Ανάπτυξη εργαλείου εύρεσης ευπαθειών στο πρωτόκολλο **OAuth 2.0**

Η Μεταπτυχιακή Διατριβή κατατέθηκε στο Τμήμα Μηχανικών Πληροφοριακών και Επικοινωνιακών Συστημάτων της Πολυτεχνικής Σχολής του Πανεπιστημίου Αιγαίου σε μερική εκπλήρωση των απαιτήσεων για το Μεταπτυχιακό Δίπλωμα ειδίκευσης στην Ασφάλεια Πληροφοριακών και Επικοινωνιακών Συστημάτων

Καράμπαλης Ευάγγελος

Επιτροπή
Επιβλέπων: Καθηγητής Γεώργιος Καμπουράκης
Αναπληρωτής Καθηγητής Αλέξιος Καπόρης
Επίκουρος Καθηγητής Δημήτριος Σκούτας

Τμήμα Μηχανικών Πληροφοριακών & Επικοινωνιακών Συστημάτων
Πανεπιστήμιο Αιγαίου
Οκτώβριος 2022

Development of a vulnerability scanning tool for OAuth 2.0

A dissertation submitted to the Department of Information & Communication Systems Engineering, School of Engineering of the University of The Aegean in partial fulfilment of the requirements for the degree of Master of Science in Information and Communication Systems Security

Karabalis Evangelos

Committee
Supervisor: Professor Georgios Kambourakis
Associate Professor Alexios Kaporis
Assistant Professor Dimitrios Skoutas

Department of Information and Communications System
Engineering University of Aegean
October 2022

Δήλωση Αυθεντικότητας

Βεβαιώνω ότι είμαι συγγραφέας αυτής της διπλωματικής εργασίας και ότι κάθε βο- ήθεια την οποία είχα για την προετοιμασία της, είναι πλήρως αναγνωρισμένη και ανα- φέρεται στην εργασία. Επίσης έχω αναφέρει τις πηγές από τις οποίες έκανα χρήση δε- δομένων, ιδεών ή λέξεων είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Τέλος, βεβαιώνω ότι αυτή η διπλωματική εργασία προετοιμάστηκε από εμένα προσωπικά ειδικά για τις απαιτήσεις του μεταπτυχιακού προγράμματος σπουδών του Τμήματος Μηχανι- κών Πληροφοριακών και Επικοινωνιακών Συστημάτων του Πανεπιστημίου Αιγαίου στη Σάμο.

Καρλόβασι, 5 Οκτωβρίου 2022

Όνομα Επώνυμο

(Υπογραφή)

Statement of Authenticity

I declare that this Master's thesis is my own work and was written without literature other than the sources indicated in the bibliography. Information used from the published or unpublished work of others has been acknowledged in the text and has been explicitly referred to in the given list of references. This Master's thesis has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education.

Karlovasi, 5th October 2022

Student's name

(Signature)

Περίληψη

Στις μέρες μας, οι σύγχρονες εφαρμογές στο διαδίκτυο είναι συνδεδεμένες με διαφορετικές διεπαφές προγραμματισμού εφαρμογών προκειμένου να χρησιμοποιήσουν διάφορους πόρους που ανήκουν σε ένα χρήστη. Το γεγονός αυτό εγείρει ζητήματα εξουσιοδότησης σχετικά με τον τρόπο τον οποίο οι προγραμματιστικές διεπαφές μπορούν να αλληλεπιδράσουν έχοντας κατάλληλη εξουσιοδότηση. Στο πρόσφατο παρελθόν, ο τρόπος με τον οποίο γινόταν αυτή η ανταλλαγή της εξουσιοδότησης ήταν μέσω ορισμένων – αμφίβολης αποτελεσματικότητας τρόπων – διαμοιρασμού κωδικών που εξέθεταν τα διαπιστευτήρια ενός χρήστη στην εκάστοτε εφαρμογή που ήταν υλοποιημένη με τέτοιες συνδέσεις. Καθώς το ζήτημα αναγνωρίστηκε ως ένα σημαντικό ρίσκο για την ασφάλεια, ξεκίνησε να εισάγεται η θεμελίωση ενός προτύπου εξουσιοδότησης προκειμένου να εμποδιστούν τέτοιου είδους ρίσκα και να αποφευχθούν μια πληθώρα προσαρμοσμένων λύσεων που είχαν και ζητήματα ασφαλείας και παρουσίαζαν ασυνέπειες στις υλοποιήσεις τους. Η θεμελίωση αυτή εκφράστηκε με το πρότυπο OAuth 2.0 το οποίο υλοποιεί ένα πρωτόκολο εξουσιοδότησης που είναι σχεδιασμένο να παρέχει εξουσιοδότηση μεταξύ των εμπλεκομένων συστημάτων.

Με τα παραπάνω ως κίνητρο, η συγκεκριμένη διπλωματική μεταπτυχιακή εργασία επικεντρώνεται στη μελέτη της των διάφορων επιθέσεων που θα μπορούσαν να ανακύψουν στο OAuth Code Flow καθώς και στη σχεδίαση και ανάπτυξη ενός εργαλείου λογισμικού που είναι ικανό να ανιχνεύσει εσφαλμένενες διαμορφώσεις σε διακομιστές OAuth 2.0. Τέτοιες διαμορφώσεις θα μπορούσαν να οδηγήσουν σε πληθώρα ευπαθειών, επομένως το προτεινόμενο πλαίσιο μπορεί να χρησιμοποιηθεί για την ενίσχυση της ασφάλειας των διακομιστών OAuth.

Καράμπαλης Ευάγγελος
Τμήμα Μηχανικών Πληροφοριακών Επικοινωνιακών Συστημάτων
Πανεπιστήμιο Αιγαίου

Abstract

Nowadays, modern web applications are connected to different APIs in order to use different resources that the user has in their possession. This raises authorization issues about how the APIs are able to interact with proper authorization. A few years ago, the only way to do that was through different password-sharing anti-patterns that inevitably exposed user credentials to any client application that was implementing such connections. When the issue was recognized as a serious security risk, the establishment of such a delegation framework started in order to prevent such security risks and also to avoid a huge variety of custom solutions that also were facing both security issues and inconsistencies in their implementations. This establishment was the OAuth 2.0 protocol which stands as a delegation protocol that is designed to provide authorization across systems.

Motivated by this fact, this Master thesis focuses on the study of the diversity of the attacks that could be faced in OAuth Code Flow and also on the design and development of a framework which would be able to detect misconfigurations in OAuth servers. Such misconfigurations could potentially lead to a plethora of vulnerabilities, therefore such a framework can be used towards enhancing the security of OAuth servers.

Karabalis Evangelos
Department of Information and Communication Systems Engineering
University of the Aegean

Contents

List of Figures

List of Tables

LIST OF ACRONYMS

API Application Programming Interface

HTML HyperText Markup Language

CIA Confidentiality Integrity Availability

OAuth Open Authorization

RFC Request For Comments

SSO Single Sign On

PKCE Proof Key for Code Exchange

URL Uniform Resource Locator

URI Uniform Resource Identifier

CSRF Cross Site Request Forgery

C Client

A/AS Authorization Server

SHA Secure Hashing Algorithm

HTTP Hypertext Transfer Protocol

TLD Top Level Domain

NPM Node Package Manager

AWS Amazon Web Services

JSON JavaScript Object Notation

JWKS JSON Web Key Set

DoS Denial of Service

JWT JSON Web Token

RSA Rivest, Shamir, Adleman - the creators of the RSA algorithm

CLI command-line interface

XHR XMLHttpRequest

XML Extensible Markup Language

OIDC OpenID Connect

1 Introduction

1.1 Background and Context

In recent years, the use of dynamic pages that are related to application programming interface (API) is increasing. For instance, from 2014 to 2018 the API-related traffic increased from 47 to 83 percent and HTML related traffic (meaning from static resources) decreased from 53 to 17 percent [3].

In this context, the main factors that should be taken into account in the risk assessment and composition of the API attack surface are the following:

1. Exploitability – The available tools, the user interaction, the repeatability and the privileges that are required to exploit the target; for example there may be a need to elevate rights to compromise a target.

2. Dominance – The lack of awareness of the implementer, the complicated concepts, the immaturity of tools, and the lack of time.

3. Detectability – The existing tools and the number of the false positive alarms

4. Impact – This is broken down to the confidentiality, integrity and availability (CIA) triad.

Among other security issues that could be spotted in APIs, one of the most crucial is the lack of or the improper authorization. Such issues enforced the use of password-sharing anti-patterns that expose user credentials and caused several other disabilities in a proper authorization process.

The main reason that OAuth exists is the fact that in the classic authentication model, the user's account credentials are generally shared with the third party website, which results in several problems; these are well documented in the OAuth 2.0 RFC 6749.

1. The third party can save the credentials in plaintext.

2. The third party acquires a large amount of access to users' data, typically full account access.

3. There is no proper method to revoke access given to a third party without revoking all other third parties because the credentials are common to all third parties.

4. If any third party is compromised, it will result in compromise of the credentials of the end user.

For dealing with such issues, it has been decided to adopt a common strategy and the solution came with the introduction of OAuth 2.0. As it is well-known, this protocol is an authorization framework for web applications. It permits selective access to a user's resources without disclosing the password to the website which asks for the resource. Nowadays, it is one of the most common and widely used authorization frameworks in the industry and has already become a standard. The OAuth 2.0 Authorization Framework has been adopted by Google, Microsoft, Facebook, Instagram, GitHub, Meetup and many other popular technology platforms. The OAuth framework supports scope-restricted access to client data, without requiring the client to share its credentials with the third party applications. For example, LinkedIn may require access to the contact list of a user account registered with Facebook to suggest people to connect with. In this case, the Facebook application, holding users' data, wishes to delegate the access of limited user data to a third party application.

As it has already been mentioned, and in order to explain it thoroughly, OAuth provides a way in which user credentials are not shared with any of the third party applications, but only one-time tokens are shared, which are good enough for temporary access and for a well-defined scope. The OAuth specification allows a person to control the access to protected resources available on one application from another application. Therefore, the users control exactly what to share with another application. OAuth 2.0 is primarily an authorization protocol and does not deal with how the resource owner authentication can be done with any of the existing ways such as password or more sophisticated ways like single sign-on (SSO).

Since OAuth is an evolutionary technology, it consists of a great variety of specifications related to enhanced countermeasures to prevent more sophisticated attacks (e.g., PKCE) from happening, to support authentication mechanism that could stand on top of the protocol itself (e.g., OpenID) or to address the way for all of the implemented solutions to become more flexible in interoperability related issues (e.g., servers' metadata). A thorough list of the related RFCs contains more than 35 of them.

1.2 Scope and Objectives

The scope of the thesis research was focused on creating a software scanner that could detect misconfigurations in the OAuth Code Grant protocol for web applications that can keep secrets (three tier web applications). The scanner was designed to follow a grey box methodology, meaning that the tester should have knowledge of client credentials or server credentials in order to proceed with the scan. Additionally, the scanner is capable of conducting passive and active scans.

The objectives of the thesis research were to study and identify potential misconfigurations in the OAuth Code Grant protocol, as reported in RFC documentations [8], [7] and best security practices [18] guidelines, and to select a set of misconfigurations to be implemented in the scanner. The goal of the implemented tool was to help improve the overall security posture of OAuth servers and provide indications of potential exposure to misconfigurations for security engineers who are responsible for configuring these servers. The research also aimed to contribute to the understanding of misconfigurations in the OAuth Code Grant protocol and the risks they pose to web applications that can keep secrets.

1.3 Thesis Structure

The rest of this Master thesis is structured as follows. Section 2 defines all the elements involved in OAuth that are useful for understanding the subsequent analysis, including an overview of OAuth and descriptions of different types of applications and grant types. Section 3 focuses on potential flaws and factors of exploitation in OAuth systems, as well as a summary of selected attacks and an explanation of how they are implemented. Section 4 describes real-case scenarios that demonstrate the need for improved security in OAuth systems and argues for the need to develop security tools to ensure the security of OAuth systems, given that many of these cases are due to poor configuration of OAuth servers. In Section 5, the methodology followed in the research and development of a security tool for OAuth systems is described. Section 6 provides a detailed description of the implementation of the security tool, including the structure and features of the implemented framework. Section 7 presents the results of the tool's analysis of OAuth servers with a defined configuration, as well as servers that are not known cases, and discusses the implications of these results. The last chapter summarizes the main conclusions of the research and discusses potential future implementations of the security tool.

2 OAuth Essentials

One of the most essential parts of the OAuth is that it separates the roles of the involved parties for handling them as separate entities.

2.1 OAuth 2.0 Roles

1. Resource owner – The user whose data the client application wants to access.
2. Resource server – The server that stores restricted resources.
3. Client application – The website or web application that wants to access the user's data.
4. User Agent – The browser.
5. Authorization Server – The server that is in charge of creating and approving the authorization and the authentication data for the Resource Server

The protocol works by defining a series of interactions among three distinct parties, namely a client application, a resource owner, and the OAuth service provider. These parties are visualized in figure 1

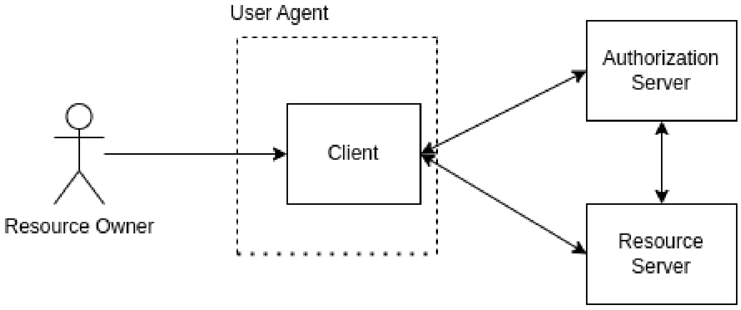


Figure 1: OAuth Roles Overview

Resource Owner

In the OAuth 2.0 flow, the resource owner is simply the user that is interested in granting a registered OAuth application to access their account. Again, there is no disclosure of passwords here or full access to the user's account. The extent to which the user data can be accessed is defined by means of scope. A different scope results in diverse kinds of OAuth 2.0 permission dialogues. Generally, scopes allow permissions such as read or write access to the account data, but it is up to the provider to declare scopes as per their usage.

Resource Server

Resource server contains protected information or user data which can be accessed by the means of access tokens. Simply put, a resource server allows/denies access of a specific resource to an application.

Authorization Server

It is the server that is capable of granting or denying a client an access token. The authorization server authenticates the resource and, generally through various interactions, issues an access token to the client if everything goes as expected.

User Agent

The user agent application is used by the client applications in the user's device, which acts as the scripting language instance such as JavaScript running in a browser. One can store the user agent application on a web server.

2.2 Other useful parameters in OAuth

Redirect URI

Redirect URL are a crucial part of the OAuth flow. After a user successfully authorizes an application, the authorization server will redirect the user back to the application. Due to the sensitive information that the redirect URL contains, it is crucial that the service does not redirect the user to arbitrary locations.

The best way to ensure that the user will only be redirected to appropriate locations, is to require the developer to register one or more redirect URL when they create the application.

Client id

The client ID is a unique identifier that is returned when the application is registered successfully. It is not secret information and it is crucial in the working of OAuth applications. The Client ID is used to identify the application that is used into an OAuth flow.

Client secret

It is a unique token generated during the registration process and is tied to the client ID. As the name suggests, a client secret is private information and should not be exposed. Instead, it is used internally while generating access tokens. During registration, a client secret should be issued only for confidential applications, that could keep secrets.

OAuth Code

An authorization code is an intermediate token used in the server-side application flow. It is returned to the client after the authorization step, and then the client exchanges it for an access token.

State

State parameter is a unique, randomly generated, opaque, and non-guessable string that is sent when starting an authentication request and validated when processing the response. It can be considered as a unique generated value that is used to protect the session from cross site request forgery (CSRF) attacks.

Scope

Scope is a way to limit an application's access to a user's data. Rather than granting complete access to a user's account, it is often useful to give applications a way to request a more limited scope of what they are allowed to do on behalf of a user. Scope is a way to limit what the client can do.

Access Token

The access token represents the authorization of a specific application to access specific parts of a user's data.

2.3 Application Types

Applications could be categorized in two types according to their capability to store secrets.

Confidential are the clients (or applications) that can be deployed with a client which is not visible from someone that uses the application. For example, applications that are running into an application server. API keys that are stored into the application server are not visible out of the application. So, the application could be considered confidential.

Public is the opposite since these clients are unable to store secrets in them. In this case, users that have access to the application could read these values, hence they would not be kept as secrets. Such examples are native applications, SPAs, or mobile apps.

The OAuth server should be able to recognize whether the application is able to store secrets or not in order to be properly configured. In case that a real application is unable to store secrets, the OAuth Server should prevent the issuing of such secrets. Moreover, the OAuth Server should support different policies and be able to act according to the client type that requests authorization.

2.4 PKCE operation

Proof Key for Code Exchange (PKCE) is a protective mechanism that is described in RFC 7636 and tries mostly to protect the handshake from authorization code injection attacks.

The OAuth aims at authorizing applications to access users' resources without the users having to provide their credentials to those applications. Within the grant authorization flow, a code is generated which is a token that the client can exchange (together with its credentials) for an access token.

The PKCE uses a challenge-response protocol to prevent attackers from using intercepted authorization codes. The client generates a code verifier (cryptographically random 43-128 chars in length), calculates a challenge, which is either the same value as verifier or a SHA256 hash of it) and sends the challenge to the Authorization Server in the auth request. The Authorization Server generates an authorization code and associates it with the received challenge. The server then returns the authorization code to the client. This application subsequently exchanges the code verifier together with the authorization code for an access token by contacting Auth Server. The latter entity verifies the challenge using the receiving code verifier. If the verification is successful (challenge matches the code verifier for the plain code challenge method, or the challenge matches a SHA256 hash of the code verifier for the SHA356 code challenge method) the authorization server returns a new access token. If verification fails, it rejects the request.

Schematically:

1. C: generates (code verifier).
2. C-A: sends hashed (code verifier) as (code challenge).
3. A-C: sends authorization Code.
4. C-A: sends token request with (code verifier).
5. A: hashes the (code verifier) and compares to (code challenge) were sent in (b)
6. A-C: if (e) is correct responds with access token.

C: Client – native app mobile or desktop
A: Authorization server

2.5 OAuth Grant Types

There are different ways that the actual OAuth process can be implemented. These are known as OAuth“flows” or “grant types”. These types are differentiated based on the type of application they are supporting and the purpose of this application. Main flows that are most commonly used for web applications are Access Code Grant and Implicit and also other existing flows that are used for other purposes like IoT devices or service to service communications are Client Credentials and Resource Owner Password Credentials grant types.

2.5.1 Access Code Grant

The client application and the OAuth service initially use redirects to exchange a series of browser-based HTTP requests that initiate the flow. The user is asked whether they consent to the requested access. If they accept, the client application is granted an “authorization code”. The client application then exchanges this code with the OAuth service to receive an “access token”, which they can use to make API calls to fetch the relevant user data.

All communication that takes place from the code/token exchange onward is sent server-to-server over a secure, preconfigured back-channel and is, therefore, invisible to the end user. This secure channel is established when the client application first registers with the OAuth service. At this time, a client-secret is also generated, which the client application must use to authenticate itself when sending these server-to-server requests.

As the most sensitive data (the access token and user data) is not sent via the browser, access code grant is arguably the most secure. Server-side applications should ideally always use this grant type if possible. Figure 2 illustrates the anatomy of the handshake for Access Code Grant.

2.5.2 Anatomy of Handshake in Access Code Grant Type

1. Authorization Request (GET from client to AS - /auth).
   (a) response_type = code (indicates the implemented flow).
   (b) client_id = s6BRkt (constitutes a public identifier of client).
   (c) redirect_uri = http://client.example.com/callback (indicates the caller application’s endpoint to the AS).

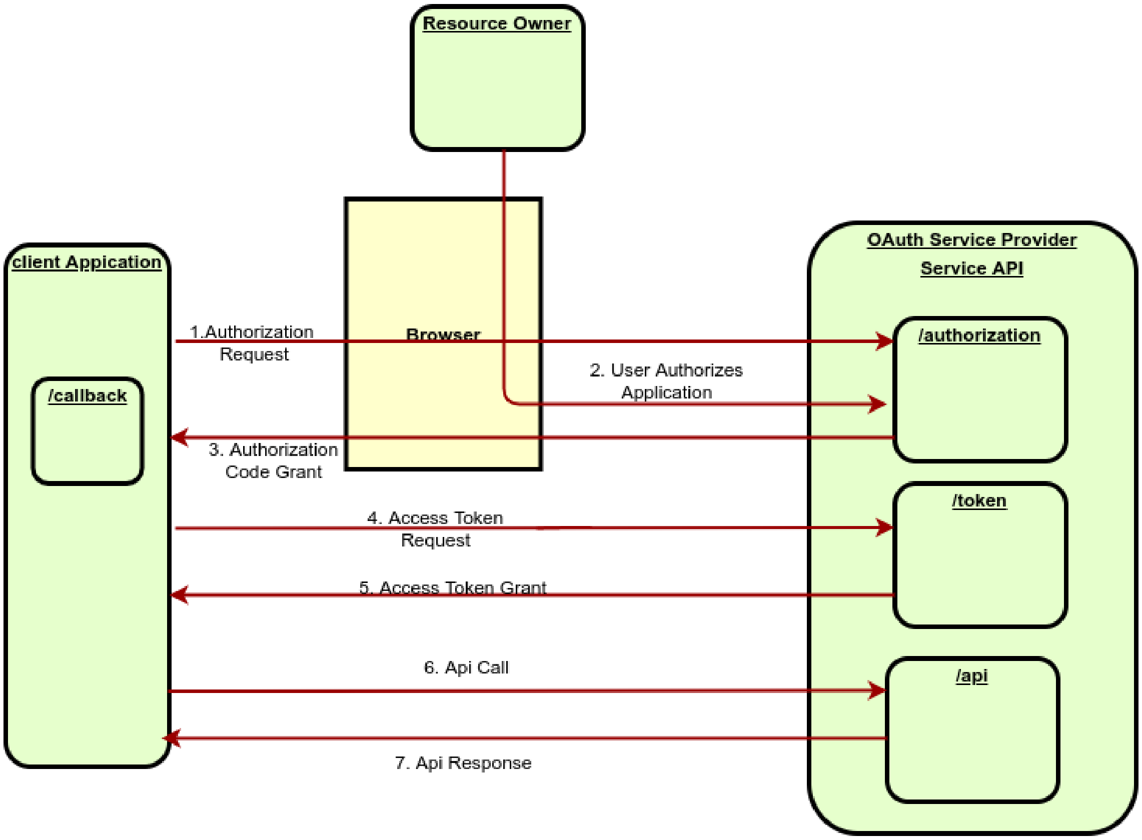


Figure 2: Authorization Code Flow

(d) state = xyz (csrf token - should be treated as all common CSRF tokens - unpredictable, not missing from request, being verified on AS).

(e) scope = api1 api2.read (contain the scope of authority, if does not exist should be assumed that is default - this logic should be implemented on AS).

*(Here let's assume that user gave the authorization from their browser)

2. Authorization Response (it is a redirect on redirect_uri that is sent on the previous request).

(a) code = Sase1AE2Df (authorization code generated uniquely by the authorization server).

   (b) state = xyz (same value as in request above).

3. Token Request (POST from client to /token endpoint).

   (a) (Header variable) Authorization: Basic czZCa3ersgasdfadsf (the application's client Id and secret combined in base64).

   (b) grant_type: authorization_code ().

   (c) code = Sase1AE2Df (authorization code generated uniquely by the authorization server).

   (d) redirect_uri = http://client.example.com/callback (indicates the caller application's endpoint).

   (e) client_id=a32rasdf (application's client id - same as above).

   (f) client_secret=gX12dsf (application's client secret).

   *(The request should contain either header authorization basic or the post body parameters client_id and client_secret)

   Basic Authentication is calculated by RFC 6749 standard:

   Base64(urlformencode(client_id)+":"+ urlformencode(client_secret)).

4. Token Response (application JSON).

   (a) access_token : 2Yoas23aasdfa33aw (token to access the API).

   (b) token_type : Bearer.

   (c) expires_in : 3600.

   (d) scope: api2.read (as above).

2.5.3 Implicit Flow

The implicit grant type is simpler. Rather than first obtaining an authorization code and then exchanging it for an access token, the client application receives the access token immediately after the user gives their consent.

One may be wondering why client applications don't always use the implicit grant type. The answer is relatively simple – it is far less secure. When using the implicit grant type, all communication takes place via browser redirects; there is no secure back-channel like in the authorization code flow. This means that the sensitive access token and the user's data are more exposed to potential attacks.

As it is depicted in figure 3, the implicit grant type is more suited to single-page applications and native desktop applications, which cannot easily store the client-secret on the back-end, and therefore, do not benefit as much from using the authorization code grant type.

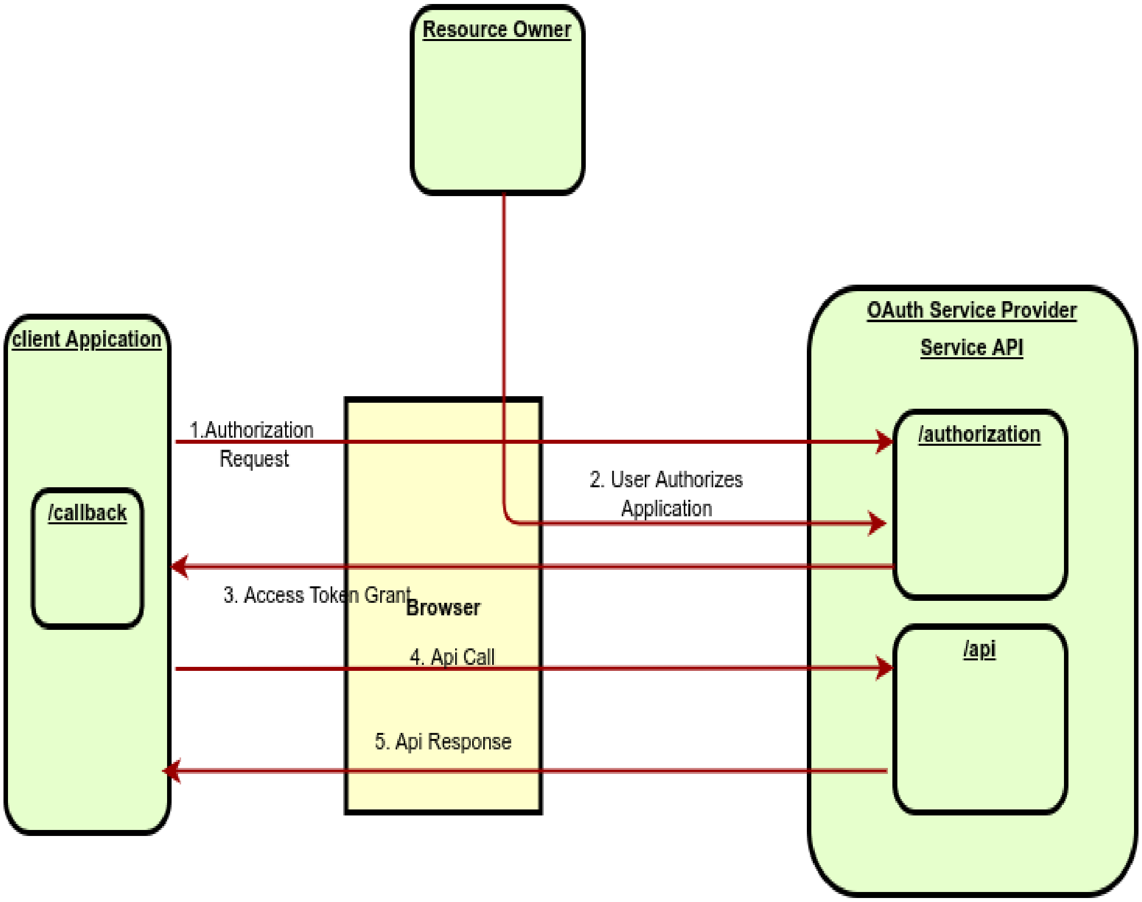


Figure 3: Implicit Flow

2.5.4 Anatomy of Handshake in Implicit Flow

1. Authorization Request (GET from client to AS - /auth).

    (a) response_type = token (indicates the implemented flow).

    (b) client_id = s6BRkt (constitutes a public identifier of client).

(c) redirect_uri = http://client.example.com/callback (indicates the caller application's endpoint to the AS).

(d) state = xyz (CSRF token - should be treated as all common CSRF tokens – unpredictable, not missing from request, being verified on AS).

(e) scope = api1 api2.read (contain the scope of authority, if does not exist it should be assumed that is the default - this logic should be implemented on AS).

*(Here let's assume that the user gave the authorization from his browser)

2. Authorization Response (it is a redirect on redirect_uri sent on the previous request that contains the access token as hashtag).

   This response URL usually contains the access token's value as a hash:

   #access_token = 2Yoaa3af3sdfe

   (a) token_type = example.

   (b) expires_in = 3600.

   (c) state = xyz (same value as in request above).

As it has already been mentioned, there are several other OAuth Grant Types that are used for diverse purposes. In the scope of this master thesis we will focus on the OAuth Code Grant, which is the most common flow to authorize confidential clients.

3 OAuth Exploitation

Regarding how the authorization would be applied there are several potential threats that could lead to authorization exploitation. The landscape of those threats is well-defined according to RFC 6819, titled “OAuth 2.0 Threat Model and Security Considerations”, and could be translated into specific attacks that could be performed against OAuth flows. In this chapter, we will avoid repeating the RFC, therefore our effort is focused on explaining a part of the potential attacks utilized for creating the OAuth Vulnerability Scanner project that it is introduced by this thesis. However, the aforementioned RFC has been studied thoroughly to decide which of the attacks decided to be selected; The next chapter will cover the attacks that are most relevant to the methodology. For the convenience of the reader, we present a simplistic handshake for OAuth Grant Flow, in figure 4 to use it as a reference point.

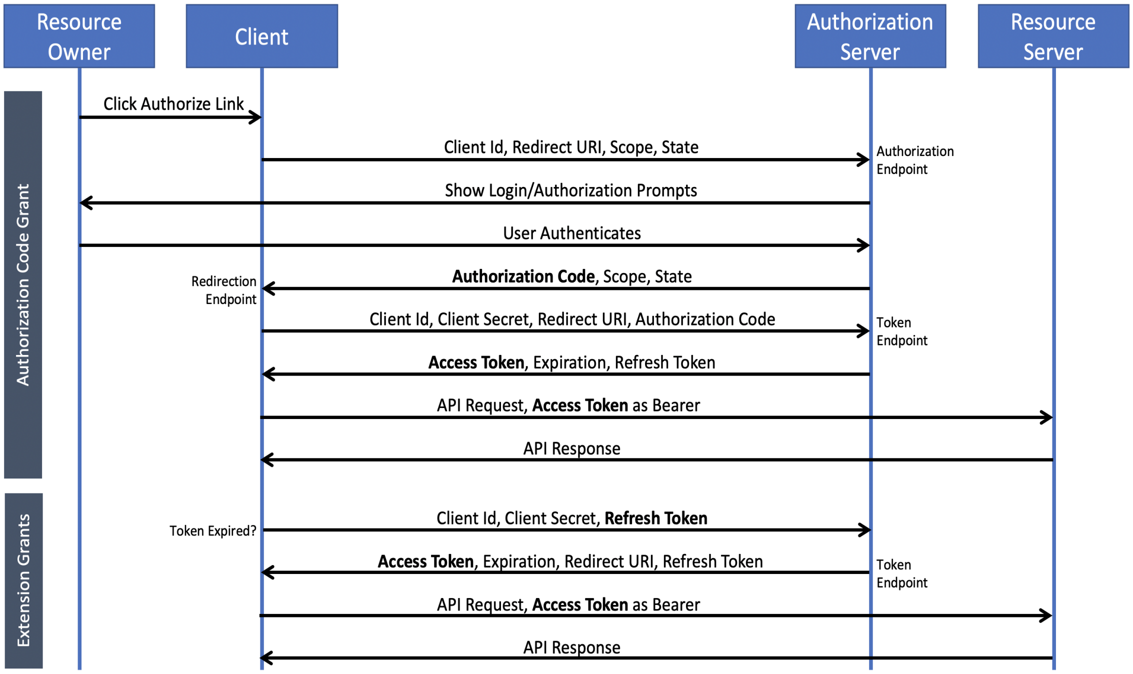

Resource Owner
Client
Authorization Server
Resource Server
Authorization Code Grant
Extension Grants
Click Authorize Link
Client Id, Redirect URI, Scope, State
Authorization Endpoint
Show Login/Authorization Prompts
User Authenticates
Redirection Endpoint
Authorization Code, Scope, State
Client Id, Client Secret, Redirect URI, Authorization Code
Token Endpoint
Access Token, Expiration, Refresh Token
API Request, Access Token as Bearer
API Response
Token Expired?
Client Id, Client Secret, Refresh Token
Access Token, Expiration, Redirect URI, Refresh Token
Token Endpoint
API Request, Access Token as Bearer
API Response

Figure 4: Handshake analysis of Authorization Code Flow.
This resource was taken from `https://levelup.gitconnected.com/oauth-2-0-in-go-846b257d32b4`

3.1 OAuth Potential Flaws or Weaknesses

1. Flawed CSRF protection (State parameter).

   Even if many components of the OAuth flows are optional, some of them are strongly recommended unless there is an important reason not to use them. Such an example is the state parameter. The latter is a value that should ensure the protection of the messages from CSRF attacks that could lead to access someone else resources. Therefore, it is up to the Authorization Server to ensure that the parameter is cryptographically strong enough and also non-repeatable.

2. Flawed redirect_uri validation

   Redirect URI is a crucial parameter as already mentioned in subsection 2.2. Its security is ensured in Authorization Server, so when a new client is registered, the relevant redirect URI has to be defined in the registration process. So, if the Authorization Server does not validate properly the redirect URI or it permits the presence of known anti-patterns (e.g., wildcards), that could lead to another vulnerability. The attacker could be able to use an open redirect attack for obtaining the authorization code from the 4th step of the handshake because it would potentially be redirected to the attacker's client.

3. Flawed scope validation

   Scope is a parameter that ensures the exact resources and the privileges on them that the client could potentially access. However it is up to the Authorization Server to check this value between the steps of the handshake. If this value is not validated properly, a malicious actor could utilize a request to access resources with escalated privileges.

4. Flawed Authorization Code's validation

   Auth code is a sensitive parameter which is used as the primary token to issue a new access token. However it is up to Authorization Server to ensure that this parameter is cryptographically non-repeatable and is implemented by the security standards that the RFC proposes. Such tokens should be one time tokens and to expire in a short time as the relevant RFC about best practices suggests.

5. Flawed PKCE validation

As already mentioned, PKCE is a protective strategy to bind the session in which a specific client requests an authorization code to the action which the same client tries to redeem to an access token. Nevertheless, this validation is partially implemented on the Authorization Server's side, so if the Authorization Server either accepts requests that do not contain PKCE related parameters or improperly validates them, a potential risk would be raised.

6. Flawed Nonce (OpenID) validation

   Based on OpenID specs, nonce parameter is used to associate a client session with an Id_token and to mitigate replay attacks. If there are no measures to prevent replay, an attacker may be able to retrieve a valid authorization request and potentially access/modify the user's resources. Authorization server accepts a request containing a nonce value and issues a new auth code in response.

3.2 Summarizing selected attacks in OAuth

Nonce and Scope related attacks

Implementation of nonce and scope related attacks would be avoided because it is obvious that the attacker could utilize them by an arbitrary value to exploit the flaws already described in paragraphs 3 and 6 in subsection 3.1. The only issue in scope's validation is to detect a valid scope, which would be accepted from the server, in order to try and exploit the privileges. This is part of the results' observation of the passive scan of the implemented framework that will be discussed in next chapter. Nonce attacks stand to the repeatable attempts to use the same value and the inability of the server side to associate the same repeated nonce value with different requests.

URL Redirect Attacks

Such attacks try to target the redirect URI parameter that was defined during the registration of a new client. Each client requires a redirect URI for the Authorization Server to respond to it with an authorization code after a valid user's response in authentication process. These attacks could follow known patterns for exploiting a potential flaw in the definition of such URI.

1. Domain Whitelisting

If the OAuth provider allows clients to be configured without a specific redirect_uri, and the only check is the domain part of the URL, as well as ensuring that the scheme is either https or http and whitelisted, then all of the subdomains of the configured domain could be an issue.

2. Prefix matching

   A valid registered URI could include https://domain.com/a, and https://domain.com/abc could also be accepted.

3. Arbitrary scheme

   If the implementation supports custom schemes it could potentially be another defective case which for the sake of the flexibility could be followed by the native applications. In such a case, URL in the form of x://domain.com/a are allowed.

4. TLD Confusion

   An attacker can bypass certain checks if a suitable top-level domain (TLD) is specified. Someone can bypass the redirect_uri with a .com TLD by replacing it with a suffix such as .com.mx .com.br.

5. Encoding/Decoding attacks

   An attacker could try to use a crafted Unicode in order to exploit the question mark's resolution against the server. For example, %ff or %bf could be used to represent the ? symbol in a URL in order to exploit the query parameter's sign.

6. Backslash separator's trick

   Another trick that could be used is the so-called slash separator trick that can take advantage of the inconsistency between URL validator's and browser's understanding of the / and \. The problem is the potential that the parser has to resolve differently forward slash as path separator in relation to the browser.

Authorization Code Attacks

We have already mentioned the PKCE's importance in paragraph 5 in subsection 3.1. The lack of this value could lead either to CSRF attacks, if the state parameter is improperly implemented, or in Authorization Code Injection attacks.

1. Obtain access token with known client ID and client secret.

   If an attacker have in their possession a valid authorization code plus a client secret and client ID and either an improper or entirely lack of PKCE protection in the OAuth server, they would be able to craft a request using these parameters to redeem an access token.

2. Obtain access without known client ID and client secret (Authorization Code Injection)

   However, if the attacker is unable to obtain client ID and client secret they would have another opportunity to obtain valid access. According to RFC about Best Security Practices [18], in an authorization code injection attack, the attacker attempts to inject a stolen authorization code into the attacker's own session with the client. Specifically, the aim is to associate the attacker's session at the client with the victim's resources or identity.

   This attack is useful if the attacker cannot exchange the authorization code for an access token themselves. Hence, in case that the attacker has already obtained an auth code that has been leaked from a past attack then they can try to inject it in an session initiated by themselves. In particular, they perform a regular OAuth authorization process with the legitimate client on their device. After that, the attacker injects the stolen authorization code in the response of the authorization server to the legitimate client. Since this response is passing through the attacker's device, the attacker can use any tool that can intercept and manipulate the authorization response to this end. Interestingly, the attacker does not need to control the network. Then, the legitimate client sends the code to the authorization server's token endpoint along with the client's ID and secret and actual "redirect_uri". The authorization server checks the client's secret regarding whether the code was issued to the particular client, and whether the actual redirect URI matches the "redirect_uri" parameter. Finally, if all checks succeed the authorization server issues access and other tokens to the client. If so, the attacker has associated their session with the legitimate client with the victim's resources and/or identity.

3. Authorization Replay Attacks

   Apart from PKCE, which if properly implemented can protect the session, it is also the authorization code that should be implemented properly.

Replay attacks could occur if this parameter is not invalidated properly after the first use or if it has a long term lifespan.

4 Real Cases of OAuth Exploitation

Many cases of real-world vulnerable scenarios have taken place in the past. This indicates an increased probability for these to happen again and therefore the implementations must be very restricted to the RFC best practices [18]. Several of such attacks are detailed in the following.

4.1 Slack's Case

A common OAuth vulnerability occurs when a developer improperly configures or validates the redirect_uri parameter, allowing an attacker to steal the OAuth tokens[28]. In 2013 the slack redirect URI restrictions were found that could be bypassed by appending anything to a whitelisted redirect_uri parameter. In other words, Slack was only validating the beginning of the redirect_uri parameter. If a developer registered a new application with Slack and whitelisted https://www.test_domain.com an attacker could append a value to the URL and cause the redirect to go somewhere unintended. Therefore, if the attacker modifying the URL to pass redirect_uri= https://attacker.com would be rejected but passing redirect_uri= https://www.attacker.com.mx would be accepted.

To exploit it, the attacker only has to create a matching subdomain on their malicious site. If a targeted user visits the malicious modified URL, Slack sends the OAuth token to the attacker's site. An attacker could invoke the request on behalf of the targeted victim by attempting a CSRF attack.

Conclusion: Vulnerabilities in which the redirect_uri has not been strictly checked are a common OAuth misconfiguration. Sometimes the vulnerability is the result of an application registering a domain such as *.test_domain.com as an acceptable redirect_uri. Other times [1], it is the result of a resource server not performing a strict check on the beginning and end of the redirect_uri parameter. In this example it was the latter.

4.2 NPM Credentials OAuth Breach

As this case indicates, OAuth related attacks can be very powerful since they can abuse other parties as part of supply chain attacks. Such a case [16] was the steal of user credentials for 100.000 NPM (Node Package Manager) users when a prior attack leaked OAuth access tokens for Heroku and Travis CI. The attack chain involved the attacker abusing the OAuth tokens to exfiltrate private NPM repositories containing AWS access keys, and subsequently leveraging them to gain unauthorized access to the registry's infrastructure.

Heroku has since acknowledged that the theft of GitHub integration OAuth tokens further involved unauthorized access to an internal customer database, prompting the company to reset all user passwords.

Conclusion: Since OAuth credentials are used for service authorization, they are critical and must be kept secret. Any potential compromise could lead to immediate actions like the instant revocation of such credentials. The aforementioned attack is not exactly a case of misconfiguration in OAuth, but can present different angles of security in a supply chain attack that involves OAuth credential stealing as modern attacks try to do.

4.3 Twitter's cases

Two cases reported for OAuth's redirect_uri improper validation in Twitter Periscope mobile application which would potentially lead to account takeover and another one that could lead to unauthorized access to Microsoft Outlook victim's emails. The first [2] Periscope's case disclosed the victim's access token which leads to an account take over because of the existence of a path traversal vulnerability and an open redirect one.

Regarding the second case[17] of Microsoft integration with Twitter, the issue was that Microsoft accepts all the twitter's subdomains as potential clients which is wrong. Under these circumstances, the attacker could handle one of Twitter's subdomain, in this case cards.twitter.com, which is used for advertisement purposes and can redirect to a specific landing page of the advertiser (e.g., https://test.com). Therefore, if the attacker tries to perform a CSRF attack by using a redirect_uri of `https://cards.twitter.com/advertisment_related_path%2523` and a JavaScript script that uses the ”location.hash” API in ”test.com”, the attacker could potentially obtain an access token and use it to access the victim's resources, even if the victim's email address is also included in the scope parameter of the CSRFed URL.

Conclusion: Both vulnerabilities existed due to improper redirect_uri validation but they became exploitable due to the existence of other vulnerabilities, that is open redirects, which was the same in both cases. Redirect URI should be ensured to follow the best practices, do not accept wildcards or other antipatterns in any of the parts of the URL, because exploitation could take place in different places.

5 The Research Methodology

Taking all this information (relevant RFC, upcoming attacks, etc) under consideration it is reasonable to think that having a tool that can detect such misconfigurations would be valuable for developers and administrators of OAuth servers in avoiding relevant vulnerabilities. The relevant work about best practices regarding the protocol was also taken into account along with other RFCs that would help the development of such a tool. That is, it is considered reasonable for an up-to-date OAuth Server to follow the best practices and also be consistent with the RFC 8414, which describes a feature for OAuth in which servers advertise their Metadata.

The followed methodology can be divided in two parts.

Observation: The different implementations of the RFCs were collected and tested to make patterns of them that were used in the implementation given in [26]. From what we managed to patternize, we made the decision about the attacks that could potentially be easier to automate.

Experimental: We tried to perform the applied knowledge of the described attacks as they are mentioned in the RFC threat model document, in an attempt to patternize and to automate some of the most important attacks.

The systems that were used as testbed were six popular OAuth Servers. These servers are summarized in table 1

Server	Stars	Git URL	Type
Keycloak	13.8k	Keycloak	Open Source
Casdoor	4.3k	Casdoor	Open Source
Glewlwyd	398	Glewlwyd	Open Source
a12n-server	345	a12n-server	Open Source
Omejdnr	7	Omejdn	Open Source
Okta	N/A	Okta	Enterprise

Table 1: Popularity of OAuth Servers which were used as testbed

When experimenting with the aforementioned servers, we tried to identify at first if they are compliant to the RFCs that we rely on in order to form the attacks. Moreover, differences in implementations and discrepancies which exist may cause interoperability issues which it is very common phenomenon in OAuth since it is still a prototype and it is up to anyone to follow it . Because of that, we managed to automate the processes only for three of the servers

(Keycloak, Casdoor, OKTA), which also are the most popular ones.

The attacks were selected by the ease of their implementation and their explanation have already been analyzed in a previous subsection 3.2. For the sake of the experiments, we used a demo environment of a dummy malicious client application and each of several OAuth implementations, as the system under test. Moreover, the decision about the Authorization Code Flow has been taken because it is the most common case of the flow especially on three tier web applications' implementation.

6 Implementation

The actual objective of this thesis was to create a new framework that can detect several security issues regarding OAuth misconfigurations in order to aid the developers in configuring properly and securely such servers.

There were several implementation related decisions that we have to address, including the language, the underlying technologies, and more. In this chapter we detail all such decisions that we have to deal with for finalizing a basic implementation. The project is publicly accessible to a Github repository [25] and the contribution to it is more than welcome.

6.1 Architectural Structure

OAuth Vulnerability Scanner is a framework that is implemented in Python language because of its simplicity, highly compatibility, and ease of its use. The framework is organized in a modular base in order to be extendable, maintainable, and easily understandable. The implementation can be represented in a high level as in figure 5.

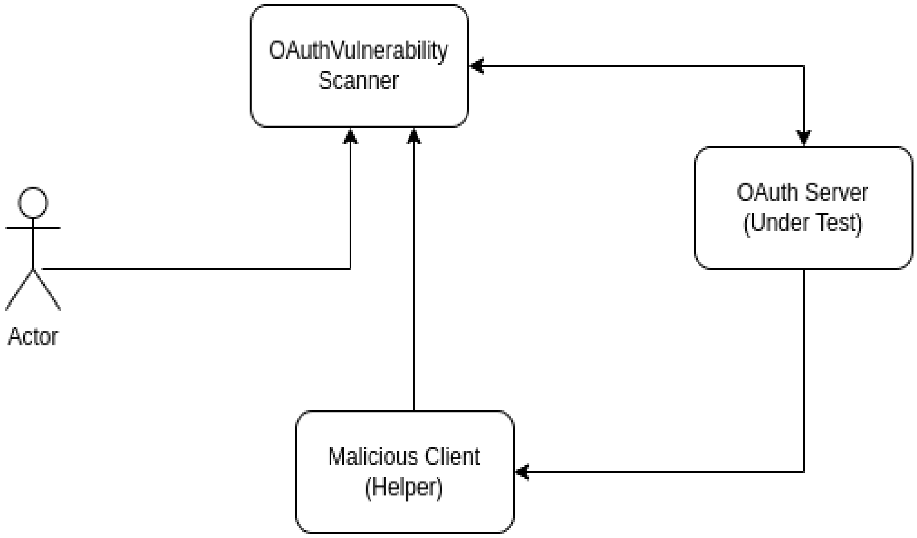

OAuthVulnerability Scanner
OAuth Server (Under Test)
Actor
Malicious Client (Helper)

Figure 5: OAuthVulnerabilityScanner - the involved parties

The framework (OAuthVulnerabilityScanner) acts as the main point which starts the passive and active scans against the tested OAuth Server. The scans require the involvement of another party (Malicious Client) which acts as helper of OAuthVulnerability scanner in order to trick the OAuthServer to trust them.

6.2 Framework implementation details

OAuthVulnerabilityScanner can be explained better by looking at the entities that is composed as depicted in figure 6

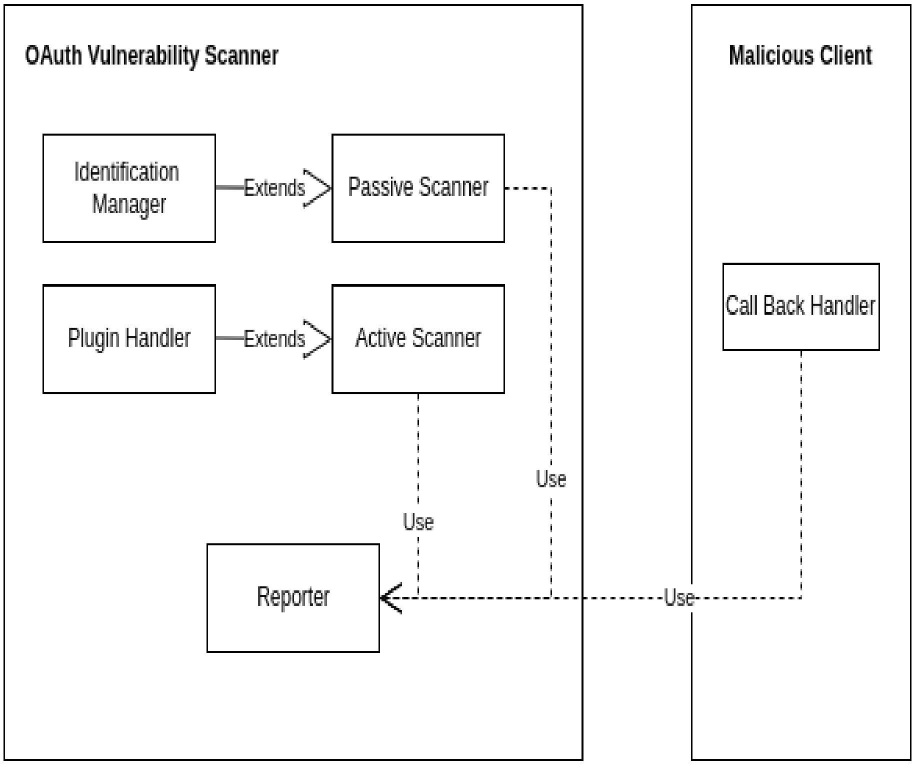


Figure 6: Entity Diagram

Specifically, the framework comprises two main modules: the *Identification Manager* and the *Plugin Handler* which are the abstract classes which are containing the whole functionality for passive and active scanning features. Identification Manager uses internally the passive scanner's module and the *Plugin handler* uses the active scanner's one. Both of them are consuming the *Reporter's* module which is responsible to report the results of the scans that were selected by the user. The other involved party, namely the *Malicious Client*, runs a python Flask server and uses an instance of the reporter to collect the results from its side.

6.3 Implemented Features

The framework's features are modular and function as outlined in subsection 6.2

6.3.1 Passive Scanning

Passive scanning features run during the enumeration phase to retrieve useful information about the targeted OAuth Server. These features are depicted in figure 7

1. Discover well-known endpoint

   Supported OAuth Servers are those that are RFC 8414 compliant. According to this RFC, OAuth Servers should advertise their details to clients so that the clients can use this information appropriately. Thus, well-known endpoint is the place that this information can be retrieved if the OAuth Server implements it. The framework uses a dictionary, which was created during the development of the tool observing different implementations that are compliant to it, to detect the correct well-known endpoint. The information that is described there contains the authorization endpoint, the token endpoint, the supported grant types, the types of scopes, the introspection endpoint if OpenID is supported, and many more. The list of the extracted attributes is defined in the `attr.lst` under the resources directory.

2. Discover JWKS

   The JSON Web Key Set (JWKS) is a set of keys containing the public keys used to verify any JSON Web Token (JWT) issued by the authorization server and signed using SHA256 with RSA (RS256) signing algorithm. JWKS is described thoroughly in the RFC 7517, the enumeration in this feature is to detect other potential endpoints that this information could be presented if it is not in the .well-know retrieved information. An actual case of this enumeration is the OKTA's case that JWKS endpoint is not contained in the well-known results. This info can be utilized to perform attacks in JWTs which potentially could be a part of a future feature.

3. Discover OpenID support

   It is very common for OAuth Servers also to be OpenID Connect providers. OpenID Connect is an authentication layer built on OAuth 2.0. The authorization server (or “OpenID provider” in this case) contains information about a person. OAuth is used to protect this information, allowing the client to access it on behalf of a person. To authorize the dissemination of information, the person is authenticated and the OpenID Provider provides the client application with details including the identity of the person and the time of authentication. If OAuth Server supports OpenID connect, the attack surface is expanding and other attacks that target the OpenID implementation can be performed.

4. Discover other interesting endpoints

   As already mentioned in paragraph 5 in section 5, it is up to implementer what would be implemented from the protocol. Some OAuth Servers would implement more endpoints, and therefore expose useful information that an attacker can use in an attempt to attack the system. Such endpoints could be used for a normal process of the OAuth Server as well but its presence could also be helpful for a threat actor. For this purpose, this part of enumeration tries to identify such endpoints using a dictionary that could easily extend with future discoveries. For example, we are able to access the swagger documentation of a server and to have full knowledge about the implemented endpoints or to have easier access to a considered public piece of information (e.g., client ids) which would not be so easy to find otherwise.

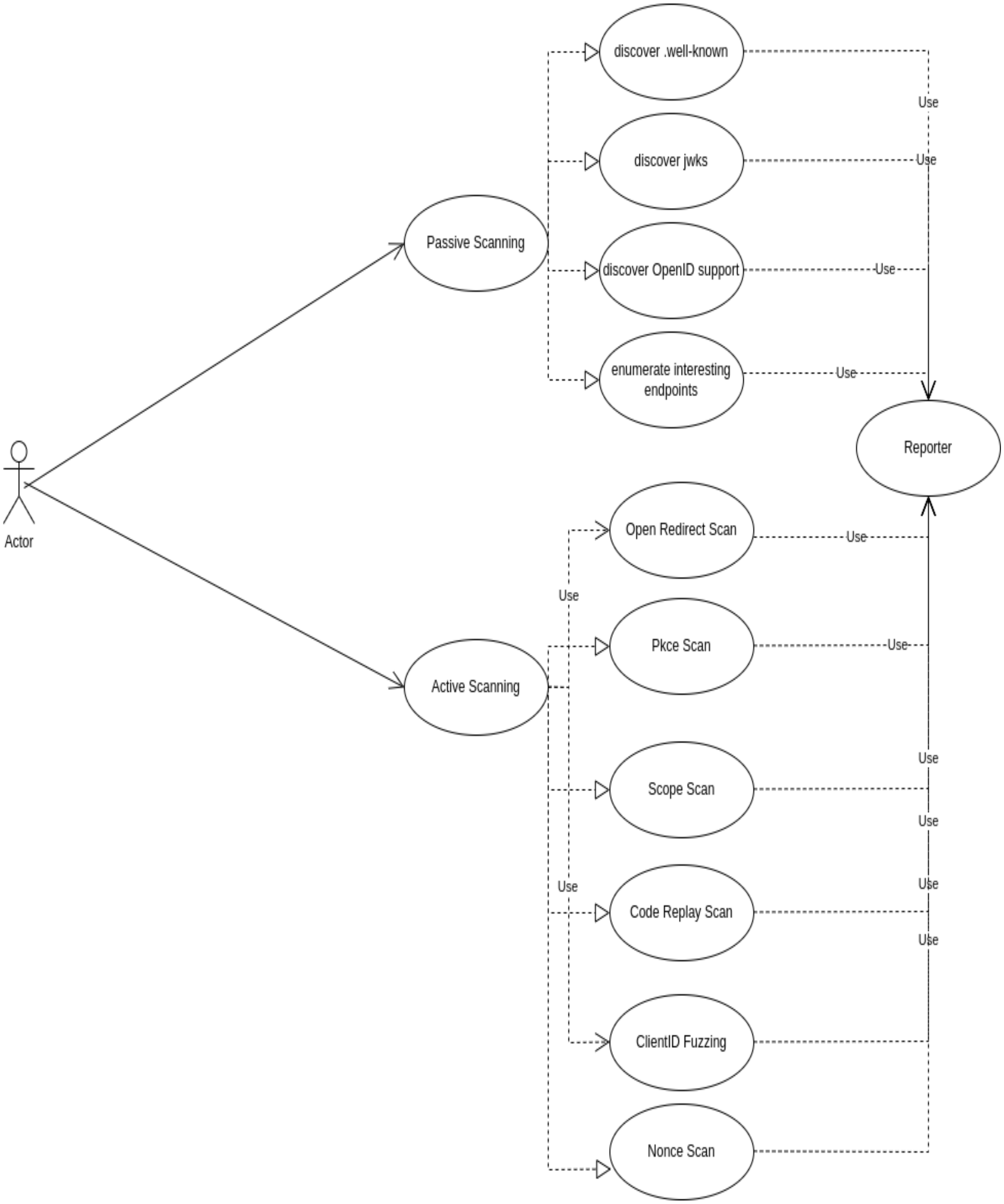


Figure 7: Implemented features

6.3.2 Active Scanning

The background of the flaws and the relevant attacks have already been covered in subsections 3.1 and 3.2, so in this subsection we will explain how the scanner tries to identify these flaws. The absence or the presence of specified parameters, the tolerance of an implementation to the absence of a specific parameter, the system's response in specific crafted requests are some of the applied patterns to address the corresponding issues.

1. Open Redirect Scan

   For this scan, we used an existing Github open source project [22] which was modified to fit in our case. The changes that we applied were mostly to reduce the manual input of session cookies, which is a requirement for the tool to run. To this end, we provide the already captured cookie from the session to the scanner to proceed without any other user involvement. This scan tries to fuzz the redirect URL to detect potential open redirects.

2. PKCE Scan

   It is a suite of 3 test scenarios and it targets to find how the system responds in a PKCE which uses plain text as a challenge method. This is feasible performing two requests per test scenario and comparing the responses. The first request is a normal and expected request and the second one is as it is described in each of the test scenarios. By comparing the responses of those requests we are able to detect if the system is actually protected by PKCE or if this protection could be bypassed by sending the same values for code verifier and code challenge, which means that either the system does not mandate the PKCE presence or it is not evaluated properly. Such a case could be that plain-text is an accepted hashing method which cannot protect the confidentiality of the sending value of the code verifier on the handshake's first step.

3. Scope Scan

   This scan tries to identify if a predefined and improper scope could be used into a request in order to check if the server validates the scope value in general.

4. Code Replay Scan

   This scan attempts to obtain a valid auth code providing the actual credentials, which are required for this test in `settings.json` and to replay it to redeem a new access token from it. If doing so, it is a serious risk since such primary tokens are one time and have to be invalidated after their first use to prevent such kind of actions that could be used to create many access tokens.

5. Nonce Scan

   As already mentioned in paragraph 6 in subsection 3.1, nonce scan is a parameter to protect the client side mostly. Even if it is up to the client to implement its part properly in an OpenID handshake, it could also be partially protected - even if it is not a responsibility of the server to protect it. Therefore, the OAuth server, which in this case also should act as an OpenID Connect provider, could prevent the usage of the same nonce value continuously. The implemented scan tries to use the same predefined nonce value in two different initialization requests to detect if the server allows the nonce replay. Another test that also is implemented tries `nonce=''` value to see if the server evaluates the nonce value or not.

6. ClientID Fuzzing

   This feature uses the known fuzzer `https://wfuzz.readthedocs.io/`, and tries to enumerate the OAuth Server to find some actual client_ids. Even if this information is not considered as private, it requires enumeration to be found and so it is a difficult task to find all client ids. However, if such ids where to be found, the system's security wouldn't be directly compromised but it would be helpful for the attacker to proceed further.

6.3.3 Capabilities and the modes of operation

The scanner provides several capabilities in order to be easier for anyone who works in tool's further development, or on its debugging or for a tester who attempts to provide a specific configuration which does not apply in an already known case.

The scanner uses as input the `settings.json` file which describes the minimum required information for the scanner to work. It contains the `base_url` which is the OAuth server's domain, and also could contain client credentials or/and user credentials in order for the relevant active scans which are required to proceed. In case of navigation mode, it can contain the `firefox_profile` field which is used to resemble an actual user and to avoid the SSO blocking as fraud because of the automated way that Selenium bot works. The selection of the user's profile is a feature of Selenium that the scanner supports and therefore provides the argument to the selenium driver accordingly. If the target is one of the known cases (keycloak,casdoor, okta) it is suggested to use `oauth_vendor` field as well for possibly having better scanning results.

Regarding the supported modes of operations, the scanner covers two main types as it has already been mentioned in subsections 6.3.1 and 6.3.2. Passive scanning, using `type p` argument, runs the whole suite of passive scans to produce the intermediate file `config.json`, which retains the information for the following scans. After the first successful passive scan, the required information has been recorded into the `config.json` file and passive re-scan of the target can be omitted.

After this required step of passive scan, the scanner has the ability to start scanning with the active scan mode `type a` with different values, either to run all the scans providing `mode=ALL` or a single scan choice which can be found providing `--help` argument in CLI.

The scanner also supports the `navigation=selenium` mode which is used to proceed on authentication constraints, so someone could use it to proceed in case that authentication applies differently from user credentials (e.g SSO) or the server is not in the supported ones. As mentioned, Selenium tries to resemble an actual user action to authorize the user to the OAuth server and to become able to proceed to the next steps of the handshake.

A couple features more are the `log` argument, which gives the capability to the tool to create graceful logs and the `proxy`, which enables the scanner to proxy the requests through an intermediate system and it can be used for debugging and development purposes.

Furthermore, some other options are available to the users. Setting the `avoid_replays` argument, enforces the issuing of a new authorization token for every request and the `pkce` which calculates and applies PKCE in the handshake properly if it is a strong requirement from the server's side.

6.3.4 Known Limitations

Open Redirect Scan feature is only supported without the use of navigation mode. This is a known limitation because the implemented scanner lacks the capability to capture the cookie session when runs in navigation mode and this is a limitation of Selenium driver which does not provide this functionality.

Casdoor in `navigation=selenium` mode returns false positive results. It can be used without navigation to get the actual results.

Fuzzing Client returns false positives in casdoor's case because it always returns the same JS response which performs further XMLHttpRequest (XHR) calls. Since the status is coming as a result of these XHR calls the actual status cannot be detected correctly.

`OAuth Vendor` field in `settings.json` config can currently have "Keycloak" or "Casdoor" as values or to be omitted, since only the two servers have already been implemented in the tool.

6.3.5 Environment Setup

If someone needs to run the test scenarios, at first they should deploy and configure the OAuth Servers. For this reason, the repository includes six different OAuth Servers which can run either through docker-compose or with init.sh which is placed under the OAuth Server's directory. For our convenience, we used init.sh during the development process in order to avoid continuously configuring the environment on docker interruption.

Activating the virtual environment:

```
python -m venv /<projects_dir>/OAuthVulnerabilityScanner/OAuthScanner/venv

source venv/bin/activate
```

Moreover, the tester has to start the malicious client which is configured to run in port 4200 by default, by starting the `malicious_client.py` which is under the scanner's directory. The malicious client can be configured changing the port number at the end of the relevant file.

Finally, the user can run the desirable tests by installing the framework's dependencies and running it. It is suggested to use a virtual environment for avoiding polluting the system. To do so, one can follow the next steps accordingly:

```
pip3 install -r requirements
```

Following, access the local or remote servers' environments to configure them properly and use the relevant configuration as input in `settings.json`

6.3.6 Visualization of attacks and potential updates for future development

Figure 8 depicts the results of our research about the selected risk categories (gray) the relevant attacks that could potentially happen (green/red) and potential future upgrades of the framework (blue).

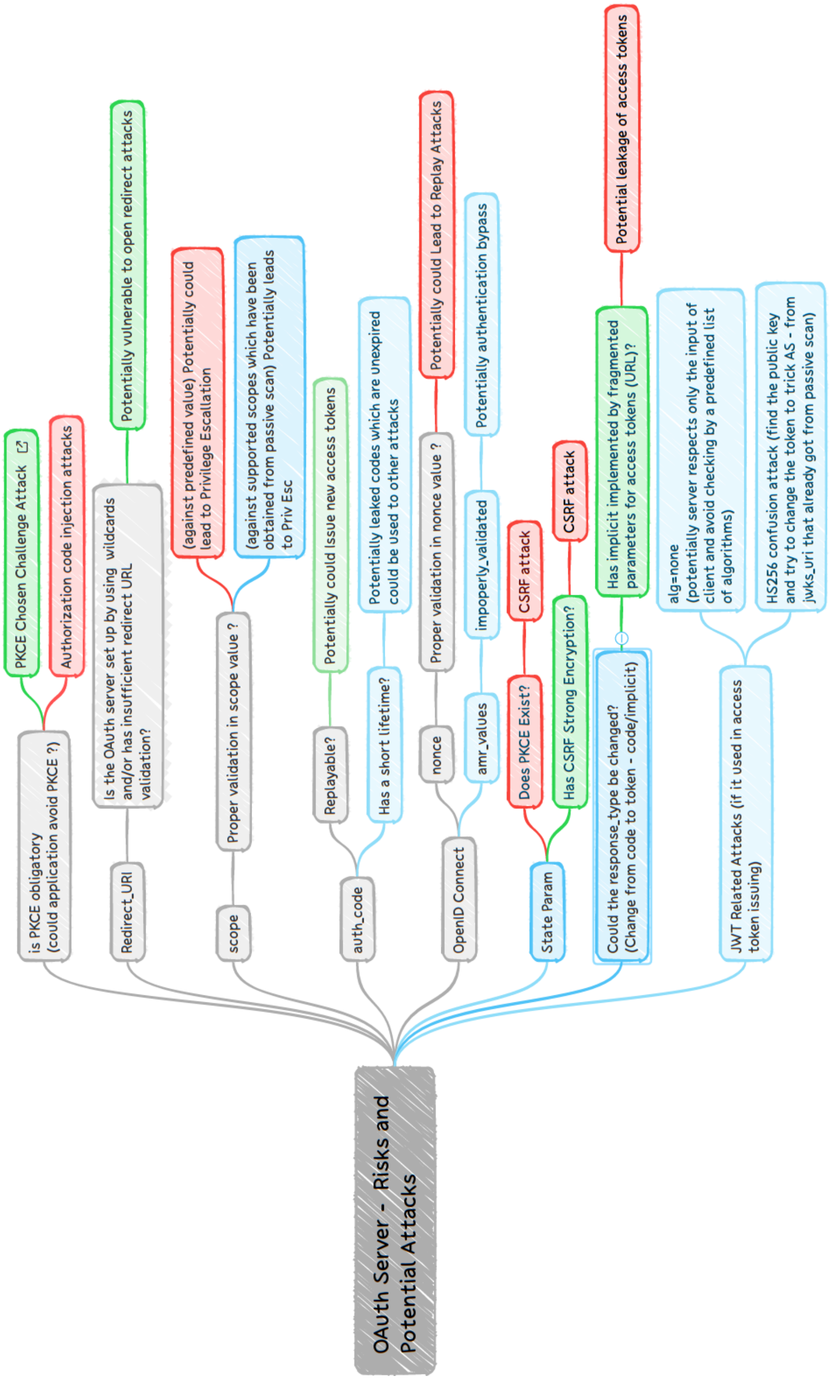

OAuth Server - Risks and Potential Attacks
is PKCE obligatory (could application avoid PKCE ?)
PKCE Chosen Challenge Attack
Authorization code injection attacks
Redirect_URI
Is the OAuth server set up by using wildcards and/or has insufficient redirect URL validation?
Potentially vulnerable to open redirect attacks
scope
Proper validation in scope value ?
(against predefined value) Potentially could lead to Privilege Escallation
(against supported scopes which have been obtained from passive scan) Potentially leads to Priv Esc
auth_code
Replayable?
Potentially could Issue new access tokens
Has a short lifetime?
Potentially leaked codes which are unexpired could be used to other attacks
OpenID Connect
nonce
Proper validation in nonce value ?
Potentially could Lead to Replay Attacks
amr_values
impoperly_validated
Potentially authentication bypass
State Param
Does PKCE Exist?
CSRF attack
Has CSRF Strong Encryption?
CSRF attack
Could the response_type be changed? (Change from code to token - code/implicit)
Has implicit implemented by fragmented parameters for access tokens (URL)?
Potential leakage of access tokens
JWT Related Attacks (if it used in access token issuing)
alg=none (potentially server respects only the input of client and avoid checking by a predefined list of algorithms)
HS256 confusion attack (find the public key and try to change the token to trick AS - from jwks_uri that already got from passive scan)

Figure 8: Implemented and future attacks

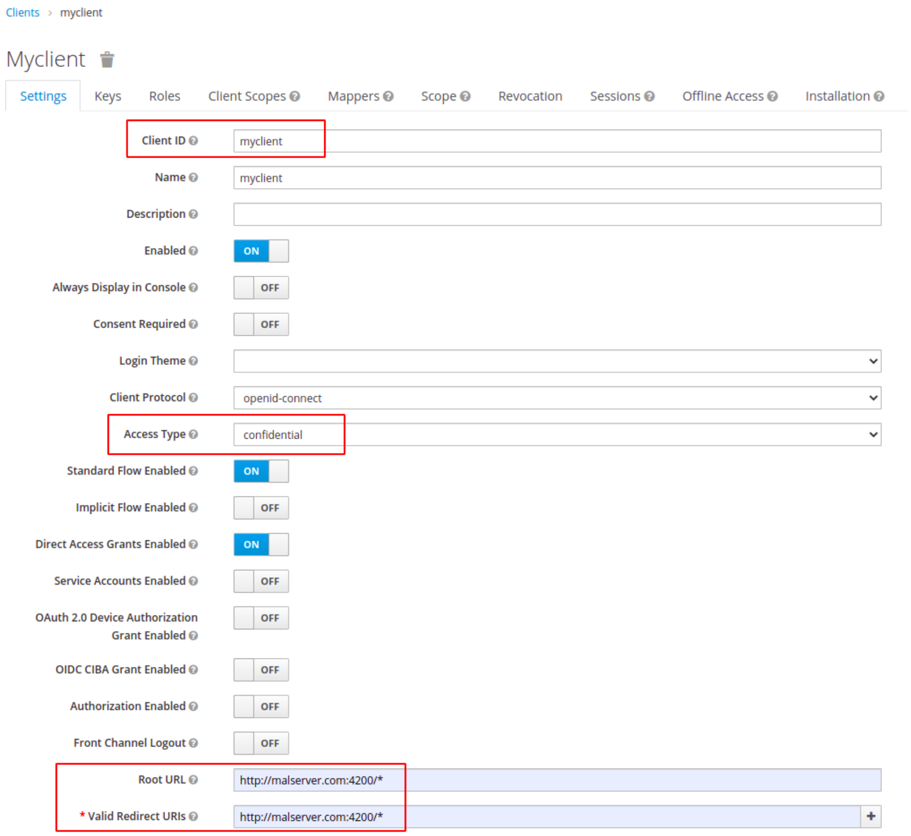
7 Results

Finalizing the research and the implementation of OAuthVulnerabilityScanner, we ran it against the servers included in table 1.

7.1 Keycloak

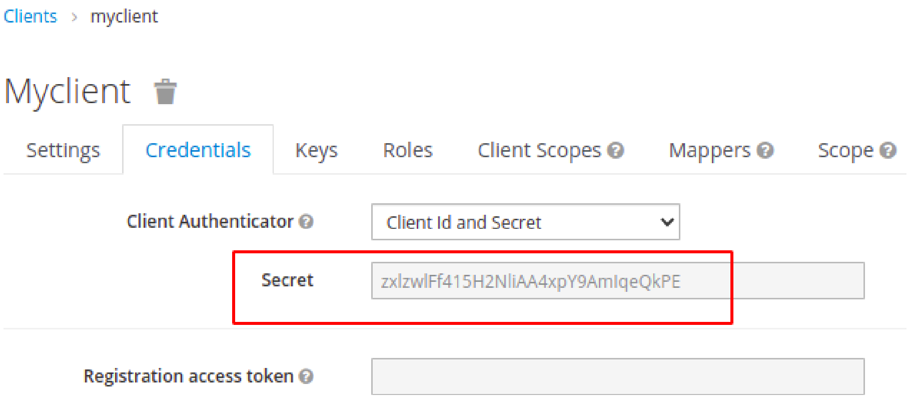
7.1.1 Server Setup and Configuration of the Tool

The server setup that is followed was intentionally misconfigured for the scanner to present some results. We configured a Redirect URI which uses a wild card, which as already have been mentioned in paragraph 2 in subsection 3.1, is a known anti-pattern. This situation is depicted in figure 9.



Client ID	myclient
Name	myclient
Description	
Enabled	ON
Always Display in Console	OFF
Consent Required	OFF
Login Theme	
Client Protocol	openid-connect
Access Type	confidential
Standard Flow Enabled	ON
Implicit Flow Enabled	OFF
Direct Access Grants Enabled	ON
Service Accounts Enabled	OFF
OAuth 2.0 Device Authorization Grant Enabled	OFF
OIDC CIBA Grant Enabled	OFF
Authorization Enabled	OFF
Front Channel Logout	OFF
Root URL	http://malserver.com:4200/*
* Valid Redirect URIs	http://malserver.com:4200/*

Figure 9: Server's setup



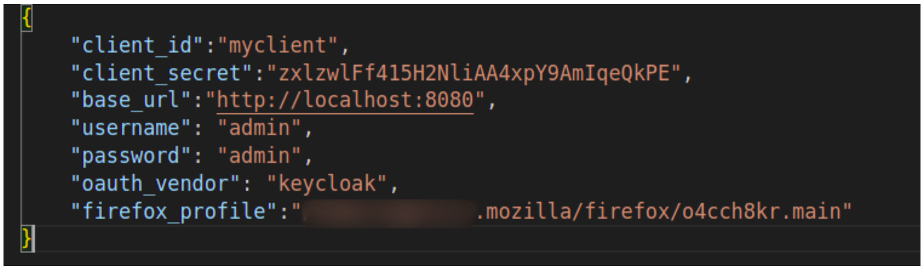
Client Authenticator: Client Id and Secret

Secret: zxlzwlFf415H2NliAA4xpY9AmIqeQkPE

Registration access token:

Figure 10: Server's setup - credentials

We are using the client Id and credentials in order to properly configure the scanner to be able to replicate a malicious client. Moreover, in the config given in figure 11, we provide all the required information, as they are depicted in the figures 9 and 10, for the framework to be able to create requests that require authentication by applying the admin credentials in the file.

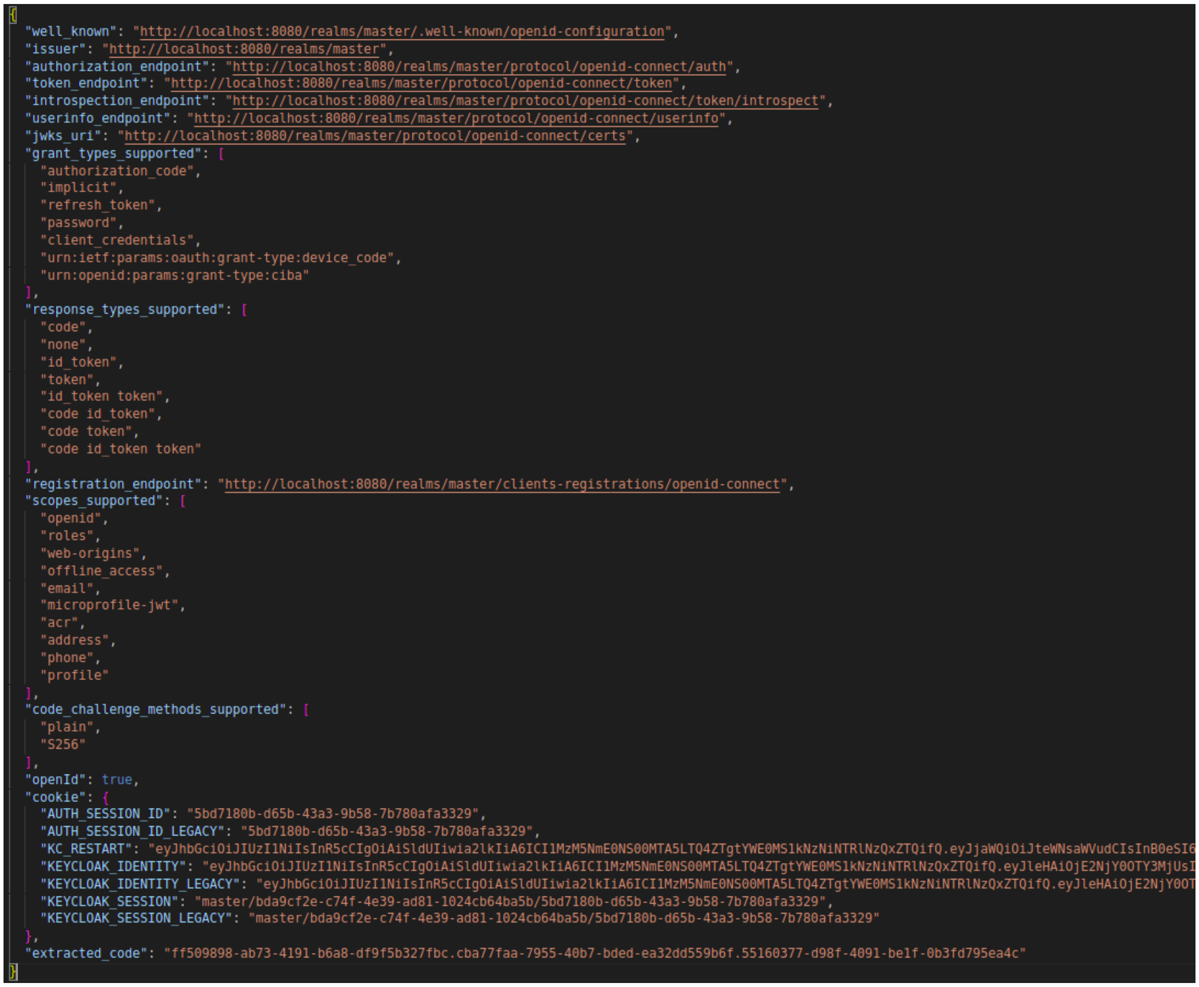


```
{
    "client_id":"myclient",
    "client_secret":"zxlzwlFf415H2NliAA4xpY9AmIqeQkPE",
    "base_url":"http://localhost:8080",
    "username": "admin",
    "password": "admin",
    "oauth_vendor": "keycloak",
    "firefox_profile":"[redacted].mozilla/firefox/o4cch8kr.main"
}
```

Figure 11: Configuration of the tool for Keycloak

7.1.2 Passive Scanning

Results from Keycloak's passive scanning:



```
{
  "well_known": "http://localhost:8080/realms/master/.well-known/openid-configuration",
  "issuer": "http://localhost:8080/realms/master",
  "authorization_endpoint": "http://localhost:8080/realms/master/protocol/openid-connect/auth",
  "token_endpoint": "http://localhost:8080/realms/master/protocol/openid-connect/token",
  "introspection_endpoint": "http://localhost:8080/realms/master/protocol/openid-connect/token/introspect",
  "userinfo_endpoint": "http://localhost:8080/realms/master/protocol/openid-connect/userinfo",
  "jwks_uri": "http://localhost:8080/realms/master/protocol/openid-connect/certs",
  "grant_types_supported": [
    "authorization_code",
    "implicit",
    "refresh_token",
    "password",
    "client_credentials",
    "urn:ietf:params:oauth:grant-type:device_code",
    "urn:openid:params:grant-type:ciba"
  ],
  "response_types_supported": [
    "code",
    "none",
    "id_token",
    "token",
    "id_token token",
    "code id_token",
    "code token",
    "code id_token token"
  ],
  "registration_endpoint": "http://localhost:8080/realms/master/clients-registrations/openid-connect",
  "scopes_supported": [
    "openid",
    "roles",
    "web-origins",
    "offline_access",
    "email",
    "microprofile-jwt",
    "acr",
    "address",
    "phone",
    "profile"
  ],
  "code_challenge_methods_supported": [
    "plain",
    "S256"
  ],
  "openId": true,
  "cookie": {
    "AUTH_SESSION_ID": "5bd7180b-d65b-43a3-9b58-7b780afa3329",
    "AUTH_SESSION_ID_LEGACY": "5bd7180b-d65b-43a3-9b58-7b780afa3329",
    "KC_RESTART": "eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI1MzM5NmE0NS00MTA5LTQ4ZTgtYWE0MS1kNzNiNTRlNzQxZTQifQ.eyJjaWQiOiJteWNsaWVudCIsInB0eSI6
    "KEYCLOAK_IDENTITY": "eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI1MzM5NmE0NS00MTA5LTQ4ZTgtYWE0MS1kNzNiNTRlNzQxZTQifQ.eyJleHAiOjE2NjY0OTY3MjUsI
    "KEYCLOAK_IDENTITY_LEGACY": "eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICI1MzM5NmE0NS00MTA5LTQ4ZTgtYWE0MS1kNzNiNTRlNzQxZTQifQ.eyJleHAiOjE2NjY0OT
    "KEYCLOAK_SESSION": "master/bda9cf2e-c74f-4e39-ad81-1024cb64ba5b/5bd7180b-d65b-43a3-9b58-7b780afa3329",
    "KEYCLOAK_SESSION_LEGACY": "master/bda9cf2e-c74f-4e39-ad81-1024cb64ba5b/5bd7180b-d65b-43a3-9b58-7b780afa3329"
  },
  "extracted_code": "ff509898-ab73-4191-b6a8-df9f5b327fbc.cba77faa-7955-40b7-bded-ea32dd559b6f.55160377-d98f-4091-be1f-0b3fd795ea4c"
}
```

Figure 12: Passive Scanning Results

As it is presented in the figure 12, the scanner detected all the relevant information trying the "well-known" endpoint. As observed from figure 12, the scanner also found that the server supports OpenID and managed to use non-navigation mode producing the relevant cookies which were used to authenticate the user's session. Moreover, from the results we infer other useful information that could be used for further testing beyond the features of the scanner. Our current scanner implementation does not fully utilize all available information to identify all potential vulnerabilities. Further considerations regarding these vulnerabilities are discussed in section 8.

For example, figure 12 presents the **JWKs endpoint**. Knowing the JWKS endpoint of an OAuth server can potentially allow an attacker to learn about the keys that are used by server to sign and verify JWTs. This information can be used in a number of ways to attack the OAuth server or its clients.

One potential threat is that an attacker could use this information to impersonate a client and obtain access to protected resources. For example, if an attacker can obtain a JWT signed by the OAuth server, they can use it to access resources on behalf of a legitimate client. This could allow the attacker to gain unauthorized access to sensitive data or perform actions on behalf of the client.

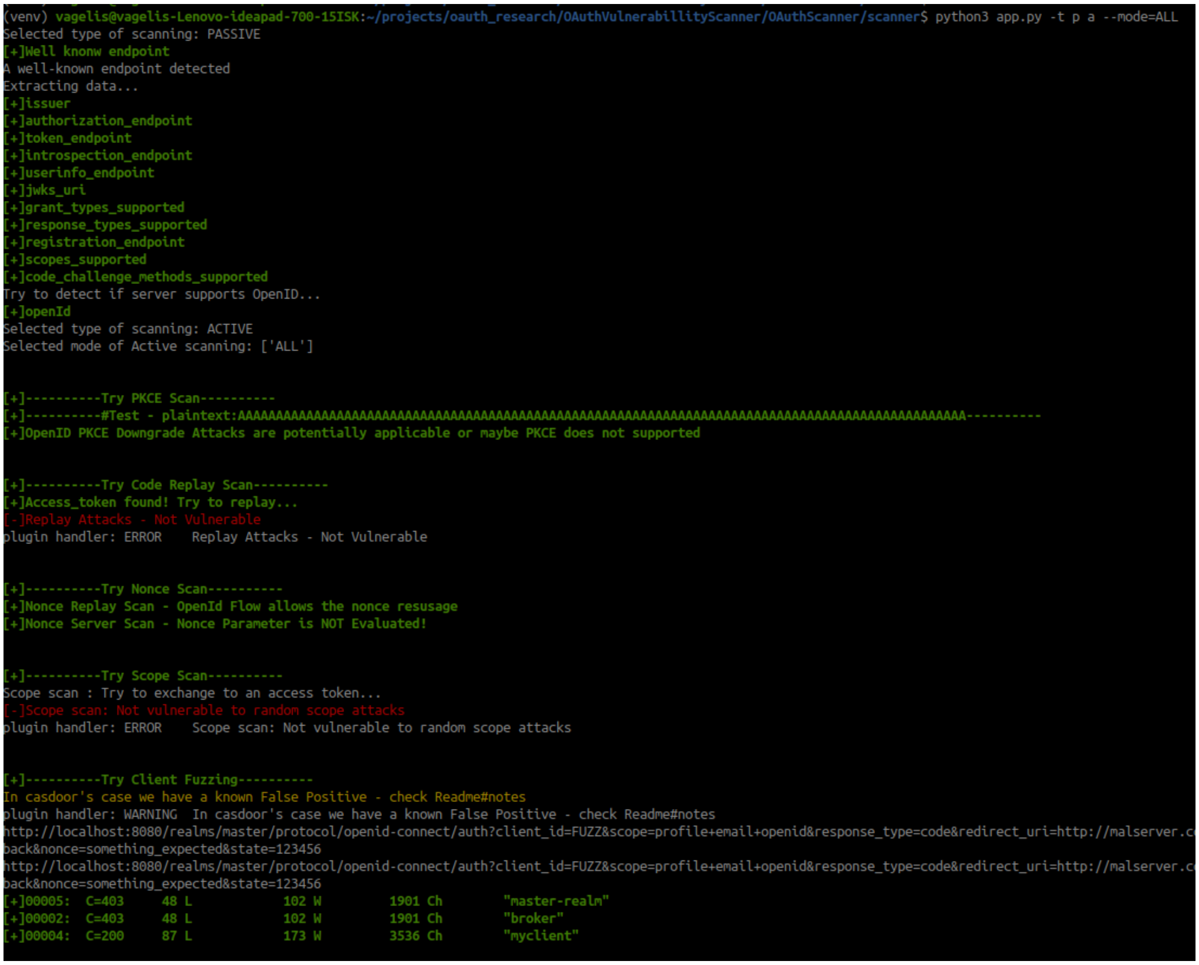
Another potential threat is that an attacker could use the JWKS endpoint to learn about vulnerabilities in the server's cryptographic keys. For example, if the keys are weak or have been compromised, an attacker could potentially use this information to launch an attack on the server or its clients.

Information exposure about **registration endpoint** would potentially become an issue. Knowing the registration endpoint can potentially allow to an attacker to register new client applications with the server. This can be a serious threat, as it could allow to the attacker to gain access to protected resources or perform actions on behalf of the server. An attacker could use this information to register a malicious client application that is designed to capture sensitive data or perform unauthorized actions (it can be an app which appears legitimate, but actually captures user credentials or access tokens and sends them to the attacker). In another scenario, the attacker could use the registration endpoint to register multiple client apps, potentially overwhelming the server with requests and causing a Denial of Service (DoS) attack.

Also the presence of the **introspection endpoint** can potentially allow an attacker to obtain information about JWTs that are used by the server. Additionally the **userinfo endpoint** of an OAuth server can potentially allow an attacker to obtain information about users who are registered with the server.This can be a serious threat, as it could allow the attacker to gain access to sensitive personal information or perform actions on behalf of the user. Utilizing information for a particular user like name, email address etc could be used to launch targeted phising attacks or to impersonate the user.

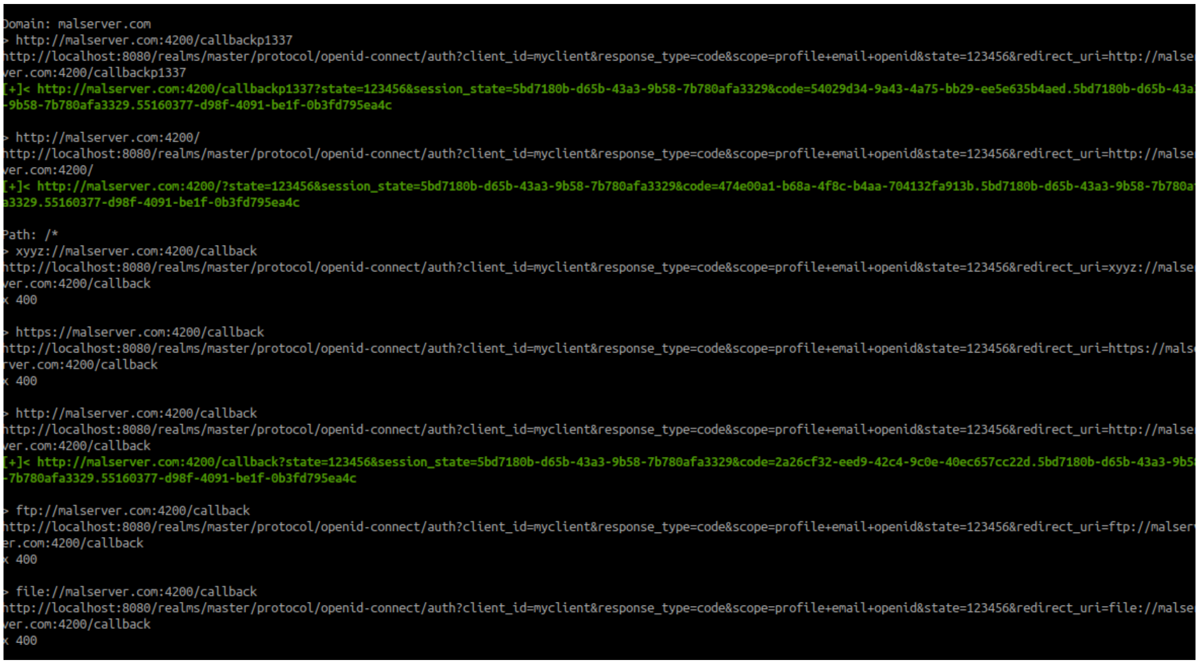
7.1.3 Active Scanning

Figure 13 depicts the CLI in action. It runs both passive and active types of scan and uses all the tests from the active scan suite. The earlier part of the results states that the scanner started passive scanning and the results already have been presented in subsection 7.1.2. Next, the scanner indicates the beginning of active scanning, which contains all the results from the running test of the test suite.



```
(venv) vagelis@vagelis-Lenovo-ideapad-700-15ISK:~/projects/oauth_research/OAuthVulnerabillityScanner/OAuthScanner/scanner$ python3 app.py -t p a --mode=ALL
Selected type of scanning: PASSIVE
[+]Well knonw endpoint
A well-known endpoint detected
Extracting data...
[+]issuer
[+]authorization_endpoint
[+]token_endpoint
[+]introspection_endpoint
[+]userinfo_endpoint
[+]jwks_uri
[+]grant_types_supported
[+]response_types_supported
[+]registration_endpoint
[+]scopes_supported
[+]code_challenge_methods_supported
Try to detect if server supports OpenID...
[+]openId
Selected type of scanning: ACTIVE
Selected mode of Active scanning: ['ALL']


[+]----------Try PKCE Scan----------
[+]----------#Test - plaintext:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA----------
[+]OpenID PKCE Downgrade Attacks are potentially applicable or maybe PKCE does not supported


[+]----------Try Code Replay Scan----------
[+]Access_token found! Try to replay...
[-]Replay Attacks - Not Vulnerable
plugin handler: ERROR    Replay Attacks - Not Vulnerable


[+]----------Try Nonce Scan----------
[+]Nonce Replay Scan - OpenId Flow allows the nonce resusage
[+]Nonce Server Scan - Nonce Parameter is NOT Evaluated!


[+]----------Try Scope Scan----------
Scope scan : Try to exchange to an access token...
[-]Scope scan: Not vulnerable to random scope attacks
plugin handler: ERROR    Scope scan: Not vulnerable to random scope attacks


[+]----------Try Client Fuzzing----------
In casdoor's case we have a known False Positive - check Readme#notes
plugin handler: WARNING  In casdoor's case we have a known False Positive - check Readme#notes
http://localhost:8080/realms/master/protocol/openid-connect/auth?client_id=FUZZ&scope=profile+email+openid&response_type=code&redirect_uri=http://malserver.c
back&nonce=something_expected&state=123456
http://localhost:8080/realms/master/protocol/openid-connect/auth?client_id=FUZZ&scope=profile+email+openid&response_type=code&redirect_uri=http://malserver.c
back&nonce=something_expected&state=123456
[+]00005:  C=403     48 L          102 W       1901 Ch       "master-realm"
[+]00002:  C=403     48 L          102 W       1901 Ch       "broker"
[+]00004:  C=200     87 L          173 W       3536 Ch       "myclient"
```

Figure 13: Active Scanning Results



```
Domain: malserver.com
> http://malserver.com:4200/callbackp1337
http://localhost:8080/realms/master/protocol/openid-connect/auth?client_id=myclient&response_type=code&scope=profile+email+openid&state=123456&redirect_uri=http://malse
ver.com:4200/callbackp1337
[+]< http://malserver.com:4200/callbackp1337?state=123456&session_state=5bd7180b-d65b-43a3-9b58-7b780afa3329&code=54029d34-9a43-4a75-bb29-ee5e635b4aed.5bd7180b-d65b-43a
-9b58-7b780afa3329.55160377-d98f-4091-be1f-0b3fd795ea4c

> http://malserver.com:4200/
http://localhost:8080/realms/master/protocol/openid-connect/auth?client_id=myclient&response_type=code&scope=profile+email+openid&state=123456&redirect_uri=http://malse
ver.com:4200/
[+]< http://malserver.com:4200/?state=123456&session_state=5bd7180b-d65b-43a3-9b58-7b780afa3329&code=474e00a1-b68a-4f8c-b4aa-704132fa913b.5bd7180b-d65b-43a3-9b58-7b780a
a3329.55160377-d98f-4091-be1f-0b3fd795ea4c

Path: /*
> xyyz://malserver.com:4200/callback
http://localhost:8080/realms/master/protocol/openid-connect/auth?client_id=myclient&response_type=code&scope=profile+email+openid&state=123456&redirect_uri=xyyz://malse
ver.com:4200/callback
x 400

> https://malserver.com:4200/callback
http://localhost:8080/realms/master/protocol/openid-connect/auth?client_id=myclient&response_type=code&scope=profile+email+openid&state=123456&redirect_uri=https://mals
rver.com:4200/callback
x 400

> http://malserver.com:4200/callback
http://localhost:8080/realms/master/protocol/openid-connect/auth?client_id=myclient&response_type=code&scope=profile+email+openid&state=123456&redirect_uri=http://malse
ver.com:4200/callback
[+]< http://malserver.com:4200/callback?state=123456&session_state=5bd7180b-d65b-43a3-9b58-7b780afa3329&code=2a26cf32-eed9-42c4-9c0e-40ec657cc22d.5bd7180b-d65b-43a3-9b5
-7b780afa3329.55160377-d98f-4091-be1f-0b3fd795ea4c

> ftp://malserver.com:4200/callback
http://localhost:8080/realms/master/protocol/openid-connect/auth?client_id=myclient&response_type=code&scope=profile+email+openid&state=123456&redirect_uri=ftp://malser
er.com:4200/callback
x 400

> file://malserver.com:4200/callback
http://localhost:8080/realms/master/protocol/openid-connect/auth?client_id=myclient&response_type=code&scope=profile+email+openid&state=123456&redirect_uri=file://malse
ver.com:4200/callback
x 400
```

Figure 14: Open Redirect Results

Table 2 illustrates the results of active scanning as they depicted in figure 13.

Scan Suite	Results
PKCE Scan	Potentially Vulnerable or Not Required
Auth Replay Scan	Not Vulnerable
Nonce Scan	Vulnerable
Scope Scan	Not Vulnerable
Client Fuzzing	Detected 3 Client Ids

Table 2: Keycloak's active scanning

The results depicted in figure 13 demonstrate that PKCE scan successfully run and the reported that either the PKCE protection is not enabled and so the system is vulnerable to PKCE downgrade attacks in subsection 3.2 or the PKCE is not supported at all. Replay attacks in ”code” were not applicable which means that the server is not vulnerable to Authorization Replay Attacks as they have been presented in paragraph 3 in subsection 3.2. Additionally the Nonce values are both allowed to be reused and actually are not evaluated at all from the server. A nonce replay attack is a type of security threat in which an attacker intercepts a request that includes a nonce (a randomly generated value that is used to prevent replay attacks) and re-sends the request with the same nonce. This can allow the attacker to gain unauthorized access to a system or perform actions on behalf of the victim.

To prevent nonce replay attacks, OAuth servers can use techniques such as server-side nonce tracking and nonce expiration to ensure that nonces are only used once and are not valid for an extended period of time. It is also important for OAuth clients to verify the nonce value received in a response to ensure that it matches the nonce value sent in the request.

Moreover, the rest of the findings presented in figure 13 show that scope scan are not applicable in random scopes (far from the supported ones) to escalate privileges to a scope beyond of the server's supported ones. Finally, scanner managed to enumerate three clients by using the method which has been described in paragraph 6 in subsection 6.3.2. Client Id has been described in subsection 2.2, an attacker with knowledge of the client id can use it to impersonate the legitimate client and request access to protected resources on behalf of the user. If the OAuth server does not properly verify the client's identity, the attacker may be able to gain access to the user's resources without their knowledge or consent.

To prevent this type of attack, it is important for OAuth servers to implement proper authentication and authorization measures to verify the identity of the client before granting access to protected resources. This may include using secure communication channels and requiring the client to provide additional authentication credentials, such as a client secret or an access token.

The redirect URI was detected as a vulnerable one as we expected since it was intentionally configured beyond the best practices and it detected. The results that are presented in figure 14 and summarized in table 3 are relevant to the final part of the report, which presents partial results of open redirect scan. If an OAuth system is vulnerable to open redirects, an attacker can potentially intercept a request and redirect the user to a malicious website after they have granted access to their resources. This can allow the attacker to gain access to the user's resources or steal sensitive information.

To prevent open redirect attacks, it is important for OAuth servers to properly validate redirect URLs and ensure that they are only redirecting to trusted and authorized locations. This may include implementing measures such as URL whitelisting and using secure communication channels. It is also important for OAuth clients to properly validate redirect URLs and ensure that they are not being redirected to malicious websites.

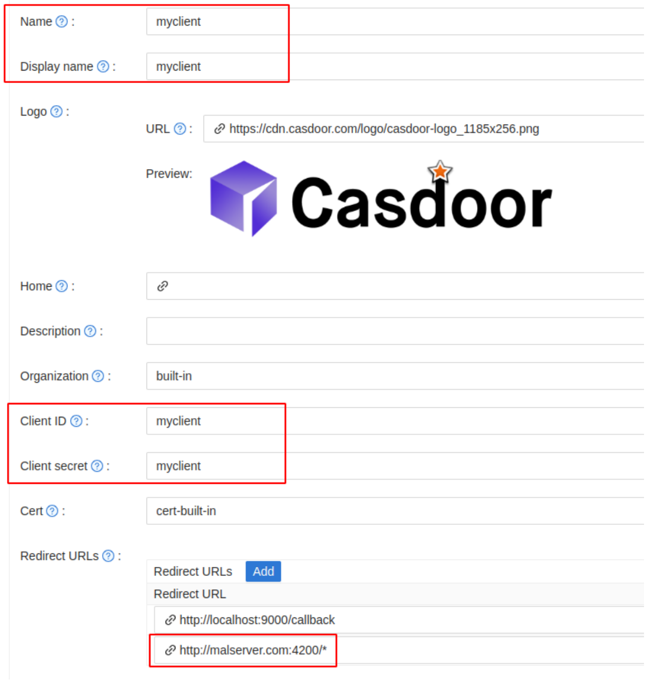
OpenRedirect Scan
http://malserver.com:4200/
http://malserver.com:4200/callback
http://malserver.com:4200/callbackp1337
http://malserver.com:4200/callback../
http://malserver.com:4200/callback..;/
http://malserver.com:4200/callback./
http://malserver.com:4200/callback%2e%2e

Table 3: Keycloak's open redirect results

7.2 Casdoor

7.2.1 Server Setup and Configuration of the Tool

Following, as depicted in figure 15, we present an intentionally misconfigured Casdoor server which was used in the tests.



Name ⓘ : myclient

Display name ⓘ : myclient

Logo ⓘ :

URL ⓘ : https://cdn.casdoor.com/logo/casdoor-logo_1185x256.png

Preview:

Home ⓘ :

Description ⓘ :

Organization ⓘ : built-in

Client ID ⓘ : myclient

Client secret ⓘ : myclient

Cert ⓘ : cert-built-in

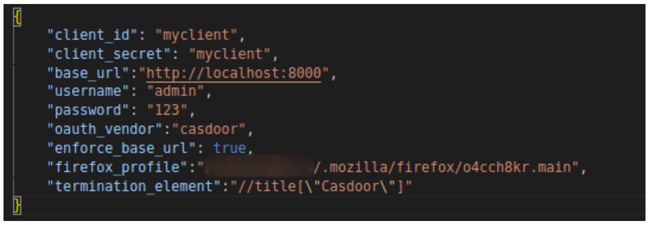
Redirect URLs ⓘ :

http://localhost:9000/callback

http://malserver.com:4200/*

Figure 15: Casdoor configuration

Figure 16 illustrates the relevant configuration that was used at tool's side to run tests for the Casdoor's case:

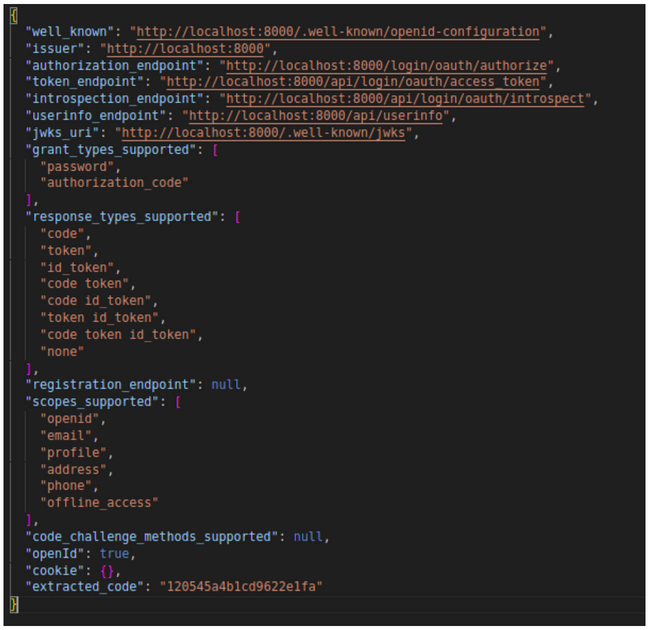


```
{
    "client_id": "myclient",
    "client_secret": "myclient",
    "base_url":"http://localhost:8000",
    "username": "admin",
    "password": "123",
    "oauth_vendor":"casdoor",
    "enforce_base_url": true,
    "firefox_profile":"[redacted]/.mozilla/firefox/o4cch8kr.main",
    "termination_element":"//title[\"Casdoor\"]"
}
```

Figure 16: Configuration of the Tool for Casdoor

7.2.2 Passive Scanning

The results of passive scanning present the OAuth Server's domain and the important endpoints, grant types which are supported and the relevant scopes accordingly. As we notice in figure 17, the server also supports OpenID Connect and does not support PKCE without providing further configuration. Additionally, there are potential attacks in paragraph 7.1.2 in subsection 7.1, that are outside the scope of the current implementation but could be carried out using the information in figure 17.

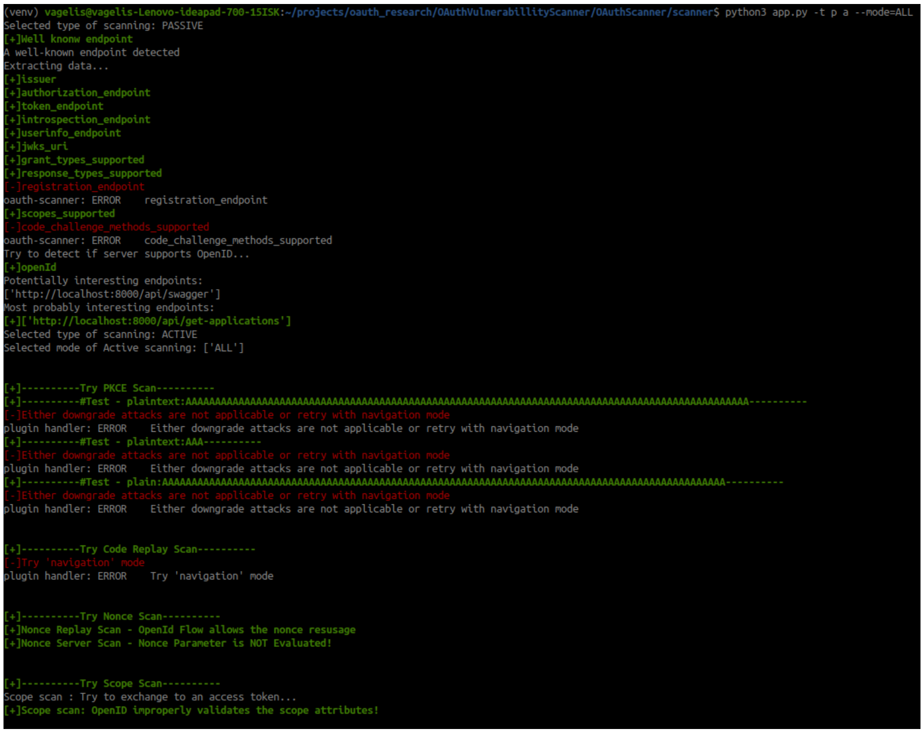


```
{
  "well_known": "http://localhost:8000/.well-known/openid-configuration",
  "issuer": "http://localhost:8000",
  "authorization_endpoint": "http://localhost:8000/login/oauth/authorize",
  "token_endpoint": "http://localhost:8000/api/login/oauth/access_token",
  "introspection_endpoint": "http://localhost:8000/api/login/oauth/introspect",
  "userinfo_endpoint": "http://localhost:8000/api/userinfo",
  "jwks_uri": "http://localhost:8000/.well-known/jwks",
  "grant_types_supported": [
    "password",
    "authorization_code"
  ],
  "response_types_supported": [
    "code",
    "token",
    "id_token",
    "code token",
    "code id_token",
    "token id_token",
    "code token id_token",
    "none"
  ],
  "registration_endpoint": null,
  "scopes_supported": [
    "openid",
    "email",
    "profile",
    "address",
    "phone",
    "offline_access"
  ],
  "code_challenge_methods_supported": null,
  "openId": true,
  "cookie": {},
  "extracted_code": "120545a4b1cd9622e1fa"
}
```

Figure 17: Casdoor Passive Scanning results

7.2.3 Active Scanning

Regarding the results presented in this subsection, we used the same approach as previously by scanning Casdoor server both with passive and active scanning using all suites for the active one. As depicted in figure 18, on the upper part of it we can see the passive scan which identifies all the information that was reported in the previous section. Furthermore, we notice that the code challenge method and registration endpoint have not been discovered, which means that most probably PKCE is not a default feature for this server and the server potentially has not an automatic registration endpoint for clients.

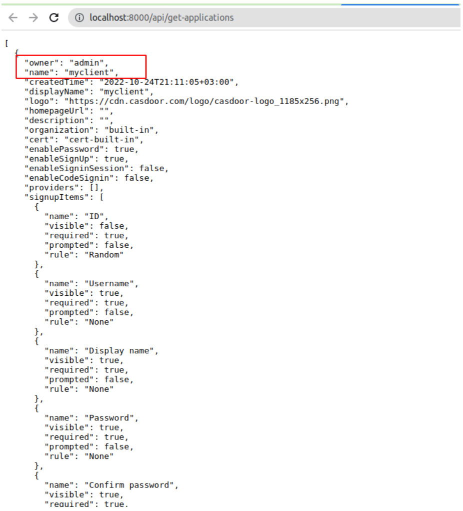


```
(venv) vagelis@vagelis-Lenovo-ideapad-700-15ISK:~/projects/oauth_research/OAuthVulnerabillityScanner/OAuthScanner/scanner$ python3 app.py -t p a --mode=ALL
Selected type of scanning: PASSIVE
[+]Well knonw endpoint
A well-known endpoint detected
Extracting data...
[+]issuer
[+]authorization_endpoint
[+]token_endpoint
[+]introspection_endpoint
[+]userinfo_endpoint
[+]jwks_uri
[+]grant_types_supported
[+]response_types_supported
[-]registration_endpoint
oauth-scanner: ERROR    registration_endpoint
[+]scopes_supported
[-]code_challenge_methods_supported
oauth-scanner: ERROR    code_challenge_methods_supported
Try to detect if server supports OpenID...
[+]openId
Potentially interesting endpoints:
['http://localhost:8000/api/swagger']
Most probably interesting endpoints:
[+]['http://localhost:8000/api/get-applications']
Selected type of scanning: ACTIVE
Selected mode of Active scanning: ['ALL']


[+]----------Try PKCE Scan----------
[+]----------#Test - plaintext:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA----------
[-]Either downgrade attacks are not applicable or retry with navigation mode
plugin handler: ERROR    Either downgrade attacks are not applicable or retry with navigation mode
[+]----------#Test - plaintext:AAA----------
[-]Either downgrade attacks are not applicable or retry with navigation mode
plugin handler: ERROR    Either downgrade attacks are not applicable or retry with navigation mode
[+]----------#Test - plain:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA----------
[-]Either downgrade attacks are not applicable or retry with navigation mode
plugin handler: ERROR    Either downgrade attacks are not applicable or retry with navigation mode


[+]----------Try Code Replay Scan----------
[-]Try 'navigation' mode
plugin handler: ERROR    Try 'navigation' mode


[+]----------Try Nonce Scan----------
[+]Nonce Replay Scan - OpenId Flow allows the nonce resusage
[+]Nonce Server Scan - Nonce Parameter is NOT Evaluated!


[+]----------Try Scope Scan----------
Scope scan : Try to exchange to an access token...
[+]Scope scan: OpenID improperly validates the scope attributes!
```

Figure 18: Casdoor Active Scanning results

As we notice from the figure above we see that PKCE validations don't pass the test so the suite is failing for all the provided tests (we will retry later with navigation mode), also Code Replay scan suggests to run it again in navigation mode. Also the results which are from `Nonce Scan` and `Scope Scan` indicate that the Server is vulnerable or misconfigured for both of these cases.

From the results of upper part of figure 18 it is also discovered that the server has an endpoint that potentially is an interesting one (figure 19). We inspected the endpoint and we found that it contains information about the registered clients, therefore an attacker who is able to access this endpoint could be able to enumerate further the OAuth Server's clients.

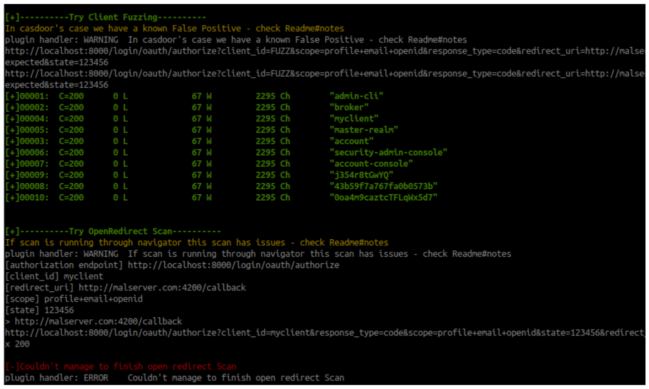


localhost:8000/api/get-applications

```
[
  {
    "owner": "admin",
    "name": "myclient",
    "createdTime": "2022-10-24T21:11:05+03:00",
    "displayName": "myclient",
    "logo": "https://cdn.casdoor.com/logo/casdoor-logo_1185x256.png",
    "homepageUrl": "",
    "description": "",
    "organization": "built-in",
    "cert": "cert-built-in",
    "enablePassword": true,
    "enableSignUp": true,
    "enableSigninSession": false,
    "enableCodeSignin": false,
    "providers": [],
    "signupItems": [
      {
        "name": "ID",
        "visible": false,
        "required": true,
        "prompted": false,
        "rule": "Random"
      },
      {
        "name": "Username",
        "visible": true,
        "required": true,
        "prompted": false,
        "rule": "None"
      },
      {
        "name": "Display name",
        "visible": true,
        "required": true,
        "prompted": false,
        "rule": "None"
      },
      {
        "name": "Password",
        "visible": true,
        "required": true,
        "prompted": false,
        "rule": "None"
      },
      {
        "name": "Confirm password",
        "visible": true,
        "required": true,
```

Figure 19: Casdoor's endpoint which exposes the clients

As shown in figure 20, and as it has already been spotted in the known issues section, Casdoor returns false positives in the client's fuzzing tests. Moreover, open redirect scan did not provided the expected results because the cookie cannot be captured correctly after the first attempt.



```
[+]----------Try Client Fuzzing----------
In casdoor's case we have a known False Positive - check Readme#notes
plugin handler: WARNING  In casdoor's case we have a known False Positive - check Readme#notes
http://localhost:8000/login/oauth/authorize?client_id=FUZZ&scope=profile+email+openid&response_type=code&redirect_uri=http://malse
expected&state=123456
http://localhost:8000/login/oauth/authorize?client_id=FUZZ&scope=profile+email+openid&response_type=code&redirect_uri=http://malse
expected&state=123456
[+]00001:  C=200      0 L          67 W        2295 Ch       "admin-cli"
[+]00002:  C=200      0 L          67 W        2295 Ch       "broker"
[+]00004:  C=200      0 L          67 W        2295 Ch       "myclient"
[+]00005:  C=200      0 L          67 W        2295 Ch       "master-realm"
[+]00003:  C=200      0 L          67 W        2295 Ch       "account"
[+]00006:  C=200      0 L          67 W        2295 Ch       "security-admin-console"
[+]00007:  C=200      0 L          67 W        2295 Ch       "account-console"
[+]00009:  C=200      0 L          67 W        2295 Ch       "j354r8tGwYQ"
[+]00008:  C=200      0 L          67 W        2295 Ch       "43b59f7a767fa0b0573b"
[+]00010:  C=200      0 L          67 W        2295 Ch       "0oa4m9caztcTFLqWx5d7"


[+]----------Try OpenRedirect Scan----------
If scan is running through navigator this scan has issues - check Readme#notes
plugin handler: WARNING  If scan is running through navigator this scan has issues - check Readme#notes
[authorization endpoint] http://localhost:8000/login/oauth/authorize
[client_id] myclient
[redirect_uri] http://malserver.com:4200/callback
[scope] profile+email+openid
[state] 123456
> http://malserver.com:4200/callback
http://localhost:8000/login/oauth/authorize?client_id=myclient&response_type=code&scope=profile+email+openid&state=123456&redirect
x 200

[-]Couldn't manage to finish open redirect Scan
plugin handler: ERROR    Couldn't manage to finish open redirect Scan
```

Figure 20: Casdoor's active scanning rest results

Following, as it is represented in figure 21 we tried to use the navigator's mode to get actual results from the auth code's replay scan. For this reason, we use `--navigation=selenium` argument in the CLI and selenium spawned up a new window which requests authentication in order to proceed with the process. As we provided the correct credentials, we received the results which show that Code replay is not feasible (figure 22).

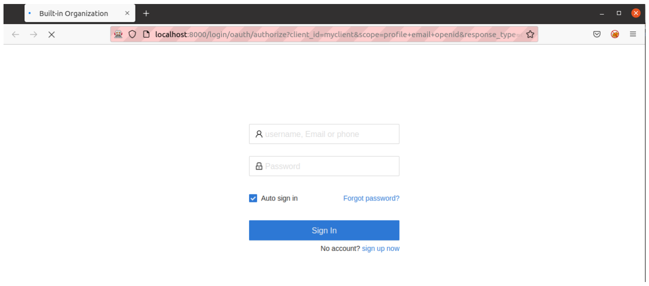
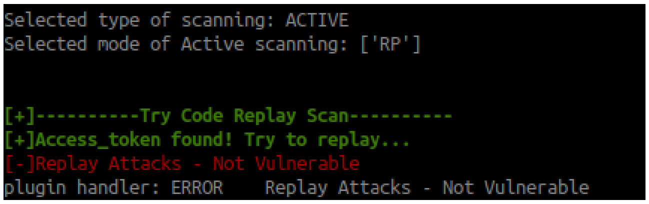
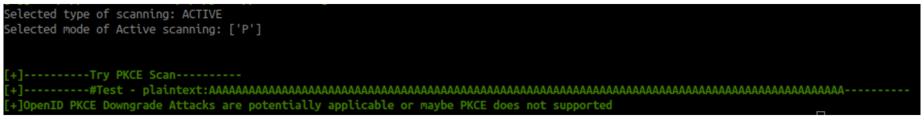


Figure 21: The tool uses the Selenium driver to get authentication



```
Selected type of scanning: ACTIVE
Selected mode of Active scanning: ['RP']


[+]----------Try Code Replay Scan----------
[+]Access_token found! Try to replay...
[-]Replay Attacks - Not Vulnerable
plugin handler: ERROR     Replay Attacks - Not Vulnerable
```

Figure 22: Casdoor replay scan results



```
Selected type of scanning: ACTIVE
Selected mode of Active scanning: ['P']


[+]----------Try PKCE Scan----------
[+]----------#Test - plaintext:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA----------
[+]OpenID PKCE Downgrade Attacks are potentially applicable or maybe PKCE does not supported
```

Figure 23: Casdoor PKCE scan results

Figure 23 shows the results from Casdoor PKCE scanning which indicate that the server is either vulnerable on PKCE related attacks or the value is not evaluated from the server by default.

We can spot that even if we are unable to enumerate because of the known issue in Casdoor, we got the results from the *api-clients* API which was discovered through passive scanning (figure 19).

Table 4 provides a summary of the results for Casdoor Server.

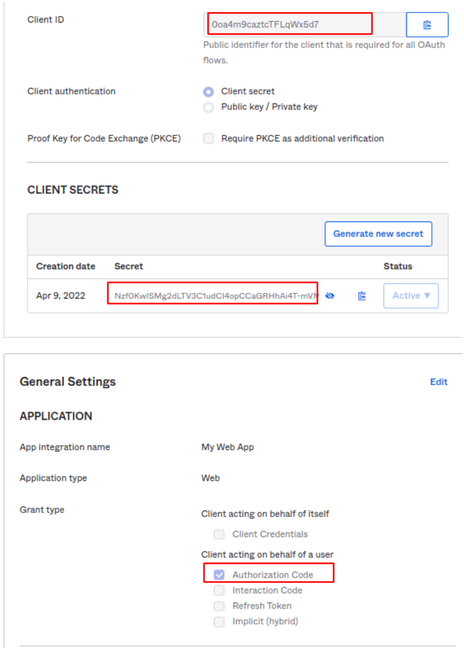
Scan Suite	Results
PKCE Scan	Potentially Vulnerable or Not Required
Auth Replay Scan	Not Vulnerable
Nonce Scan	Vulnerable
Scope Scan	Vulnerable
Client Fuzzing	False Positives
Open Redirect	Partial Results

Table 4: Active scanning for Casdoor

7.3 OKTA

As illustrated in figures 24 and 25, we configured a client which will be used in OAuth's Code Flow. In this case, the OAuth Code Flow is intentionally configured to allow wildcards as redirect URIs in order to detect this redirect vulnerability through our scanner.

7.3.1 Server Setup and Configuration of the Tool



Client ID: 0oa4m9caztcTFLqWx5d7

Public identifier for the client that is required for all OAuth flows.

Client authentication: ● Client secret ○ Public key / Private key

Proof Key for Code Exchange (PKCE): □ Require PKCE as additional verification

CLIENT SECRETS

Creation date	Secret	Status
Apr 9, 2022	Nzf0KwlSMg2dLTV3C1udCl4opCCaGRHhAi4T-mVM	Active ▼

General Settings

APPLICATION

App integration name: My Web App

Application type: Web

Grant type:

Client acting on behalf of itself

□ Client Credentials

Client acting on behalf of a user

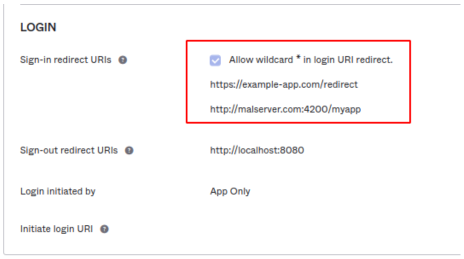
☑ Authorization Code

□ Interaction Code

□ Refresh Token

□ Implicit (hybrid)

Figure 24: OKTA's setup client config - client



LOGIN

Sign-in redirect URIs: ☑ Allow wildcard * in login URI redirect.

https://example-app.com/redirect

http://malserver.com:4200/myapp

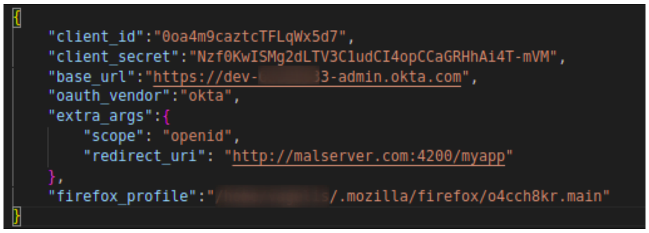
Sign-out redirect URIs: http://localhost:8080

Login initiated by: App Only

Initiate login URI

Figure 25: OKTA's setup client config - redirect_uri

As depicted in figure 25, the same configuration was used to properly configure the tool in order to be aware of the client's credentials. As shown in figure 26, the configuration also uses the `extra_args` attribute which allows for the explicit definition of values that are needed to be taken into account from the tool during the tests. To this end, we explicitly request the desired scope that we need to use and to configure the redirect_uri as a different value from the default one.

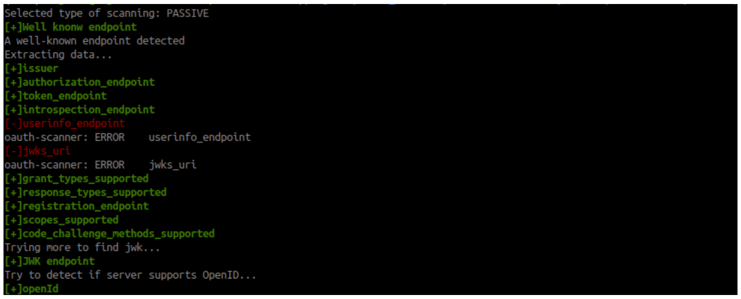


```
{
    "client_id":"0oa4m9caztcTFLqWx5d7",
    "client_secret":"Nzf0KwISMg2dLTV3CludCI4opCCaGRHhAi4T-mVM",
    "base_url":"https://dev-[redacted]3-admin.okta.com",
    "oauth_vendor":"okta",
    "extra_args":{
        "scope": "openid",
        "redirect_uri": "http://malserver.com:4200/myapp"
    },
    "firefox_profile":"[redacted]/.mozilla/firefox/o4cch8kr.main"
}
```

Figure 26: Tools configuration for OKTA

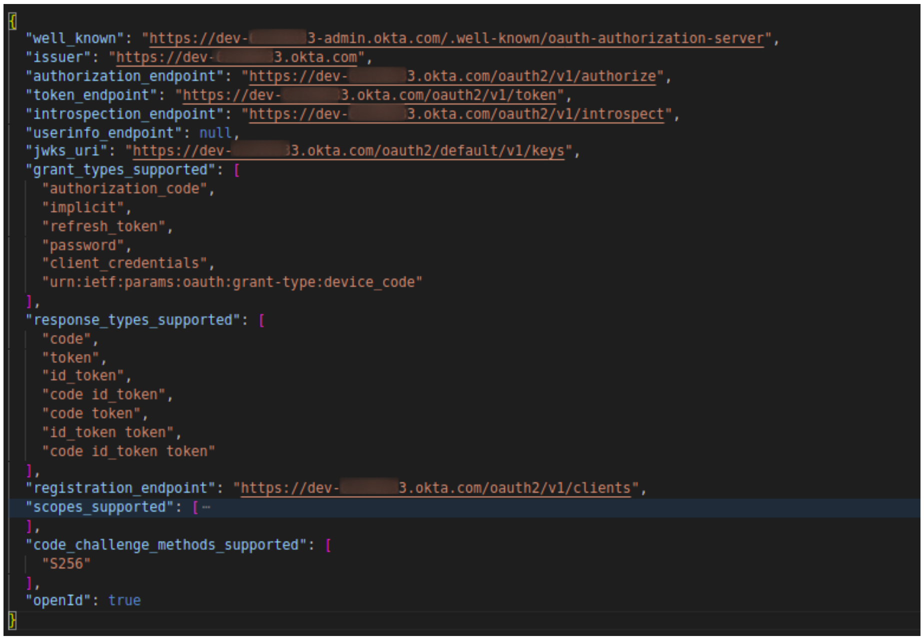
7.3.2 Passive Scanning

Proceeding to the passive scanning we obtained the results as the figures 27 and 28 indicate. We tried to enumerate, and managed to find all the information about the supported types, grants, and scopes. Even if the well-known endpoint did not contain the jwks_uri, we managed to search thoroughly and to detect it by trying potential endpoints that would contain this info which are related to this type of passive scan, using a dictionary into resources directory. Additionally, there are potential attacks in paragraph 7.1.2 in subsection 7.1, that are outside the scope of the current implementation but could be carried out using the information in figure 28.



```
Selected type of scanning: PASSIVE
[+]Well knonw endpoint
A well-known endpoint detected
Extracting data...
[+]issuer
[+]authorization_endpoint
[+]token_endpoint
[+]introspection_endpoint
[-]userinfo_endpoint
oauth-scanner: ERROR    userinfo_endpoint
[-]jwks_uri
oauth-scanner: ERROR    jwks_uri
[+]grant_types_supported
[+]response_types_supported
[+]registration_endpoint
[+]scopes_supported
[+]code_challenge_methods_supported
Trying more to find jwk...
[+]JWK endpoint
Try to detect if server supports OpenID...
[+]openId
```

Figure 27: OKTA passive scanning results in CLI



```
{
  "well_known": "https://dev-[redacted]3-admin.okta.com/.well-known/oauth-authorization-server",
  "issuer": "https://dev-[redacted]3.okta.com",
  "authorization_endpoint": "https://dev-[redacted]3.okta.com/oauth2/v1/authorize",
  "token_endpoint": "https://dev-[redacted]3.okta.com/oauth2/v1/token",
  "introspection_endpoint": "https://dev-[redacted]3.okta.com/oauth2/v1/introspect",
  "userinfo_endpoint": null,
  "jwks_uri": "https://dev-[redacted]3.okta.com/oauth2/default/v1/keys",
  "grant_types_supported": [
    "authorization_code",
    "implicit",
    "refresh_token",
    "password",
    "client_credentials",
    "urn:ietf:params:oauth:grant-type:device_code"
  ],
  "response_types_supported": [
    "code",
    "token",
    "id_token",
    "code id_token",
    "code token",
    "id_token token",
    "code id_token token"
  ],
  "registration_endpoint": "https://dev-[redacted]3.okta.com/oauth2/v1/clients",
  "scopes_supported": [⋯
  ],
  "code_challenge_methods_supported": [
    "S256"
  ],
  "openId": true
}
```

Figure 28: OKTA passive scanning results in config.json

7.3.3 Active Scanning

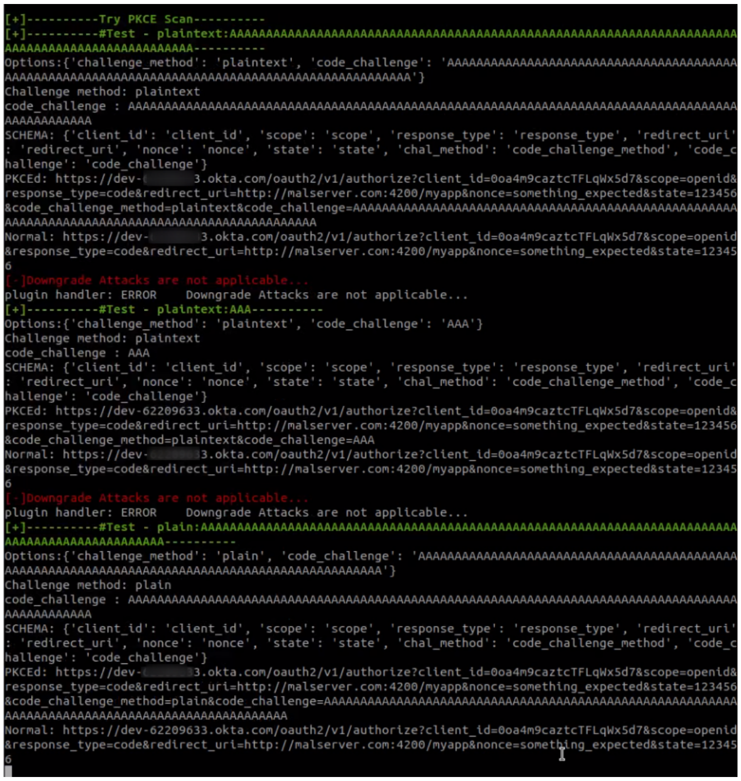
During active scanning, we used all active scanning test suites as they are being presented in the figures 29, 30 and 31. Precisely, the mode that we used to do so was `--navigation=selenium` and therefore we managed to run all the tests. This was mandatory in the case of OKTA as it has already been mentioned in paragraph 6.3.3 in subsection 6.3.3 since the account that was used for the tests to run was an account that had signed up with SSO. Therefore, in order to proceed with the proper authentication the Selenium agent was employed.

The results of the PKCE scan 29 show that the parameter is not supported on the server by the specific configuration that we applied for. It means that the server always applies this technique to secure the authorization code flow and to prevent the attacker from being able to exchange an authorization code for an access token if they somehow manage to intercept the authorization code.

Regarding the rest of the results, as expected, OKTA is not vulnerable to authorization_code replay attacks as it depicted in figure 30. `Nonce` parameter is not properly evaluated in an OpenID Flow, which means that it allows the re-usability of this parameter. Nonce replay refers to the scenario where an attacker intercepts an access token and attempts to use it to gain access to protected resources. Nonce replay can be prevented by using techniques such as nonce validation, in which the authorization server checks to ensure that the nonce (a random value that is included in the authorization request) has not been used before.

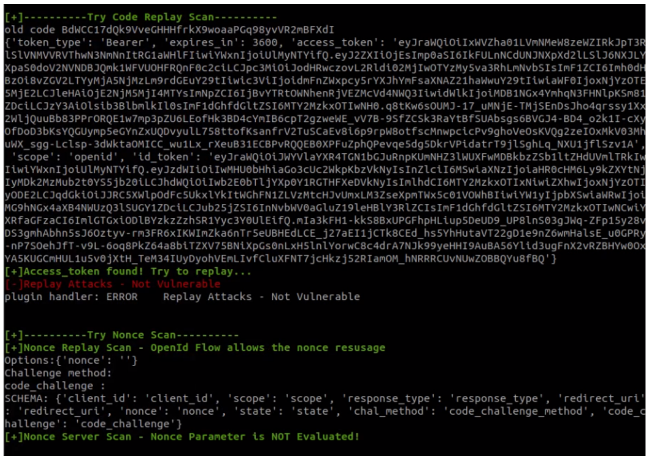
Finally, the results about scope scanning, figure 31, are indicating that the server is not vulnerable to random scope attacks. Successfully, the scanner identified a client id which was discovered by using a dictionary which contains the client id that is presented in figure 31. Also, as expected we have not obtained any result about open redirect scan since it requires the usage of cookies, which was unachievable to be obtained from Selenium's session.

The results presented in subsection 7.3.3 about Active Scanning are summarized in table 5



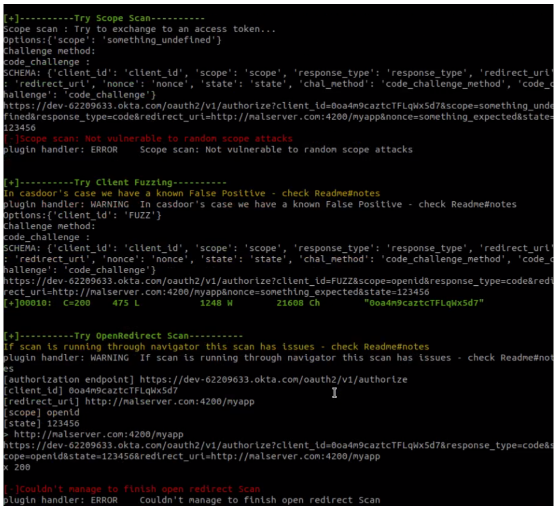
```
[+]----------Try PKCE Scan----------
[+]----------#Test - plaintext:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAA----------
Options:{'challenge_method': 'plaintext', 'code_challenge': 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'}
Challenge method: plaintext
code_challenge : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA
SCHEMA: {'client_id': 'client_id', 'scope': 'scope', 'response_type': 'response_type', 'redirect_uri'
: 'redirect_uri', 'nonce': 'nonce', 'state': 'state', 'chal_method': 'code_challenge_method', 'code_c
hallenge': 'code_challenge'}
PKCEd: https://dev-[redacted]3.okta.com/oauth2/v1/authorize?client_id=0oa4m9caztcTFLqWx5d7&scope=openid&
response_type=code&redirect_uri=http://malserver.com:4200/myapp&nonce=something_expected&state=123456
&code_challenge_method=plaintext&code_challenge=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Normal: https://dev-[redacted]3.okta.com/oauth2/v1/authorize?client_id=0oa4m9caztcTFLqWx5d7&scope=openid
&response_type=code&redirect_uri=http://malserver.com:4200/myapp&nonce=something_expected&state=12345
6
[-]Downgrade Attacks are not applicable...
plugin handler: ERROR    Downgrade Attacks are not applicable...
[+]----------#Test - plaintext:AAA----------
Options:{'challenge_method': 'plaintext', 'code_challenge': 'AAA'}
Challenge method: plaintext
code_challenge : AAA
SCHEMA: {'client_id': 'client_id', 'scope': 'scope', 'response_type': 'response_type', 'redirect_uri'
: 'redirect_uri', 'nonce': 'nonce', 'state': 'state', 'chal_method': 'code_challenge_method', 'code_c
hallenge': 'code_challenge'}
PKCEd: https://dev-62209633.okta.com/oauth2/v1/authorize?client_id=0oa4m9caztcTFLqWx5d7&scope=openid&
response_type=code&redirect_uri=http://malserver.com:4200/myapp&nonce=something_expected&state=123456
&code_challenge_method=plaintext&code_challenge=AAA
Normal: https://dev-[redacted]3.okta.com/oauth2/v1/authorize?client_id=0oa4m9caztcTFLqWx5d7&scope=openid
&response_type=code&redirect_uri=http://malserver.com:4200/myapp&nonce=something_expected&state=12345
6
[-]Downgrade Attacks are not applicable...
plugin handler: ERROR    Downgrade Attacks are not applicable...
[+]----------#Test - plain:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA----------
Options:{'challenge_method': 'plain', 'code_challenge': 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'}
Challenge method: plain
code_challenge : AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA
SCHEMA: {'client_id': 'client_id', 'scope': 'scope', 'response_type': 'response_type', 'redirect_uri'
: 'redirect_uri', 'nonce': 'nonce', 'state': 'state', 'chal_method': 'code_challenge_method', 'code_c
hallenge': 'code_challenge'}
PKCEd: https://dev-[redacted]3.okta.com/oauth2/v1/authorize?client_id=0oa4m9caztcTFLqWx5d7&scope=openid&
response_type=code&redirect_uri=http://malserver.com:4200/myapp&nonce=something_expected&state=123456
&code_challenge_method=plain&code_challenge=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Normal: https://dev-62209633.okta.com/oauth2/v1/authorize?client_id=0oa4m9caztcTFLqWx5d7&scope=openid
&response_type=code&redirect_uri=http://malserver.com:4200/myapp&nonce=something_expected&state=12345
6
```

Figure 29: OKTA's PKCE scanning results



```
[+]----------Try Code Replay Scan----------
old code BdWCC17dQk9VveGHHHfrkX9woaaPGq98yvVR2mBFXdI
{'token_type': 'Bearer', 'expires_in': 3600, 'access_token': 'eyJraWQiOiIxWVZha01LVmNMeW8zeWZIRkJpT3R
lSlVNMVVRVThwN3NmNnItRG1aWHlFIiwiYWxnIjoiUlMyNTYifQ.eyJ2ZXIiOjEsImp0aSI6IkFULnNCdUNJNXpXd2lLSlJ6NXJLY
XpaS0doV2NVNDBJQmk1WFVUOHFRQnF0c2ciLCJpc3MiOiJodHRwczovL2Rldi02MjIwOTYzMy5va3RhLmNvbSIsImF1ZCI6Imh0dH
BzOi8vZGV2LTYyMjA5NjMzLm9rdGEuY29tIiwic3ViIjoidmFnZWxpcy5rYXJhYmFsaXNAZ21haWwuY29tIiwiaWF0IjoxNjYzOTE
5MjE2LCJleHAiOjE2NjM5MjI4MTYsImNpZCI6IjBvYTRtOWNhenRjVEZMcVd4NWQ3IiwidWlkIjoiMDB1NGx4YmhqN3FHNlpKSm81
ZDciLCJzY3AiOlsib3BlbmlkIl0sImF1dGhfdGltZSI6MTY2MzkxOTIwNH0.q8tKw6sOUMJ-17_uMNjE-TMjSEnDsJho4qrssy1Xx
2WljQuuBb83PPrORQE1w7mp3pZU6LEofHk3BD4cYmIB6cpT2gzweWE_vV7B-9SfZCSk3RaYtBfSUAbsgs6BVGJ4-BD4_o2k1I-cXy
OfDoD3bKsYQGUymp5eGYnZxUQDvyulL758ttofKsanfrV2TuSCaEv8i6p9rpW8otfscMnwpcicPv9ghoVeOsKVQg2zeIOxMkV03Mh
uWX_sgg-Lclsp-3dWktaOMICC_wu1Lx_rXeuB31ECBPvRQQEB0XPFuZphQPevqe5dg5DkrVPidatrT9jlSghLq_NXU1jflSzv1A',
 'scope': 'openid', 'id_token': 'eyJraWQiOiJWYVlaYXR4TGN1bGJuRnpKUmNHZ3lWUXFwMDBkbzZSb1ltZHdUVmlTRkIw
IiwiYWxnIjoiUlMyNTYifQ.eyJzdWIiOiIwMHU0bHhiaGo3cUc2WkpKbzVkNyIsInZlciI6MSwiaXNzIjoiaHR0cHM6Ly9kZXYtNj
IyMDk2MzMub2t0YS5jb20iLCJhdWQiOiIwb2E0bTljYXp0Y1RGTHFXeDVkNyIsImlhdCI6MTY2MzkxOTIxNiwiZXhwIjoxNjYzOTI
yODE2LCJqdGkiOiJJRC5XWlpOdFc5UkxlYkItWGhFN1ZLVzMtcHJvUmxLM3ZseXpmTWx5c01VOWhBIiwiYW1yIjpbXSwiaWRwIjoi
MG9hNGx4aXB4NWUzQ3lSUGY1ZDciLCJub25jZSI6InNvbWV0aGluZ19leHBlY3RlZCIsImF1dGhfdGltZSI6MTY2MzkxOTIwNCwiY
XRfaGFzaCI6ImlGTGxiODlBYzkzZzhSR1Yyc3Y0UlEifQ.mIa3kFH1-kkS8BxUPGFhpHLiup5DeUD9_UP8lnS03gJWq-ZFp15y28v
DS3gmhAbhn5sJ6Oztyv-rm3FR6xIKWImZka6nTr5eUBHEdLCE_j27aEI1jCTk8CEd_hs5YhHutaVT22gD1e9nZ6wmHalsE_u0GPRy
-nP7SOehJfT-v9L-6oq8PkZ64a8biTZXV75BNiXpGs0nLxH5lnlYorwC8c4drA7NJk99yeHHI9AuBA56Ylid3ugFnX2vRZBHYw0Ox
YA5KUGCmHUL1u5v0jXtH_TeM34IUyDyohVEmLIvfCluXFNT7jcHkzj52RIamOM_hNRRRCUvNUwZOBBQYu8fBQ'}
[+]Access_token found! Try to replay...
[-]Replay Attacks - Not Vulnerable
plugin handler: ERROR    Replay Attacks - Not Vulnerable


[+]----------Try Nonce Scan----------
[+]Nonce Replay Scan - OpenId Flow allows the nonce resusage
Options:{'nonce': ''}
Challenge method:
code_challenge :
SCHEMA: {'client_id': 'client_id', 'scope': 'scope', 'response_type': 'response_type', 'redirect_uri'
: 'redirect_uri', 'nonce': 'nonce', 'state': 'state', 'chal_method': 'code_challenge_method', 'code_c
hallenge': 'code_challenge'}
[+]Nonce Server Scan - Nonce Parameter is NOT Evaluated!
```

Figure 30: OKTA's code replay and nonce scanning results



```
[+]----------Try Scope Scan----------
Scope scan : Try to exchange to an access token...
Options:{'scope': 'something_undefined'}
Challenge method:
code_challenge :
SCHEMA: {'client_id': 'client_id', 'scope': 'scope', 'response_type': 'response_type', 'redirect_uri'
: 'redirect_uri', 'nonce': 'nonce', 'state': 'state', 'chal_method': 'code_challenge_method', 'code_c
hallenge': 'code_challenge'}
https://dev-62209633.okta.com/oauth2/v1/authorize?client_id=0oa4m9caztcTFLqWx5d7&scope=something_unde
fined&response_type=code&redirect_uri=http://malserver.com:4200/myapp&nonce=something_expected&state=
123456
[-]Scope scan: Not vulnerable to random scope attacks
plugin handler: ERROR    Scope scan: Not vulnerable to random scope attacks


[+]----------Try Client Fuzzing----------
In casdoor's case we have a known False Positive - check Readme#notes
plugin handler: WARNING  In casdoor's case we have a known False Positive - check Readme#notes
Options:{'client_id': 'FUZZ'}
Challenge method:
code_challenge :
SCHEMA: {'client_id': 'client_id', 'scope': 'scope', 'response_type': 'response_type', 'redirect_uri'
: 'redirect_uri', 'nonce': 'nonce', 'state': 'state', 'chal_method': 'code_challenge_method', 'code_c
hallenge': 'code_challenge'}
https://dev-62209633.okta.com/oauth2/v1/authorize?client_id=FUZZ&scope=openid&response_type=code&redi
rect_uri=http://malserver.com:4200/myapp&nonce=something_expected&state=123456
[+]00010:  C=200    475 L          1248 W        21608 Ch        "0oa4m9caztcTFLqWx5d7"


[+]----------Try OpenRedirect Scan----------
If scan is running through navigator this scan has issues - check Readme#notes
plugin handler: WARNING  If scan is running through navigator this scan has issues - check Readme#not
es
[authorization endpoint] https://dev-62209633.okta.com/oauth2/v1/authorize
[client_id] 0oa4m9caztcTFLqWx5d7
[redirect_uri] http://malserver.com:4200/myapp
[scope] openid
[state] 123456
> http://malserver.com:4200/myapp
https://dev-62209633.okta.com/oauth2/v1/authorize?client_id=0oa4m9caztcTFLqWx5d7&response_type=code&s
cope=openid&state=123456&redirect_uri=http://malserver.com:4200/myapp
x 200

[-]Couldn't manage to finish open redirect Scan
plugin handler: ERROR    Couldn't manage to finish open redirect Scan
```

Figure 31: OKTA's client fuzzing, scope scanning and open redirect results

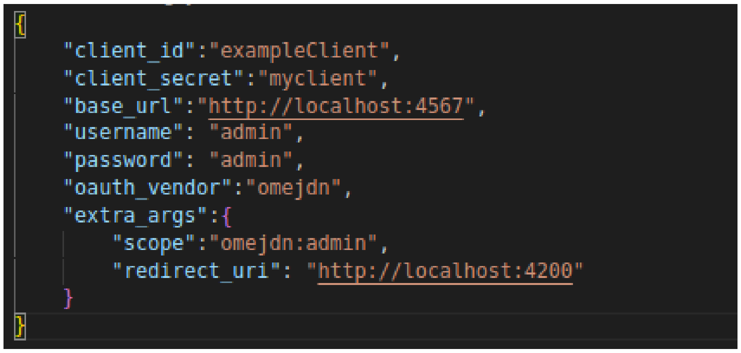
Scan Suite	Results
PKCE Scan	Not Vulnerable
Auth Replay Scan	Not Vulnerable
Nonce Scan	Vulnerable
Scope Scan	Not Vulnerable
Client Fuzzing	Detected 1 client
Open Redirect	Not Supported

Table 5: Active scanning for OKTA

7.4 The rest of the Servers

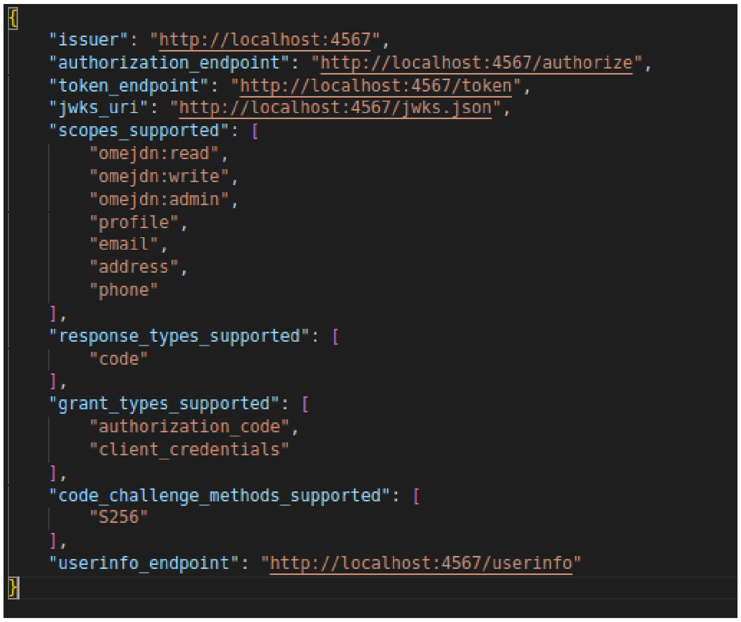
Regarding the rest of the servers that we take into consideration to build our framework we notice different cases. Despite the fact that the servers helped our research, because of the common configurations that are used also in these cases, we were unable to create patterns or to fully cover these cases.

Both Omejdn and A12n support ".well-known" endpoint which is part of best practices as it is described in the relevant RFC for the OAuth servers. However, it is a feature that as we noticed exists only in the servers that support OpenID Connect as well as the ones that thoroughly discussed in subsections 7.1, 7.2 and 7.3. The other one, namely Glewlwyd, does not support it because it also does not support OIDC. Therefore, it is impossible for our tool to proceed further with the enumeration in this case and we have to proceed manually to the tests that we have to conduct. Below, we present indicatively the configuration for Omejdn in figure 32 and the results of the scanner for the passive scans for all these servers in figures 33, 34 and 35.



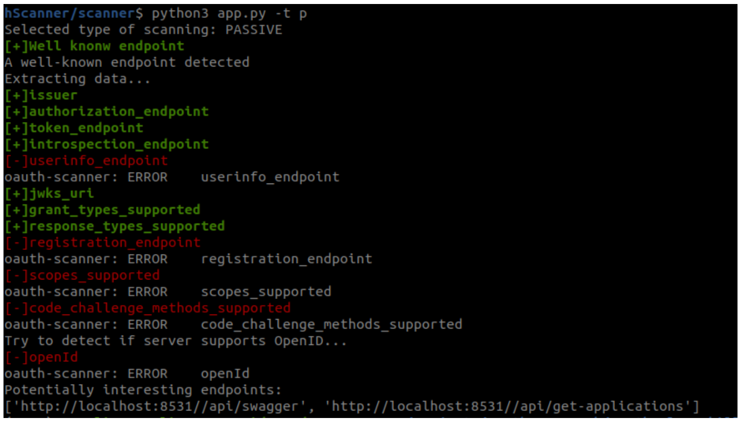
```
{
    "client_id":"exampleClient",
    "client_secret":"myclient",
    "base_url":"http://localhost:4567",
    "username": "admin",
    "password": "admin",
    "oauth_vendor":"omejdn",
    "extra_args":{
        "scope":"omejdn:admin",
        "redirect_uri": "http://localhost:4200"
    }
}
```

Figure 32: Omejdn - Config of the Tool



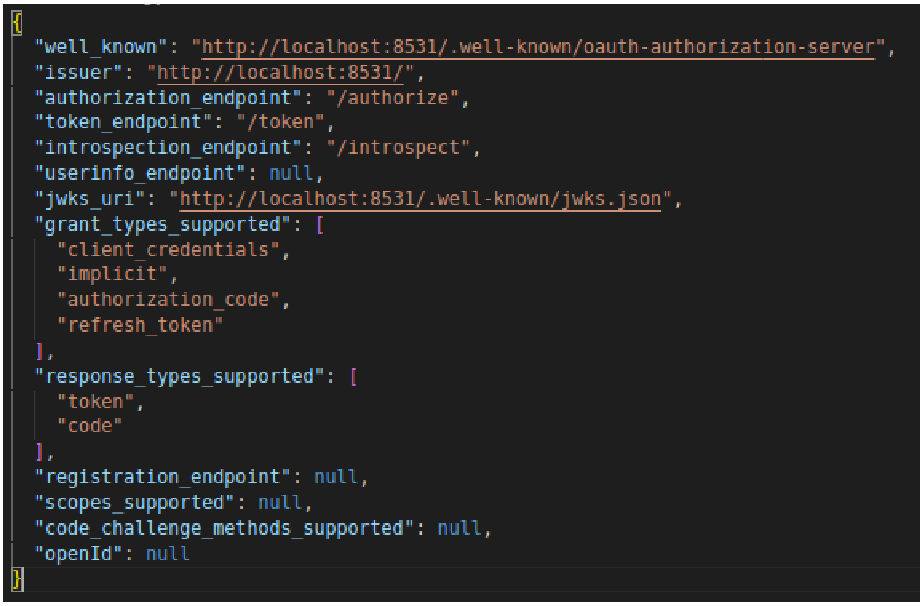
```
{
    "issuer": "http://localhost:4567",
    "authorization_endpoint": "http://localhost:4567/authorize",
    "token_endpoint": "http://localhost:4567/token",
    "jwks_uri": "http://localhost:4567/jwks.json",
    "scopes_supported": [
        "omejdn:read",
        "omejdn:write",
        "omejdn:admin",
        "profile",
        "email",
        "address",
        "phone"
    ],
    "response_types_supported": [
        "code"
    ],
    "grant_types_supported": [
        "authorization_code",
        "client_credentials"
    ],
    "code_challenge_methods_supported": [
        "S256"
    ],
    "userinfo_endpoint": "http://localhost:4567/userinfo"
}
```

Figure 33: Omejdn - Results from passive scan



```
hScanner/scanner$ python3 app.py -t p
Selected type of scanning: PASSIVE
[+]Well knonw endpoint
A well-known endpoint detected
Extracting data...
[+]issuer
[+]authorization_endpoint
[+]token_endpoint
[+]introspection_endpoint
[-]userinfo_endpoint
oauth-scanner: ERROR    userinfo_endpoint
[+]jwks_uri
[+]grant_types_supported
[+]response_types_supported
[-]registration_endpoint
oauth-scanner: ERROR    registration_endpoint
[-]scopes_supported
oauth-scanner: ERROR    scopes_supported
[-]code_challenge_methods_supported
oauth-scanner: ERROR    code_challenge_methods_supported
Try to detect if server supports OpenID...
[-]openId
oauth-scanner: ERROR    openId
Potentially interesting endpoints:
['http://localhost:8531//api/swagger', 'http://localhost:8531//api/get-applications']
```

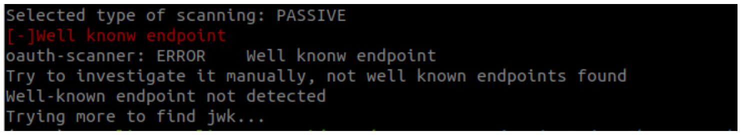
Figure 34: A12 - Results from CLI



```
{
  "well_known": "http://localhost:8531/.well-known/oauth-authorization-server",
  "issuer": "http://localhost:8531/",
  "authorization_endpoint": "/authorize",
  "token_endpoint": "/token",
  "introspection_endpoint": "/introspect",
  "userinfo_endpoint": null,
  "jwks_uri": "http://localhost:8531/.well-known/jwks.json",
  "grant_types_supported": [
    "client_credentials",
    "implicit",
    "authorization_code",
    "refresh_token"
  ],
  "response_types_supported": [
    "token",
    "code"
  ],
  "registration_endpoint": null,
  "scopes_supported": null,
  "code_challenge_methods_supported": null,
  "openId": null
}
```

Figure 35: A12 - Results from passive scan

As we notice from figure 36, Glewlyd did not return the expected results and the reason as explained above is that it does not support OIDC. As a result, our tool was unable to enumerate its case.



```
Selected type of scanning: PASSIVE
[-]Well knonw endpoint
oauth-scanner: ERROR    Well knonw endpoint
Try to investigate it manually, not well known endpoints found
Well-known endpoint not detected
Trying more to find jwk...
```

Figure 36: Glewlyd - Failed to get results from passive scan

8 Conclusions and Future Work

The research on potential vulnerabilities in OAuth servers revealed that all of the tested servers had misconfigurations that could potentially lead to threats. In the tests, the three most popular OAuth servers were scanned with a defined configuration for each of them, and vulnerabilities were found in all of them. These vulnerabilities included a lack of evaluation of the nonce parameter and a lack of PKCE implementation, which made the two open source implementations of OAuth potentially vulnerable to code replay attacks. All of the servers were also potentially vulnerable to enumeration of clients or exposing endpoints directly.

Overall, the research done in the context of this Master thesis highlights the importance of following best practices for OAuth to avoid potential vulnerabilities. It is recommended that anyone implementing or configuring OAuth servers use a vulnerability scanner to detect potential vulnerabilities in their systems and take steps to address them.

As a suggestion for future work:

1. A simple and effective technique could be to change the response_type from “code” to “token”, and test if the implicit flow is supported. By doing this impersonation attack, the opponent can directly get access_token and bypass any code injection mitigation.

2. For implicit grant case, another technique could be to inject the state variable. There are misunderstandings among devs and security researchers that the session-bound state variable can prevent code injection attacks. In many cases, the attacker can reuse any state or create a valid session-state pair by intercepting the OAuth authorization request.

3. In cases of auto consent mechanism that grants permission for authorization automatically after the first time, giving attackers the ability to perform CSRF style stealthy attack. A stealthiest technique should be creating images pointing to the constructed OAuth authorization URL.

4. Relevant to the current implemented scans would be the extension of scope testing. Its implementation relies on trying the supported scopes only that have been retrieved from the passive scans to identify the case that the server blocks completely irrelevant values that could be used as input, but not doing the same about supported values.

5. In our implementation, we check if the auth code is actually one time token or not. Future work could also cover other requirements for these tokens. According to the best practices, a token has to be invalidated after a certain and short period of time since they are intended to be used as tokens from an endpoint and not from a user. In this respect, their lifespan should be short, otherwise it could broaden the attack window of someone that potentially could find such tokens leaked.

6. A potential feature for a scanner to detect misconfigurations on acr or amr values might be the ability to validate the authenticity and integrity of the authentication methods and business rules specified in these parameters. This could involve checking for incorrect or unexpected values, such as attempting to bypass two-factor authentication as mentioned in [14].

   The scanner could also potentially check for other misconfigurations or vulnerabilities related to acr and amr values, such as checking for incorrect or inconsistent values between the two parameters, or checking for missing or incomplete values that may result in incomplete or insufficient authentication.

   Overall, the goal of this feature would be to help ensure that the authentication process is properly configured and secure, and to identify and alert on any potential misconfigurations or vulnerabilities that could compromise the authenticity and integrity of the authentication process.

In conclusion, the research done in the context of this thesis is an important contribution to the field, and we hope it will be extended and reviewed by other contributors to further improve our understanding of OAuth vulnerabilities.

References

[1] Slack disclosed on hackerone: Slack oauth2 redirect_uri bypass. https://hackerone.com/reports/2575/.

[2] Twitter disclosed on hackerone: Insufficient oauth callback. https://hackerone.com/reports/110293.

[3] Akamai. The latest state of the internet around apis. https://www.akamai.com/content/dam/site/en/documents/state-of-the-internet/soti-security-api-the-attack-surface-that-connects-us-all.pdf.

[4] Bleepingcomputer. Microsoft: Exchange servers hacked via oauth apps for phishing.

[5] K Buyens. Oauth 2.0 security cheat sheet. https://github.com/koenbuyens/oauth-2.0-security-cheat-sheet, 2022.

[6] M. Rupp Dr. N. Kobeissi BSc. C. Kean MSc. S. Moritz B. Walny BSc. T.-C. Hong Cure53, Dr.-Ing. M. Heiderich. Pentest-report keycloak 8.0 audit pentest. Technical report, cure53, 2019.

[7] datatracker.ietf.org. Rfc 6819 - oauth 2.0 threat model and security considerations. https://datatracker.ietf.org/doc/html/rfc6819.

[8] datatracker.ietf.org. rfc6749 - the oauth 2.0 authorization framework. https://datatracker.ietf.org/doc/html/rfc6749.

[9] datatracker.ietf.org. rfc7636 - proof key for code exchange by oauth public clients. https://datatracker.ietf.org/doc/html/rfc7636.

[10] Okta Developer. What is the oauth 2.0 authorization code grant type.

[11] Okta Developer. What is the oauth 2.0 implicit grant type. https://developer.okta.com/blog/2018/05/24/what-is-the-oauth2-implicit-grant-type#get-the-users-permission.

[12] OpenID Connect OAuth Server dédié. Why an authentication server?

[13] Daniel Fett, Ralf Küsters, and Guido Schmitz. A comprehensive formal security analysis of oauth 2.0. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 10 2016.

[14] Gitbook. Acr and amr misconfigurations. `https://0xn3va.gitbook.io/cheat-sheets/web-application/oauth-2.0-vulnerabilities#misconfiguration-acr-or-amr`, 2021.

[15] Gitbook. Oauth 2.0 vulnerabilities - cheat-sheets. `https://0xn3va.gitbook.io/cheat-sheets/web-application/oauth-2.0-vulnerabilities#security-issues-in-the-authorization-server`, 2021.

[16] HackerNews. Nearly 100,000 npm users' credentials stolen in github oauth breach.

[17] Hackerone. Twitter disclosed on hackerone. `https://hackerone.com/reports/131202`.

[18] Torsten Lodderstedt, John Bradley, Andrey Labunets, and Daniel Fett. OAuth 2.0 Security Best Current Practice. Internet-Draft draft-ietf-oauth-security-topics-21, Internet Engineering Task Force, September 2022. Work in Progress.

[19] mastercard developers. Mastercard developers.

[20] OAuth.net. Specs - oauth. `https://oauth.net/specs/`.

[21] A Parecki. *Oauth 2.0 Simplified*. Lulu.com, 2018.

[22] SaneBow. redirect-fuzzer. `https://github.com/SaneBow/redirect-fuzzer`, 2019.

[23] Security_Hubs. Threat model pentesting checklist. `https://web.archive.org/web/20220919040023/https://securityhubs.io/oauth2_threat_model`.

[24] Six2dez.com. Oauth - pentest book. `https://pentestbook.six2dez.com/enumeration/webservices/oauth`, 2014.

[25] vagelkara. Oauthvulnerabilityscanner. `https://github.com/vagelkara/OAuthVulnerabilityScanner`, 2022.

[26] vagelkara. Oauthvulnerabilityscanner - resources, 2022.

[27] X Wang, W Lau, R Yang, and S Shi. Make redirection evil. `https://i.blackhat.com/asia-19/Fri-March-29/bh-asia-Wang-Make-Redirection-Evil-Again-wp.pdf`.

[28] P Yaworski. *Real-world bug hunting : a field guide to web hacking.* No Starch Press, San Francisco, 2019.