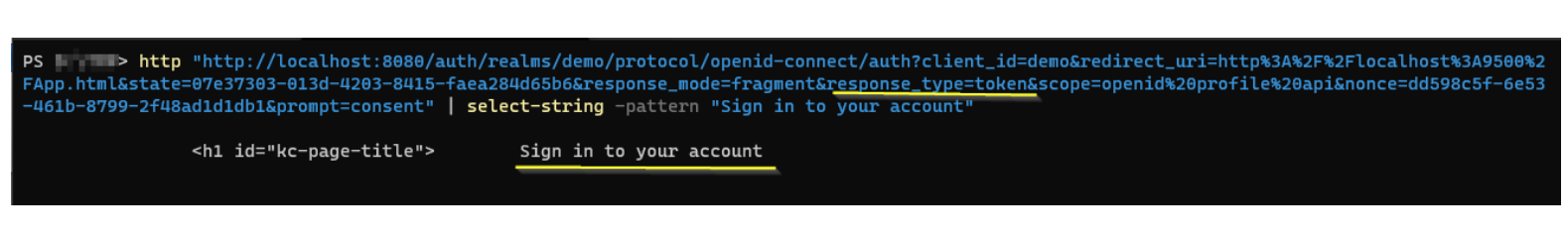
The parameter "response_type" is set to "code". However, there is no parameter named "code_challenge" so the flow used here is "Authorization Code" and not "Authorization Code with PKCE".

⚠ **Risk:** It is possible to start a flow with an insecure mode implying that, if the authorization code is intercepted by an attacker then, it can be used to obtain an access token before the Client uses it (an authorization code is valid once).

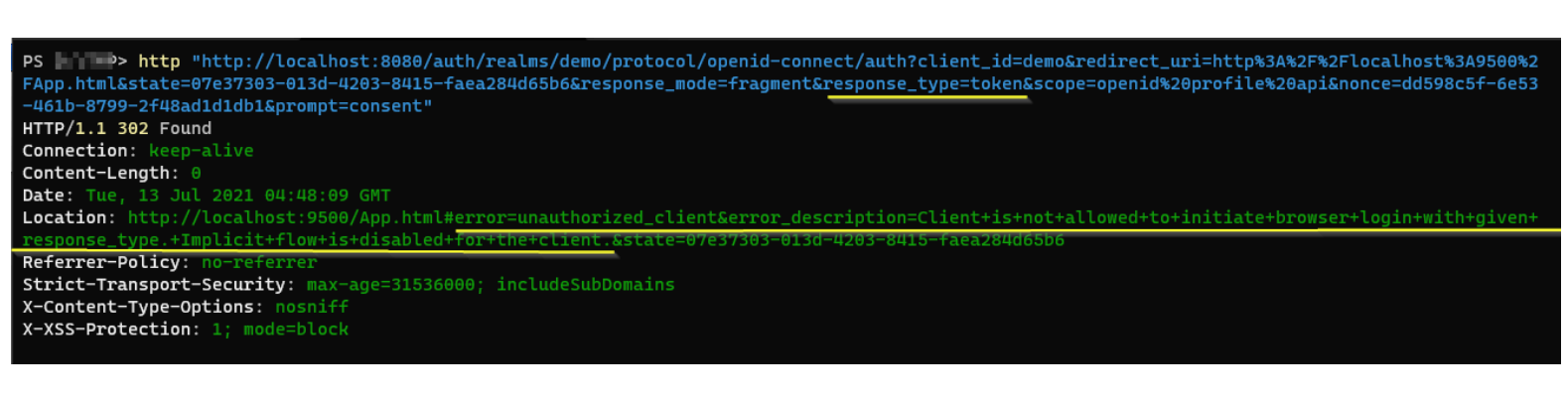
Allowed Grant Types (flow types enabled) and Response Modes for a client should be limited to the needed ones (reference STS04h)

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

Let's try to start such a one…



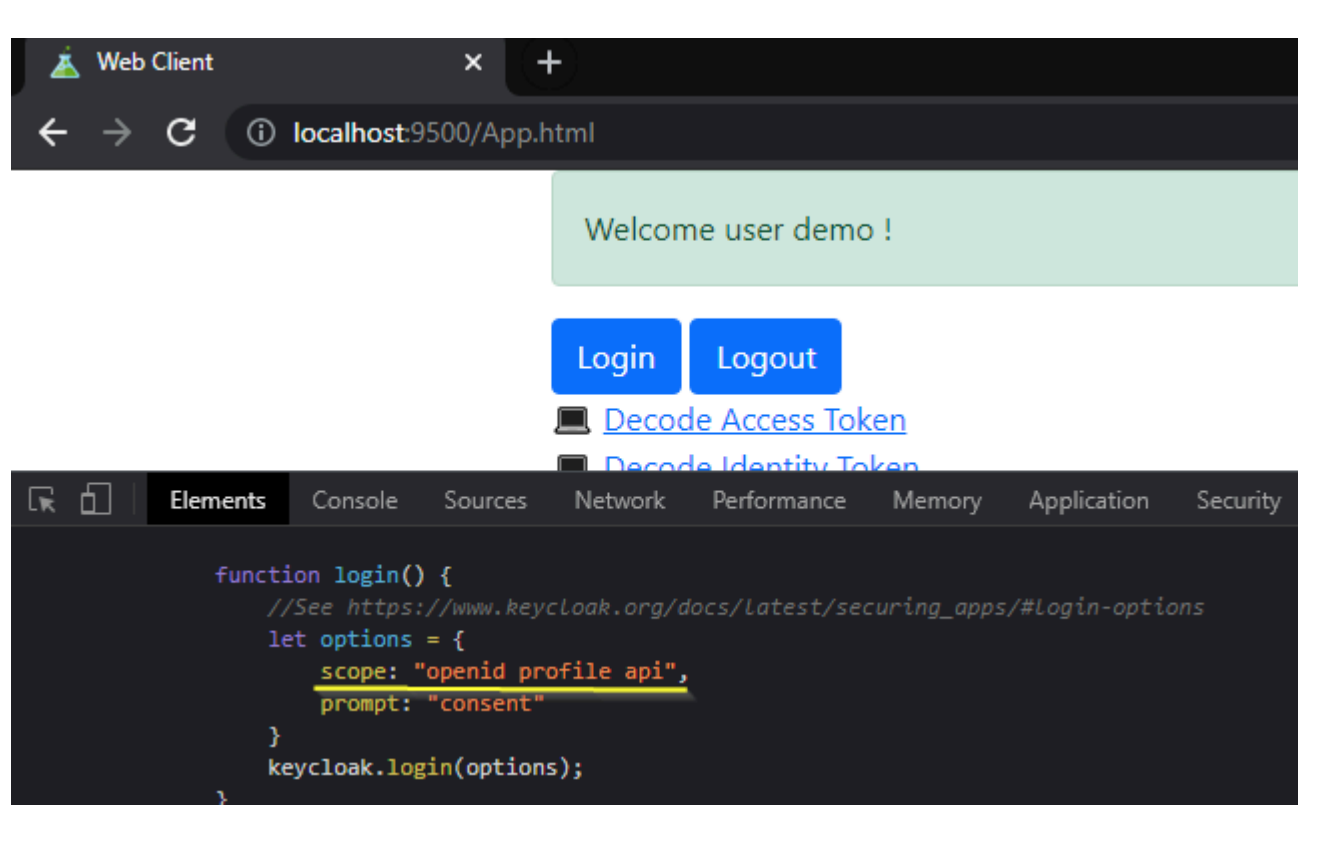
A login form is proposed so the flow is allowed. When it is not the case, the following error is received:



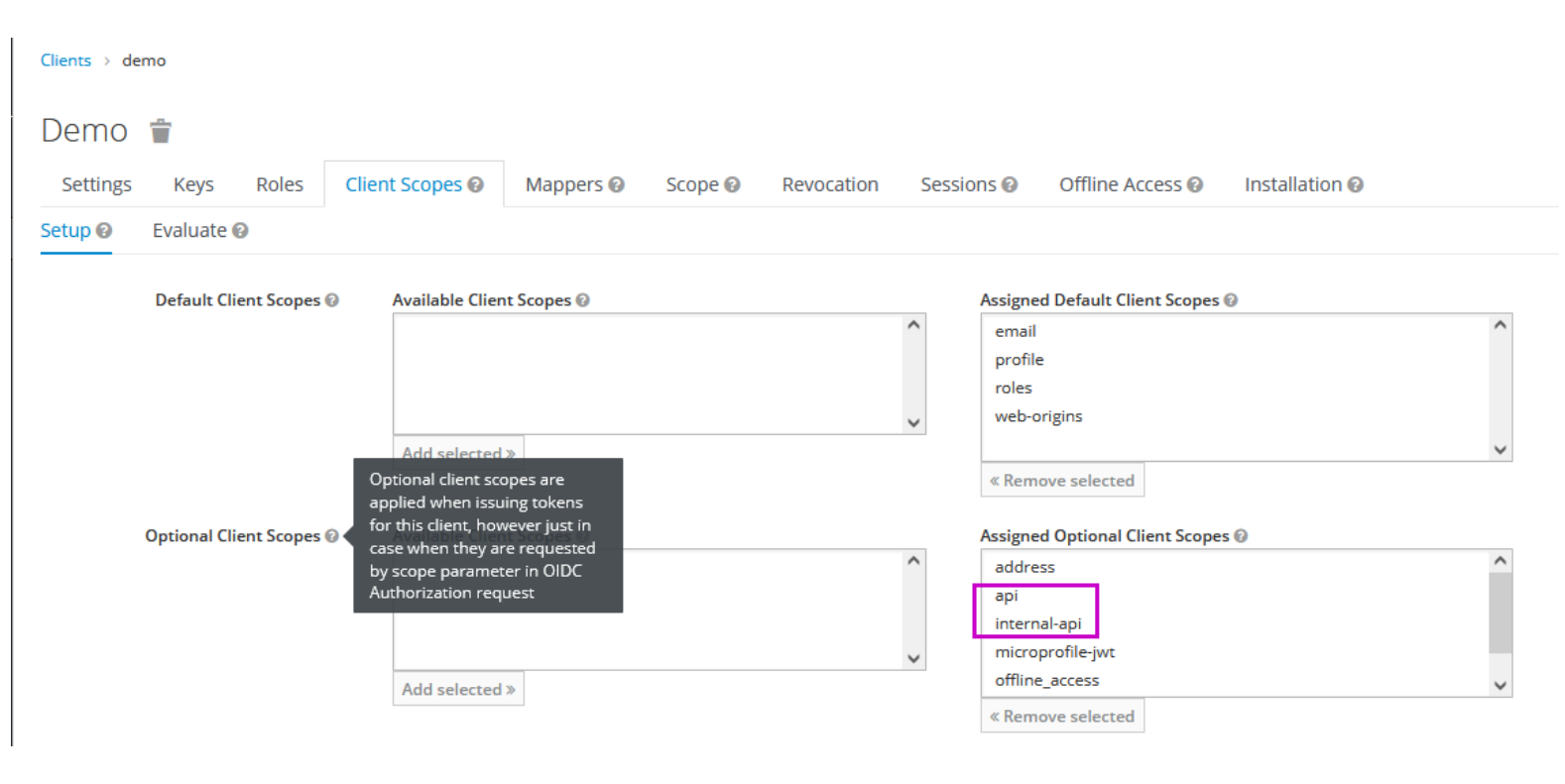
⚠ **Risk:** It is possible to start a flow using the deprecated "Implicit" mode.

Ensure that the STS rejects any request specifying a scope that is not defined for the targeted API and prevent scope enumeration/discovery operation (reference STS12)

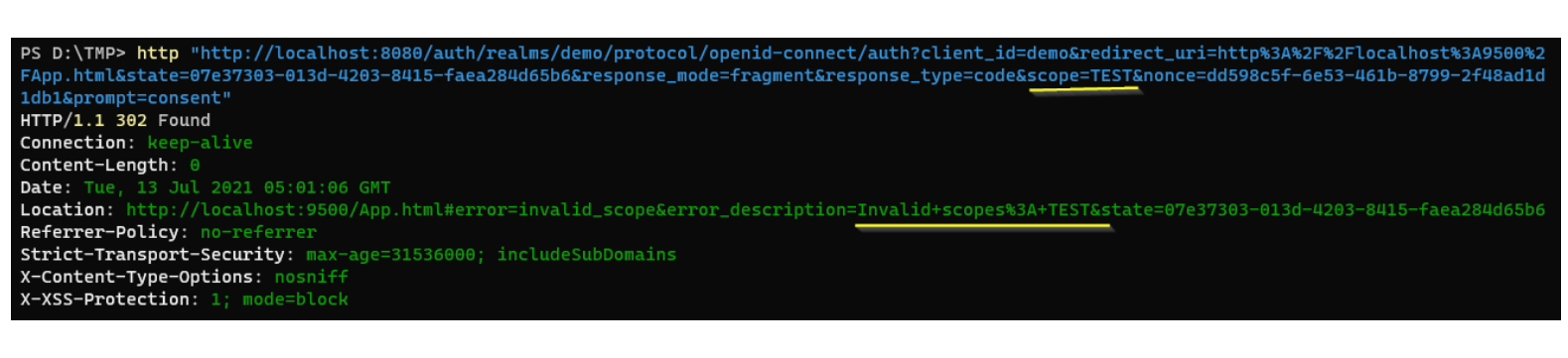
In the web client app, the following scopes are used:



Keycloak allows defining optional scopes:

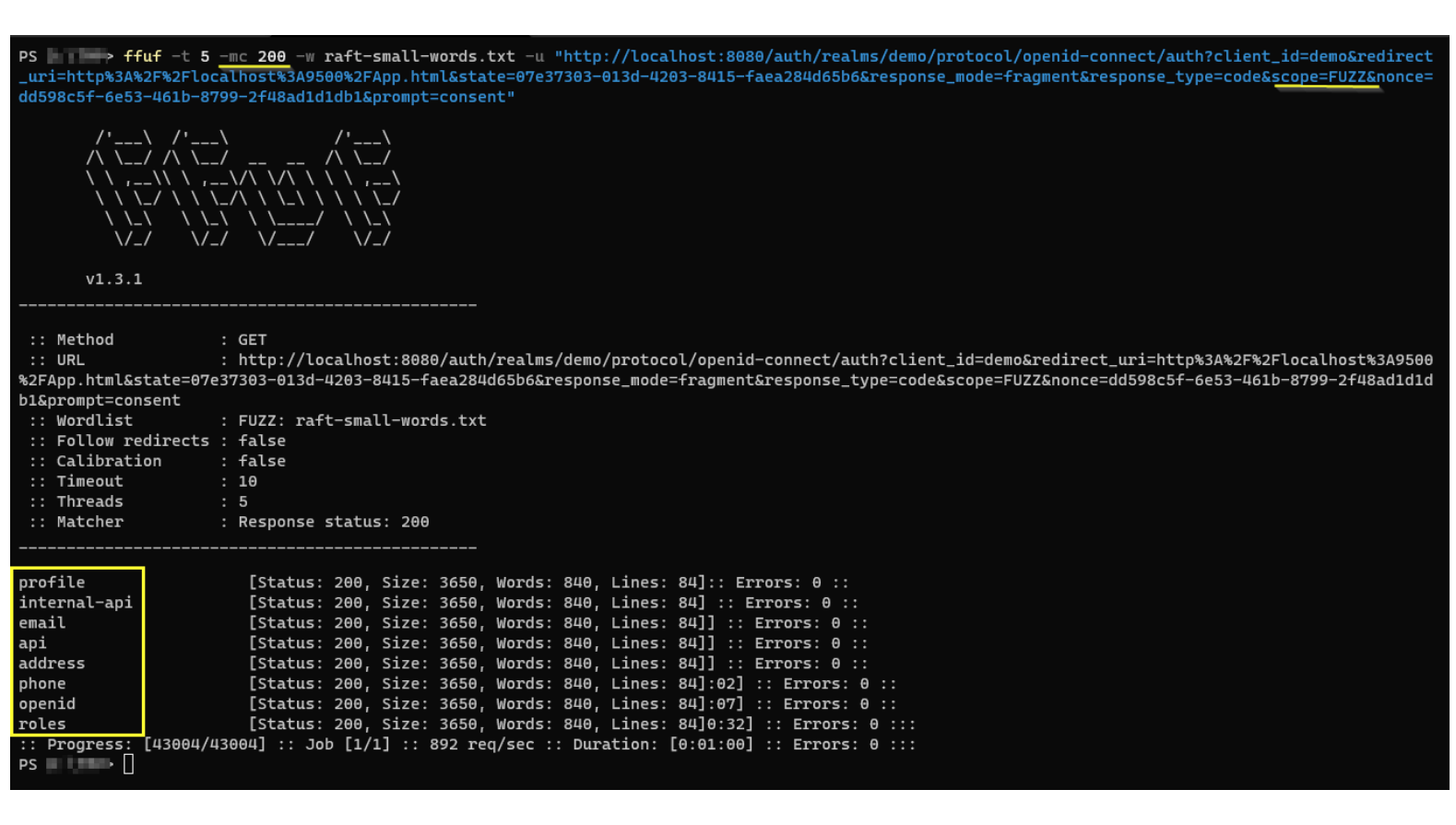


When a flow is started with an invalid scope, the following error is received:



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 with a dictionary of English words:

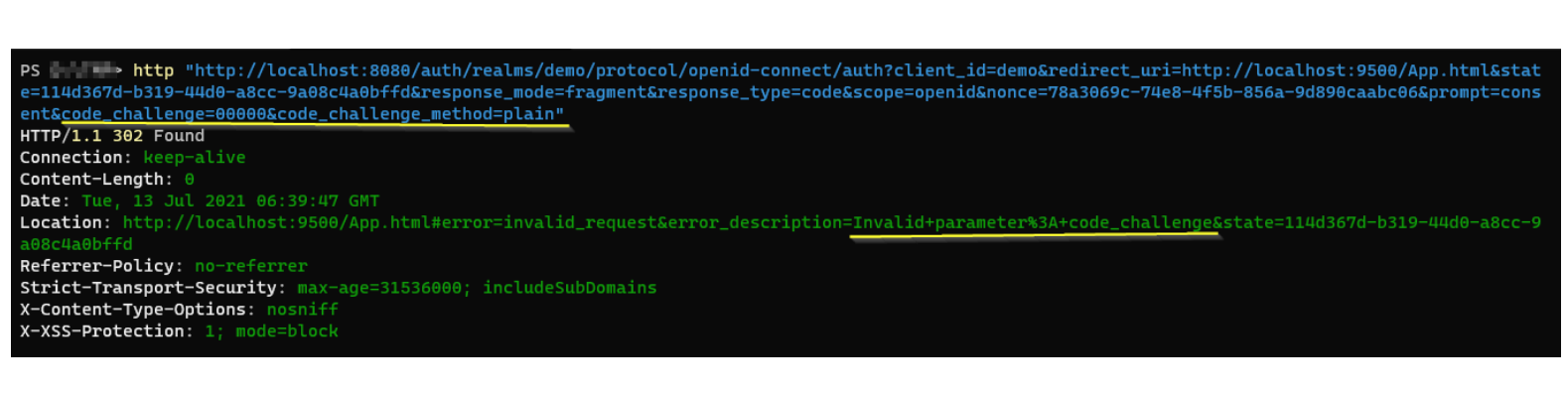


8 scopes not present in the Client code were identified.

⚠ **Risk:** It is potentially possible for a Client to access more resources if the User accepts the new scope requested.

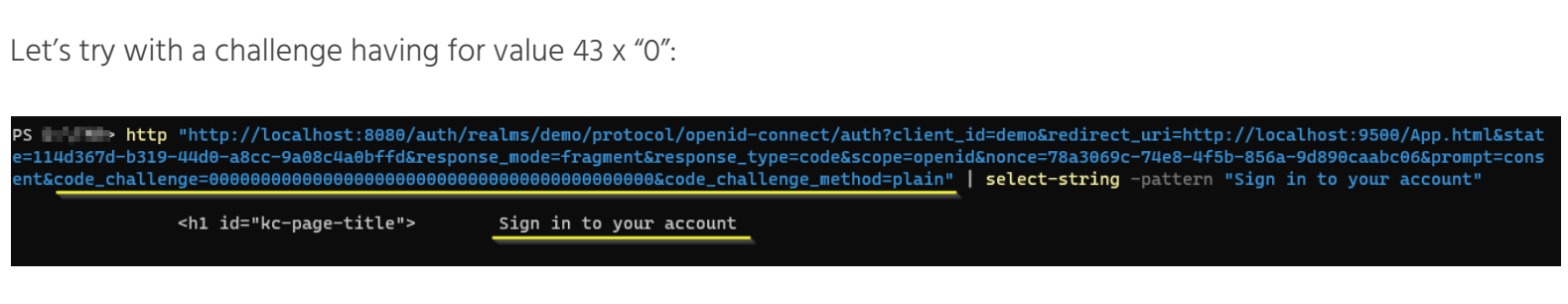
The STS does not support broken hashing algorithms like MD5 or SHA1 or even "plain" (reference STS00b)

The code verifier must have a minimum length of 43 positions according to the RFC , so, let's try to start an "Authorization Code with PKCE" flow with a "plain" code challenge algorithm (code_challenge = code_verifier) and a weak code verifier:



The "plain" algorithm is accepted but "code_challenge" is rejected.

Let's try with a challenge having for value 43 x "0":



Now it is accepted and the flow is started.

⚠ **Risk:** It is possible to start a flow that disables protection added by PKCE. It causes the value of the code verifier to be disclosed during the start of the flow.

Conclusion of my journey with OAuth and OIDC

OAuth 2.0 and OpenID Connect (OIDC) allow centralizing authorization and authentication management. On one side of the coin, it decreases the attack surface of the application by removing the need to implement some error-prone features like authentication and account management. However, on the other side, these new mechanisms are difficult to master and it is easy to introduce a weakness during the setup of the authorization/authentication flows.

Anyways, these mechanisms are a true added value from a security point of view and, like any system, it is just necessary to ensure that every component in the flow uses recommended and secure settings. It's the main reason why the checklist was created, in order to ensure during a security assessment to review a maximum of aspects. I hope that this checklist will be useful, for the defender side as well, in order to allow them to review and monitor the configuration of the involved parties.

To go further on the offensive side, the training module dedicated to OAuth 2.0, from the PortSwigger Web Security Academy, provides additional insights about interesting attack vectors

Did you like the article? Find even more articles written by the AppSec team here.

Credits

- Dominique Righetto
- Elliot Rasch
- Julien Ehrhart

Uncategorized

Recent Posts
The Necessity of Cyber Crisis Exercises
The Art of Password Spraying: A Comprehensive Analysis
Common client-side vulnerabilities of web applications

Categories
Advice
Consulting
Cyber Blog Post
General
Newsletter
PR
The Cyber Blog Times
Uncategorized

Search
Search …