UNIVERSITÀ
DI TRENTO

Department of Information Engineering and Computer Science

Bachelor's Degree in
Computer Science

Final Dissertation

Design and implementation of an attack pattern language for the automated pentesting of OAuth/OIDC deployments

Supervisor

Prof. Silvio Ranise

Student

Wendy Briggit Flora
Barreto Flores

Co-Supervisors

Dott. Andrea Bisegna

Dr. Roberto Carbone

Academic year 2019/2020

Acknowledgements

I would like to express my gratitude to Prof. Silvio Ranise,
for giving me the opportunity to work on this project.

I would like to offer my sincere thanks to Andrea and Roberto,
for their support, patience and availability.

I would like to thank my dearest friends,
for their encouragement.

I would like to dedicate this thesis to my family,
for always supporting me.

Contents

Abstract

Governments are increasingly introducing their citizens to the use of digital identities, in order to provide them with high-quality services. As a consequence the management of the digital identity is an important issue in the Cybersecurity field.

In this thesis we focus on identity management (IdM) protocols, whose aim is to ensure that only authenticated and authorized users have access to protected services. In particular, a widely adopted IdM protocol is OAuth 2.0 (hereafter OAuth). It is an authorization framework, that manages the access to an Hypertext Transfer Protocol (HTTP) service and allows to access it using external credentials. The OpenID Connect (OIDC) protocol is built on top of OAuth, it manages authentication and allows to implement Single Sign-On (SSO). SSO is an authentication scheme that allows a user to access several services with a single set of credentials and delegates the identity management to a third party.

On May 2020, researchers discovered an attack exploiting OAuth to access Microsoft Office 365, a set of cloud based applications offered through a subscription. The attacker was able to access all the services in the suite, bypass the multi factor authentication and steal victims' data [12, 1]. Therefore, it is crucial to prevent that sensitive data falls into the wrong hands. To avoid that, it is necessary to employ tools able to spot vulnerabilities on IdM protocols. To perform these tests some tools are available, but the specification of the tests is hard-coded inside them. This is, for instance, the case of Micro-Id-Gym OAuth/OIDC, a pentesting tool for OAuth and OIDC deployments. The tool provides built-in tests, executes them and automates the login process using an input trace. The tests are hard-coded inside the plugin, therefore the definition of new tests requires the implementation of a new plugin. The goal of this thesis is to design a language that allows the user to define tests in a more natural way. We want to be able to specify tests without having to modify the code of the plugin, as this procedure could be complex and lead to errors. Our solution is a new language that is flexible and simplifies the definition of future new tests.

The starting point to design our language was a careful analysis of the security tests available in the plugin. The language exploits regular expressions, as they are a powerful tool used for search pattern, which can be applied to HTTP messages, and uses the JavaScript Object Notation (JSON) to define tests, as it allows to build complex structures using a straightforward syntax. The attack pattern language has been implemented in a plugin for Burp, using Java as programming language. In this implementation there are three main classes: Test Suite, Test and Operation. Test Suite is a set of Tests, that can be active or passive. Test consists of a set of Operations. The Operation class is the core of the language and the most complex one. Operation's main function is the search of a regex pattern inside HTTP messages exchanged during the SSO process. Operation has been designed to allow the user to insert an input, that becomes a regex pattern to search inside a HTTP message, and to choose where to apply the pattern. Operation also allows the modification of a HTTP message after executing a search. To validate the language, some of the tests hard-coded inside the Micro-Id-Gym OAuth/OIDC plugin have been re-defined using the new language and new tests have been specified. In both cases the definition was simpler and faster, and the results obtained from the execution of the tests were the same obtained with the Micro-Id-Gym OAuth/OIDC plugin.

The main contributions of the thesis are: The analysis of the Micro-Id-Gym OAuth/OIDC plugin used to execute passive and active tests, along with a detailed description of the design choices for the final product; The design of a language that will allow a formal specification of currently available and future tests. It will provide flexibility in the definition of tests and a format that is simple and natural for the user; The implementation of a plugin for Burp that translates the tests written using the language into code that executes the tests and a Test Suite written using the new language, that contains all the tests implemented inside Micro-Id-Gym OAuth/OIDC. This will be used to execute a vulnerability assessment of different services.

1 Introduction

1.1 Context and Motivation

Nowadays the Single Sign-On (SSO) authentication is widely available. Indeed most of the websites require some sort of authentication in order to access the content and services. SSO is an authentication scheme that allows users to sign into several applications with a single set of credentials. It can be implemented using the OAuth authorization framework but it is also possible to use different protocols. Identity management protocols' aim is to ensure that only authenticated and authorized users have access to protected services. These protocols can be vulnerable to attacks and in order to prevent them, it is necessary to identify their vulnerabilities. Once the weaknesses are found, it is possible to secure the process. The vulnerabilities are found through tests, that can be active or passive: active tests require to actively engage with the system and passive tests require a scan of the system.

In the final dissertation for the Bachelor's Degree in Computer Science of Claudio Grisenti [7], a plugin called Micro-Id-Gym OAuth/OIDC has been carefully developed. This plugin executes a set of tests, both active and passive, exploiting the tools provided by Burp Suite (hereafter Burp). As a matter of fact the plugin requires to be uploaded to Burp, a cybersecurity software that allows to define and execute tests on web applications. However the specification of the tests is to be found inside the code of the plugin. New tests can be created, but it will require the implementation of a new plugin.

1.2 Problem

In the modern world, the testing phase is an important part of the development of flawless systems. The testing process requires to take into consideration every possible scenario. As a consequence tests need to be varied and flexible.

In the case of Micro-Id-Gym OAuth/OIDC, it does not provide a specific approach to define new tests or modify an existing test. The tests provided by the plugin have been implemented inside the plugin, which is written using Java. The definition and declaration of every single test is entirely hard-coded inside the plugin, in other words, it is not possible to alter the data or parameters without modifying the code of the plugin. As a consequence, the Penetration Tester (pentester), whose job is to identify the vulnerabilities of an application, needs to modify the code of the plugin in order to alter or create a test. This process requires a lot of effort, can be complex and lead to multiple errors.

In this thesis we aim to provide a language that will allow to specify a test with more freedom and in a more natural way. This language requires simplicity, the ability of generalization and at the same time it needs provide the possibility to create complex structures. The tests defined in this language will be flexible and entirely modifiable by the pentester. The language developed will also provide a Test Suite, a container of active and passive tests.

1.3 Contributions

The thesis will provide four major contributions:

- An **analysis** of Micro-Id-Gym OAuth/OIDC plugin that will be exploited in order to create the new language.
- A **natural language** that can be used to define a test. As a result of this, tests can have a formal definition with a single uniform language. This language also allows modification by the user.
- A **plugin for Burp** that translates the natural language written as test case into code to be executed by the plugin, which exploits Burp tools. The plugin will also display the tests' results.

- A **Test Suite** containing all the tests previously implemented in Micro-Id-Gym OAuth/OIDC. It will be used for a vulnerability assessment of different services.

1.4 Structure of the thesis

The thesis is composed by the following sections:

Section 2 presents a brief description of the terminology used in the thesis, the OAuth authorization framework, the Single Sign-On authentication scheme and Burp.

Section 3 contains the analysis of Micro-Id-Gym OAuth/OIDC plugin, which was necessary in order to reach the final result.

Section 4 describes in detail the structure of the language developed, focusing on the design choices.

Section 5 presents the plugin for Burp, describing how it translates the natural language into active and passive tests. There is also a presentation of the current limitations of the language.

Section 6 explains the Test Suite containing passive and active tests that were implemented inside Micro-Id-Gym OAuth/OIDC and describes the result of the vulnerability assessment using the plugin for Burp.

Section 7 shows a final conclusion and future work.

2 Background

In this section, the most important concepts of the thesis would be briefly introduced: the terminology used in this thesis (Section 2.1), the OAuth 2.0 authorization framework (Section 2.3), the Single Sign-On scheme (Section 2.2) and Burp (Section 2.4).

2.1 Terminology

For a better understanding of the terminology used in the thesis, the definitions contained in different RFCs will be quoted. A Request For Comments (RFC) "contains technical and organizational documents about the Internet" [5].

According to RFC 4949 [23]:

- **Authorization** is the "process for granting approval to a system entity to access a system resource";
- **Authentication** is "the process of verifying a claim that a system entity or system resource has a certain attribute value." An authentication process consists of two different steps: "Identification step, presenting the claimed attribute value and verification step, presenting or generating authentication information".
- **Attack** is defined as "An intentional act by which an entity attempts to evade security services and violate the security policy of a system. That is, an actual assault on system security that derives from intelligent threat". According to this definition an attack can be distinguished according to its intent:
  - "An **Active Attack** attempts to alter system resources or affect their operation."
  - "A **Passive Attack** attempts to learn or make use of information from a system but does not affect system resources of that system."

Based on the definition of attack, it is possible to define two type of tests:

- Active Test: checks if the system is vulnerable to an active attack.
- Passive Test: checks if the system is vulnerable to a passive attack.

2.2 Single Sign-On

Single Sign-On (SSO) is an "authentication service that allows the user to access multiple applications using a single set of credentials" [21]. In other words, the user has to remember just one set of credentials to access several different applications, without having to log in and log out every single time. Figure 2.1 can be useful to understand better how SSO works.

The SSO approach main advantages are:

- Better user experience, as "users can move between services securely and uninterrupted without specifying their credentials every time" [13].
- The user's credentials are provided directly to the Identity Provider, not the service application the user is trying to access, "and therefore the credentials cannot be cached by the service" [13].
- Reducing the number of "password recovery" calls, the IT help desk has to deal with.

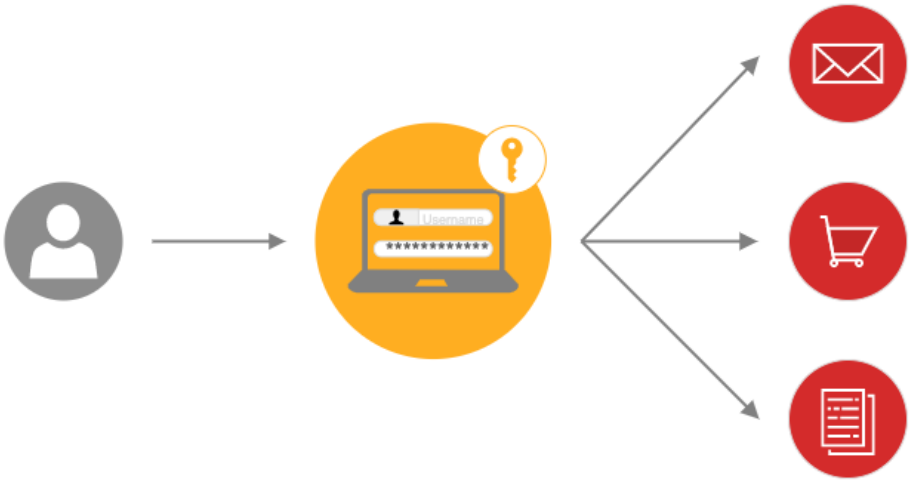


Figure 2.1: SSO: One key gives the access to several application

- Reducing password fatigue from having to remember different user name and password combinations.
- Encourage the use of stronger passwords as just one password has to be remembered.

Nowadays the Single Sign-On mechanism is widely diffused, that is why a better control during the process is needed. It is possible to implement SSO using OAuth for authorization and OIDC for authentication. The focus of the thesis is to find the vulnerabilities inside the process, and so build stronger services.

2.3 OAuth 2.0

OAuth 2.0 (hereafter OAuth) is a widely used authorization framework that "enables a third-party application to obtain limited access to an HTTP service" [9]. It is the mechanism that allows the user to access an application using external credentials. In order to understand how the OAuth framework operates, four important roles need to be introduced. The following definitions can be found in RFC 6749 [9]:

- **Resource Owner**(RO): An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.
- **Resource Server**(RS): The server hosting the protected resources, capable of accepting and responding to protected resource requests using access tokens.
- **Client**(C): An application making protected resource requests on behalf of the resource owner and with its authorization. The term "Client" does not imply any particular implementation characteristics.
- **Authorization Server**(AS): The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.

The OAuth standard document also defines **Access tokens** as "credentials used to access protected resources" [9]. These credentials differ from the user's credentials as it applies only for the specific resource.

The protocol flow of OAuth is described in the same RFC [9] as shown in Figure 2.2:

1. The Client requests authorization from the Resource Owner.

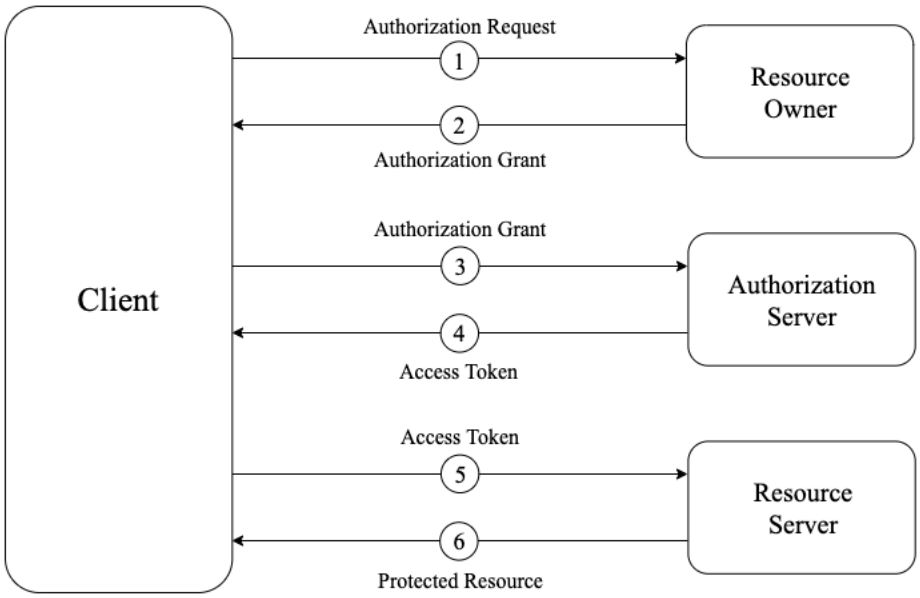


Figure 2.2: Protocol Flow of OAuth 2.0 [9]

2. The Client receives an authorization grant.
3. The Client requests an access token by authenticating with the authorization server and presenting the authorization grant.
4. The Authorization Server authenticates the client and validates the authorization grant, and if valid, issues an access token.
5. The Client requests the protected resource from the Resource Server and authenticates by presenting the access token.
6. The Resource Server validates the access token, and if valid serves the request.

A scenario can be useful to understand better the OAuth framework flow.

Let's assume Alice wants to try this new application that shows the lyrics of the song while playing it. Alice uses Spotify [1] *to listen to music and create playlists. She wants to share her playlist with this application, as she does not want to re-enter all the songs manually. In order to obtain this information, the application redirects her to the Spotify login website, where she needs to enter username and password and decide which playlists she wants to share. After checking her credentials the Spotify website redirects Alice back to the application, where she can finally find her songs and use the application.*

In this scenario it is possible to identify the four main roles previously described:

The **Resource Owner** is Alice as she is the one that can grant or deny access to her playlists; the **Resource Server** is the Spotify database that hosts Alice's data; the **Client** is the application for the lyrics and the **Authorization Server** is the Spotify login website that issues Alice's credentials.

This scenario can be also described using the steps described in the flowchart of OAuth.

1. After Alice clicks "Share your Spotify playlists" inside the application, the application sends and *Authorization Request* to Spotify.

[1] Spotify is an application for media streaming, that employs OAuth for user authorization.

2. The application receives an *Authorization Grant* "which is a credential representing the resource owner's authorization" [9] and redirects Alice to the Spotify login website.

3. Alice enters her username and password in the login website and may see a consent page to choose which playlists to share. The application, during this process, sends the *Authorization Grant* to the *Authorization Server*.

4. The *Authorization Server* behind the login website checks the credentials, validates the *Authorization Grant* and gives the *Access Token*.

5. Now the application asks for the playlists to the Spotify Server using the *Access Token*.

6. The Spotify Server validates the *Access Token* of the application and shares the playlists (*Protected Resource*) with the application.

In this scenario Spotify is a trusted website that hosts the user personal information. What would happen if the new application Alice is trying was not a trusted application and had security issues that could lead to leaks of information? Even worse, what would happen if the new application had bad intentions and wants to obtain Alice's personal data? In this case, several problems could arise: the user card data could be stolen or phishing attacks could be executed, exploiting the user information. Therefore, in order to avoid these situations, the PenTester has the task of identifying vulnerabilities inside the framework.

In order to receive an access token, the Client needs to obtain an authorization from the Resource Owner. This authorization is expressed in the form of an Authorization Grant.

The following notions are needed to understand this process and they are explained in detail in RFC 6749 [9]:

- **Refresh Token**: "Refresh tokens are credentials used to obtain access tokens. They are issued to the client by the Authorization Server and are used to obtain a new access token when the current access token becomes invalid or expires, or to obtain additional access tokens with identical or narrower scope".

- **authorization code**: a bearer credential used by the Client to obtain an access token.

- **client id**: an unique string that identifies the Client, it is issued by the Authorization Server.

- **redirect URI**: it is the URI used by the Authorization Server to redirect the Resource Owner's user-agent back to the Client.

- **state**: a random value used by the Client to maintain state between the request and response.

OAuth defines four grant types: authorization code, implicit, resource owner password credentials, and client credentials. In the Figure 2.3 the Authorization Code Grant flow is described. In Authorization Code Grant "the authorization code is obtained by using the Authorization Server as intermediary between Client and Resource Owner" [9]. In order to obtain an authorization, the Client directs the Resource Owner to the Authorization Server. Here the AS authenticates the RO, if the authorization is successfully obtained the RO goes back to the Client, via redirect URI, with an authorization code. The Client requests the access token from the AS using the authorization code previously obtained. The AS authenticates the client, validates the authorization code and ensures the redirect URI used matches the URI used in the previous step. Finally if both checks are successful the AS issues the access token, or refresh token if requested.

The Authorization Code Grant is often used as it offers a few more benefits that the others. That is why it will be the scheme considered in this thesis. This grant type is used to obtain both access tokens and refresh tokens. During this process the RO authenticates only with the AS, therefore the user's credentials are not shared with the Client. The access token is also directly transmitted from the AS to the Client, without sharing it with the RO's user-agent (web browser). This grant type also provides the ability to authenticate the Client and an almost indefinite access to the resource as it also features refresh tokens.

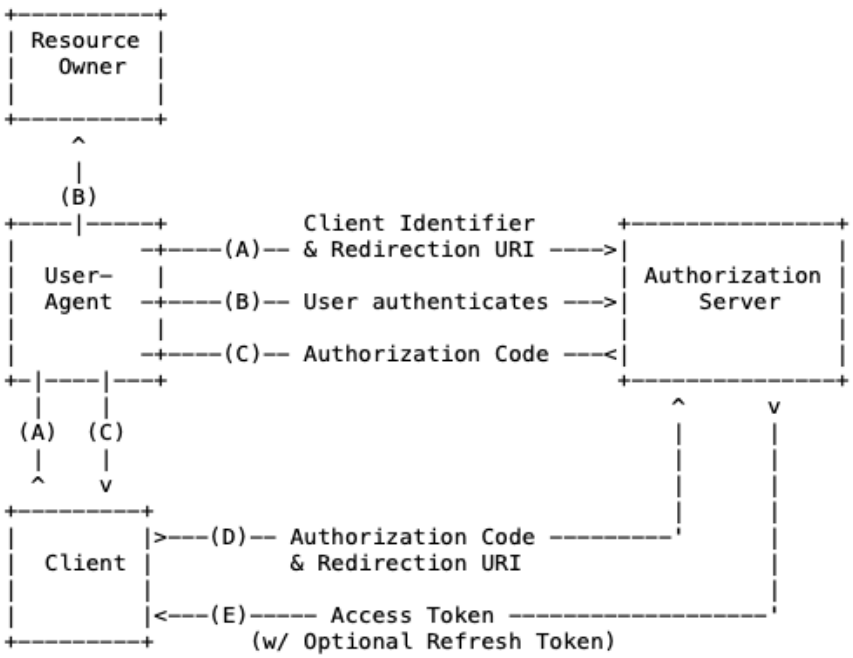


Figure 2.3: OAuth 2.0: Authorization Code Flow [9]

2.4 Burp Suite

Burp Suite Community Edition (hereafter Burp) is a "web application security testing software" [18]. This application is often used for penetration testing and it already contains several useful tools, but it also allows the user to upload inside the software external plugins, which can exploit the main functionalities of the suite.

The Proxy tool is the one useful for this thesis and an example is showed in Figure 2.4:

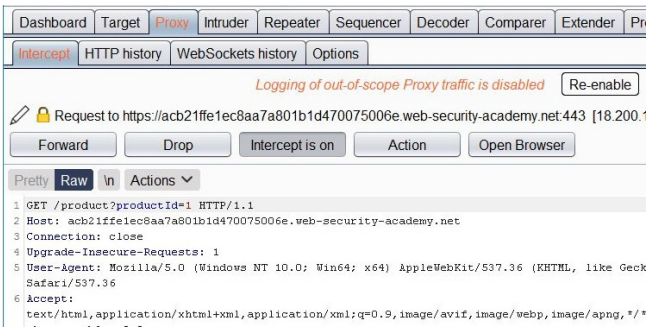


Figure 2.4: Intercepted messages trough Burp Proxy [17]

The software has an intercepting proxy, that allows the user to see and modify the content of request and response messages while they are in transit. It also allows the user to send the request/response back and they can be monitored with another Burp tool.

2.4.1 Request and Response messages

OAuth "enables a third-party application to obtain limited access to an HTTP service". HTTP is the Hypertext Transfer Protocol used to exchange data between a server and a browser. There are two type of messages employed during the exchange:

- **Request message**: A browser wants to provoke an action on the server so it sends a request message.
- **Response message**: After receiving and interpreting a request message, a server responds with an HTTP response message.

According to RFC 7230 HTTP messages "consist of a start-line followed by a sequence of octets in this format: zero or more header fields, an empty line indicating the end of the header section, and an optional message body" [20]. Figure 2.5 sums up better how request and response messages are structured. The designed language is going to exploit this particular structure of the HTTP messages to execute the tests. In every message exchanged a header, a url and a body can be identified. Response messages have status code and request messages can have parameters.

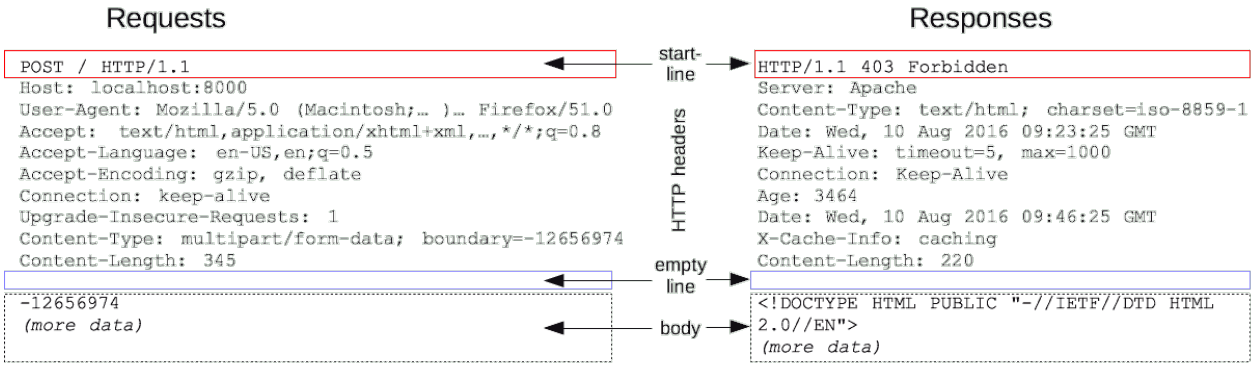


Figure 2.5: HTTP Messages structure [4]

3 Analysis of previous work

3.1 Analysis of "Micro-Id-Gym"

Micro-Id-Gym is a training environment "where users can develop hands-on experiences on how IdM solutions work" [19]. In this context Claudio Grisenti developed a pentesting tool for OAuth and OIDC Deployments [7]. The plugin for Burp implemented, provides a tool capable of working on well known OAuth/OIDC implementations and a set of well-studied test cases. This plugin is written using Java as programming language and exploits the Selenium WebDriver [24], a tool used to automate the interactions between user and web-browser. Given a trace, the tool can automate the steps for the SSO login.

Inside the plugin three different phases can be identified: Test of the trace inserted in the Selenium input field; The execution of several passive tests with a single login with SSO and the execution of a number of active tests, which require the same amount of authentications. The two type of tests run inside the plugin are described in [7] as following: Passive tests "operate in the background while the traffic passes through Burp and do not need to change any HTTP message" and active tests "need to modify specific packets as they pass through Burp which means that each test needs to reiterate the entire authentication". The traffic is intercepted thanks to the proxy tool provided by Burp, which works as an intermediary between two entities and is able to provide us with the request and response messages exchanged. The implementation of the tests inside Micro-Id-Gym OAuth/OIDC exploits these intercepted messages to execute the tests.

As declared in Section 1, the aim of this work is to develop a language that has the ability to re-write previously defined tests and allow to write future tests in a more natural way, and so improve the user experience and provide a flexible language for future tests. The analysis phase started from the code of this plugin, every test was hard-coded inside the plugin and the goal was to find what functionalities needed to be abstracted in the final product. It is possible to divide the analysis phases in two parts: analysis of passive tests and analysis of active tests.

For passive tests, what the code does is search inside the traffic of packets, for messages containing certain strings. Analysing all the passive tests implemented inside the plugin, it was possible to define a certain sequence of operations for every passive test:

1. Run a test using Selenium WebDriver to automate the steps.

2. The plugin saves the traffic.

3. The plugin analyses the traffic searching for messages with certain characteristics.

4. If a message is found then the test succeeded.

   For example if we want to check if the trace given to selenium, employs Hyper Text Transfer Protocol Secure(Https) instead of Http, that lacks the security level that Https is able to provide: The plugin checks if there is any packet in the traffic that uses http/does not use https.

From the analysis it was possible to define that every passive test looks for a message with a certain pattern inside. The type of pattern that is searched depends on the type of tests, the user wants to execute. The messages exchanged are HTTP messages that the Burp Suite is able to intercept and forward to the plugin. If the HTTP message is captured and then extracted as a text, and the characteristics can be written as patterns to find inside the message, it is possible to use a mechanism of pattern search.

For active tests, the flow of operations done by the plugin is completely different but similar at the same time. As said, it needs to re-execute the authentication process as it needs to perform an action on the packets. This action consists in the modification of a certain message, which is identified by the plugin. This identification of the message is a process similar to what the plugin does for the passive tests:

1. Reiterate the test using Selenium WebDriver to automate the steps.
2. The plugin reads the traffic messages.
3. The plugin checks every single message looking for one with certain characteristics.
4. If it finds the message, a modification is done.
5. Otherwise, it proceeds to the next message of the traffic and repeat the process from point 3.
6. The test ends if the trace of Selenium ends or the page found is an error page.

Similarly to the conclusions extracted from the passive tests, it is possible to apply a search pattern to every single message exchanged during the active test.

3.2 Conclusions of the analysis

In the previous section, some conclusions were revealed: the base method that the language needs, is the ability to search patterns inside a HTTP message. It is possible to exploit a mechanism widely used to search patterns inside a text, Regular Expressions (regex).

> *For Example, if we want to check if any packet is not using https, a protocol that provides a security layer, we look for the pattern* `https:` *inside the url field of the messages, that means "look for a string starting with* `https:`*". If this pattern is applied to every message, the failure of the test is determined by finding a message that does not start with* `https:` *so is not using https, in other words the search pattern inside the message fails.*

As a result the language designed needs to define a test as a set of searches that are done inside HTTP messages, and this search pattern needs to be specified by the user. As shown in the previous example, a search also needs to be given a part of the message where to be applied. The message's parts that can be extracted and used are: url, header, entire message, parameters and status code. Every tests needs different type of searches, the example needed to search inside the url of the exchanged messages, but it is possible that certain tests are looking for patterns inside the headers and also inside the url. Therefore the final product needs to let the user specify, in which part of the message to apply the search pattern.

Certain tests would require to apply the search pattern only to responses messages inside the traffic. As a consequence, the language needs to provide a field where to specify the type of messages in which the pattern can be searched. The types of messages found inside the tests of the plugin are: all the messages exchanged, only request messages, only response messages or a certain type of message like Authorization grant message.

4 Design of the new language

4.1 Introduction of Regex

Regular Expressions (regex) "provide a mechanism to select specific strings from a set of character strings" [8]. In other words it allows to find specific sequence of characters inside a text. A Regular Expression can be as simple as "*any string to search*", but it can also allow to define patterns to search inside a text.

> For example, the regex `the` applied to this thesis would match all the substring containing the string `the`: *"thesis", "the", "these", "there",* and many more.
>
> In the case that we are looking for all the lines starting with the article `the` inside this thesis, then we should introduce a special character: `^`
>
> The regex `^the` would match only the strings where `the` can be found at the beginning of the line. In other words it finds all the lines that start with `the`.

There are several different commands like `^`, regular expressions have their own syntax and the user must understand it before starting to use it. Thanks to these special commands, regex can be employed to define and create patterns. Usually regex patterns are used by string searching algorithms for "find" or "find and replace" operations on strings, or for input validation.

The Table 4.1 provides the user with the basic characters used to define patterns in regex:

Table 4.1: Regular Expressions Basic Syntax [11, 6]

Characters	**Function**
`^the`	Matches any string that begins with `the`
`end$`	Matches any string that ends with `end`
`^the end$`	Matches any string beginning with `the` and ending with `end`
`thesis`	Matches any string that contains the word `thesis`
`abc*`	Matches a string that contains `ab` followed by zero or more `c`
`abc+`	Matches a string that has `ab` followed by one or more `c`
`abc?`	Matches a string that has `ab` followed by zero or one `c`
`abc{2}`	Matches a string that has `ab` followed by 2 `c`
`abc{2,}`	Matches a string that has `ab` followed by 2 or more `c`
`abc{2,5}`	Matches a string that has `ab` followed by 2 up to 5 `c`
`a(bc)*`	Matches a string that has a followed by zero or more copies of the sequence `bc`
`a(bc)2,5`	Matches a string that has a followed by 2 up to 5 copies of the sequence `bc`
`a(b|c)`	Matches a string that contains an `a` followed by `b` or `c` and captures `b` or `c`
`a[bc]`	As previous, but it does not capture `b` or `c`
`\d`	Matches a single character that is a digit
`\w`	Matches a word: letters, digits and _
`\s`	Matches a whitespace character, including tabs and line breaks)
`.`	Matches any character
`\D`	Matches a single non-digit character
`[abc]`	Matches a string that has either an `a` or `b` or a `c`

In order to define the tests developed inside "Micro-Id-Gym OAuth/OIDC", it is necessary to create complex regular expressions that can match what the different tests require. The regex used for the tests are a composition of the following regular expressions. The Table 4.2 provides the basic logical operators in regex, to use inside our plugin.

Table 4.2: Useful Regular Expressions for the Plugin

Regular Expression	**Function**
`word`	Matches the messages containing the string `word`
`.*word.*`	Matches the messages containing the string `word`
`^((?!word).)*$`	Matches the messages not containing the string `word`
`word 1|word 2`	Matches the messages containing the string `word 1` or string `word 2`
`(?=.*word 1)(?=.*word 2)`	Matches the messages containing both the strings `word 1` and `word 2`

The plugin developed is written using Java and it is possible to use a package with the class Pattern that provides the necessary methods to apply search patterns to a text in Java [15].

4.2 JSON based language

For the purpose to express the tests in a simple but manageable way, the JSON format was selected. The JSON syntax is indeed simple and clean but it also allows to design sophisticated structures.

JavaScript Object Notation (JSON) is a format commonly used to exchange data between client-server in applications. According to ECMA-404 [10] "JSON defines a small set of structuring rules for the portable representation of structured data".

The JSON main component is the pair "name":"value". Every name-value pair needs to be separated by a comma.

- **JSON names** can be any valid string.
- **JSON values** can be an object, array, number, string, true, false, or null.

JSON Arrays and Objects have their own syntax. It is also possible to nest JSON Objects to create more complex structures.

- A **JSON Object** is defined using two brackets, between them there can be zero or more name-value pairs.
- A **JSON Array** is defined using two square brackets, between them there can be zero or more name-value pairs.

It is possible to see the standard used to define tests in our language in Listing 4.1.

Listing 4.1: Test Suite standard using JSON

```
{
  "Test Suite": {
    "Name": "Test Suite 1 ",
    "Description": "Description of the Test Suite"
  },

  "Tests": [
    {
      "Test": {
        "Name": "Name of the single Test",
        "Description": "Description of the test"
      },

      "Operations": [
        {
          "Action": "Action to do",

          "Keyword": "String to search",
          "Message Type": "Message",
          "Message Section": "Part of the message",
        },
        {
          "Action": "Action to do",

          "Keyword": "String to search",
          "Message Type": "Message",
          "Message Section": "Part of the message",

        }
        ...
      ],

      "Columns": [
        {
          "header": "Header 1 of the Result Table of the single Test",
          "value": "Access to a Test or a Message value"
        },
        {
          "header": "Header 2 of the Result Table of the single Test",
          "value": "Test.name"
        },
        {
          "header": "Header 3 of the Result Table of the single Test",
          "value": "Message.url"
        }
        ...
      ]
    }
    ...
  ]
}
```

4.3 Structure of the language

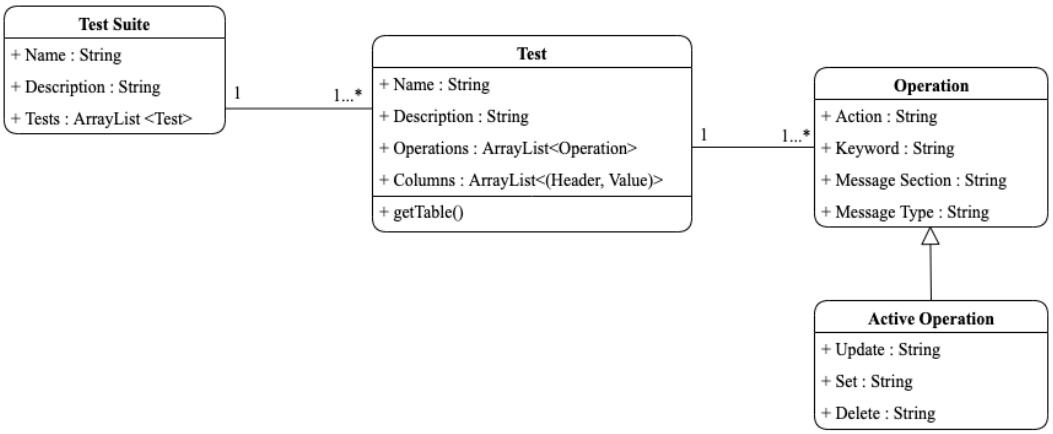


Figure 4.1: Class Diagram of the language

The structure of the language can be easily identified inside the JSON file, but the class diagram in Figure 4.1 shows better the three important components of the language: Test Suite, Test and Operation.

4.3.1 Test Suite

It is possible to define **Test Suite** object as a set of tests that the user wants to execute. Every *Test Suite* has two attributes, "Name" and "Description", and they can be any string the user wants to insert. A *Test Suite* object contains also a set of tests to be executed. This set is defined using a JSON Array, that is a set of the *Test* objects (to be described in the following section). It is possible to execute a *Test Suite* containing both passive and active tests.

As defined in Section 2.1, active and passive test are different. An active test performs an action during the exchange of messages and a passive test does not need to perform an action but checks the messages exchanged.

4.3.2 Test

Similarly to Test Suite, the **Test** object can be defined as a set of Operations the user wants a test to perform. Every *Test* has a name and a description, that can help the pentester to organize or briefly describe the test in execution and, like in the Test Suite object, every string value is accepted in these two fields. In addition, every *Test* has a table that shows the result of a test executed. This table is described in the JSON file as an array of columns. Every column is described as an object composed by two pairs, where the JSON names are "header" and "value". The user is asked to insert respectively as JSON values; the string to be used as header of the column and the value the user wants to be shown in the result table.

4.3.3 Operation

The **Operation** object is the most complex object, therefore it is the main component of the structure. The Operation class is defined by four JSON names: Action, Keyword, Message Type and Message Section. For active tests three more JSON names are needed: Update, Set and Delete. So a single *Operation* can be:

- Passive: It checks a single message and verifies that the "Keyword" value can be found in the "Message Section" value. The "Message Type" value defines what type of message needs to be checked and intercepted.
  - Action: It has to be set to Passive.
  - Keyword: Accepts any string and parses it as regex, so it can be a normal String or a regex expression.

  - Message Type: Accepts only the following Strings “Authorization grant message”, “Request messages”, “Response messages”, “All messages”, otherwise the check is not executed.
  - Section: The Strings allowed are “url”, “header”, “body”. In the case the check is executed on *Request Messages* the string “parameters” is allowed. For *Response Messages* the string “status code” is allowed too. In general, when the check is done on *All Messages* all of them are allowed and where the section is not available the search method just ignore it.

- Active(specific Operations): In this case the plugin checks for a message just like in the passive tests, but after the message is intercepted, it is taken and modified in order to execute an active attack. The check and the modification is all done during the exchanging of messages in the SSO process. In addition to the rules defined for the passive tests, for an active Operation additional JSON pairs need to be defined:

  - Action: For our implementation the following Strings are allowed: “Find And Replace Parameter”, “Find And Delete Parameter”, “Find And Modify Redirect Uri”, “Modify JWT alg type to none”. For replacing or deleting a parameter it is mandatory to define also the following values:
  - Update: Accepts any String. If it is the name of a existing parameter then an update with the value found in “Set” is executed, otherwise it is ignored.
  - Set: Accepts any String value.
  - Delete: Accepts any String. If it is the name of a existing parameter then it is deleted, otherwise ignores it.

In further implementations more Operations and Test cases can be defined.

If a *Test* consists of more than one *Operation*, it means that every *Operation* must be executed correctly. In other words, every passive test and every modification should be successful. If one Operation of the array is not successful, the Test is considered FAILED, if that does not happen it is considered SUCCESSFUL. For active tests, a test result is composed by two strings separated by (-): The first string indicates if the message was found and the action was successfully executed and the second one indicates if the last page is an error page or not.

5 Implementation of the plugin

In the re-elaboration of the existing plugin we modified the design, re-used some parts of it and implemented some new ones. The new plugin is divided in two main section: Authentication Trace part and JSON execution part. This bottom section consists of three tabs with completely different functionalities.

5.1 Structure of the Burp plugin

5.1.1 Authentication Trace

The Authentication Trace executes a trace inserted into the TextArea. This part exploits Selenium WebDriver. It is possible to run a test on the trace and verify if it works properly. This part was completely implemented by my colleague in [7] and just modified to make it work with the new version. The Figure 5.1 shows this part implemented inside the plugin.

Figure 5.1: Top Section of the plugin: Authentication Trace

5.1.2 JSON Parsing and Test Suite execution

The JSON for the generation of the *Test Suite* object is taken from an input text area. The plugin reads the input and if it is well defined creates a *Test Suite* object and consequently the *Tests* and *Operations* inside of it. In the execution of the *Test Suite* object, the array of *Tests* is sorted in order to execute the trace, previously inserted in the Authentication Trace input field. The selenium trace is executed just once for all the passive checks and for every active check the trace has to be executed one at the time. The plugin completes first the passive tests then all the active ones. Figure 5.2 shows the implementation of this part inside the plugin, with an example of Test Suite inserted in the input field.

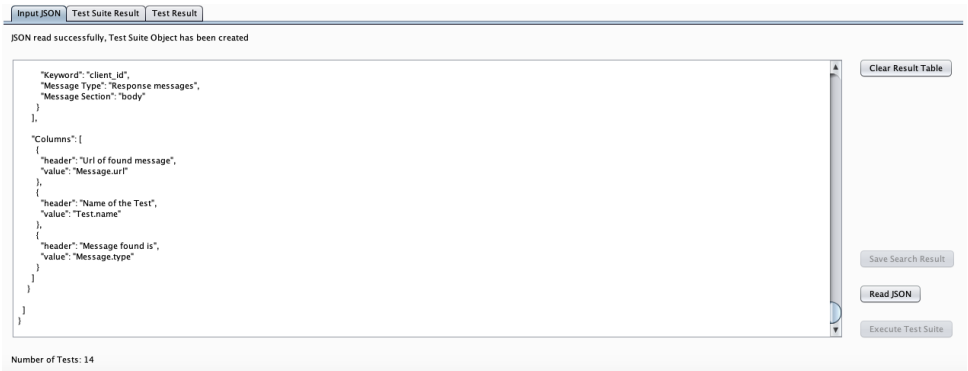


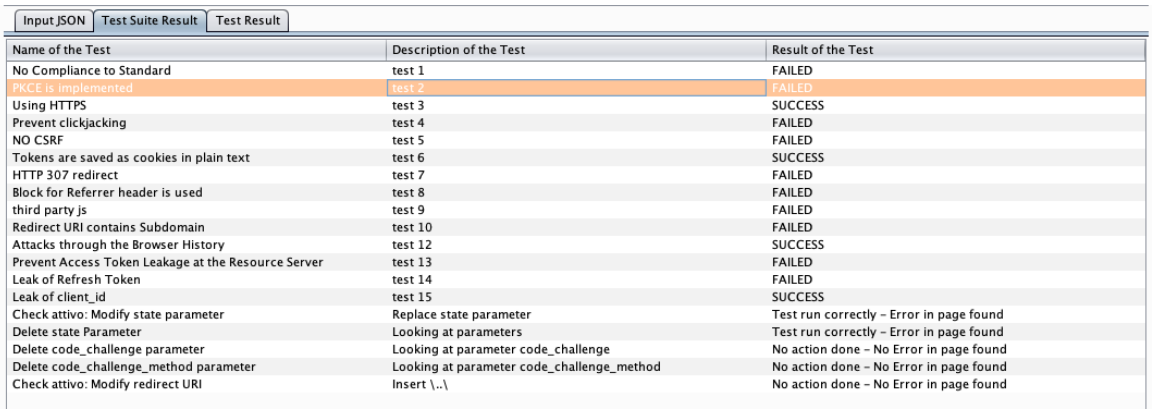
Figure 5.2: Input JSON tab: JSON input field and execution

5.1.3 Test Suite Result

The result of a *Test Suite* execution is showed in a second Tab. In this tab there is a table with a row for every Test done and for columns the following fixed values: Name of the Test, Description of the Test, Result of the Test. Figure 5.3 shows the results after executing a Test Suite correctly.

For passive checks the result can be: SUCCESS or FAIL. For Active checks the result consists of two parts separated by a dash (-):

- The first part says if a message with the required values was found and the modification could be done successfully or not.
- The second part warns the user if the final page shown during the execution of the selenium trace was an error page or not. If no action was done, as no message with the desired values was found, then it should not be an error page. In the case the active attack caused the modification of some messages the final page could be an error page.



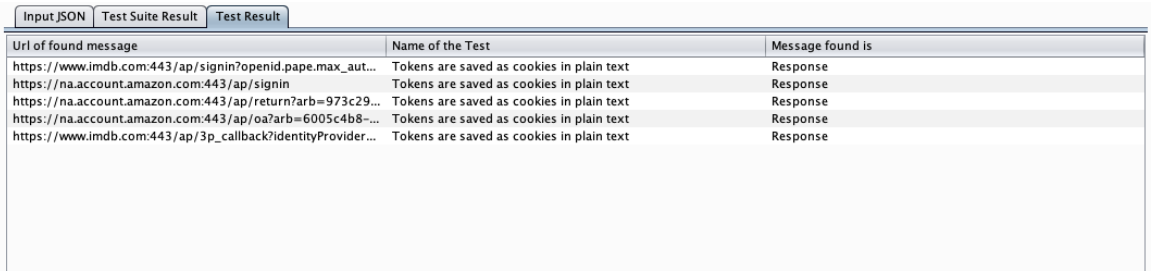
Input JSON | Test Suite Result | Test Result

Name of the Test	Description of the Test	Result of the Test
No Compliance to Standard	test 1	FAILED
PKCE is implemented	test 2	FAILED
Using HTTPS	test 3	SUCCESS
Prevent clickjacking	test 4	FAILED
NO CSRF	test 5	FAILED
Tokens are saved as cookies in plain text	test 6	SUCCESS
HTTP 307 redirect	test 7	FAILED
Block for Referrer header is used	test 8	FAILED
third party js	test 9	FAILED
Redirect URI contains Subdomain	test 10	FAILED
Attacks through the Browser History	test 12	SUCCESS
Prevent Access Token Leakage at the Resource Server	test 13	FAILED
Leak of Refresh Token	test 14	FAILED
Leak of client_id	test 15	SUCCESS
Check attivo: Modify state parameter	Replace state parameter	Test run correctly – Error in page found
Delete state Parameter	Looking at parameters	Test run correctly – Error in page found
Delete code_challenge parameter	Looking at parameter code_challenge	No action done – No Error in page found
Delete code_challenge_method parameter	Looking at parameter code_challenge_method	No action done – No Error in page found
Check attivo: Modify redirect URI	Insert \..\	No action done – No Error in page found

Figure 5.3: Test Suite Result tab: Result of a Test Suite correctly executed

5.1.4 Test Result

If a test is successful, so at least a message containing the pattern was found, a Result Table is generated. The user can click in every single test shown in the "Test Suite Result" tab and the table of the test would be shown in the last tab. If the user clicks on a failed test, the plugin will show an empty table. The table headers and values to be shown for every Test are defined completely by the user in the JSON input file. In order to access Test and Message data values the dot notation should be used inside the JSON file. An example of how a Test Result table looks like is shown in Figure 5.4. It shows the messages where "tokens are saved as cookies in plain text" were found



Input JSON | Test Suite Result | Test Result

Url of found message	Name of the Test	Message found is
https://www.imdb.com:443/ap/signin?openid.pape.max_aut...	Tokens are saved as cookies in plain text	Response
https://na.account.amazon.com:443/ap/signin	Tokens are saved as cookies in plain text	Response
https://na.account.amazon.com:443/ap/return?arb=973c29...	Tokens are saved as cookies in plain text	Response
https://na.account.amazon.com:443/ap/oa?arb=6005c4b8-...	Tokens are saved as cookies in plain text	Response
https://www.imdb.com:443/ap/3p_callback?identityProvider...	Tokens are saved as cookies in plain text	Response

Figure 5.4: Test Result tab: Result of a Test correctly executed

Listing 5.1 shows the standard that can be used to define the Test result table. It is possible to see that all the contents inside this table are decided entirely by the user.

Listing 5.1: Standard used to define headers and values of a Test Result table

```
...
"Columns": [
  {
    "header": "Header 1 of the Result Table of the single Test",
    "value": "Access to a Test or a Message value"
  },
  {
    "header": "Header 2 of the Result Table of the single Test",
    "value": "Test.name"
  },
  {
    "header": "Header 3 of the Result Table of the single Test",
    "value": "Message.url"
  }
  ...
]
...
```

5.2 Current limitations

Currently the plugin requires an understanding of the basic syntax of regular expressions, in order to define tests, therefore a brief introduction to regex is provided in this thesis. The plugin applies a search pattern to messages and eventually performs an action on the message. One big limitation of the plugin is that it cannot execute a sequence of Operations, so it is not possible to execute a test that requires to perform a sequence of actions. The plugin can define a test that looks for a message with a certain pattern inside the URL and at the same time contains another pattern inside the headers. To define this type of test, the test needs to have two different Operations, containing the necessary inputs. However it is not possible to execute a test that first looks to the URL, saves a string and after that looks at the headers for another pattern exploiting the extracted string. The language cannot define a temporal relation between different tests.

6 Use case OAuth/OIDC

6.1 Definition of tests with the language

In order to provide a practical implementation of the language developed, the tests found in Micro-Id-Gym OAuth/OIDC plugin have been defined using the new language. In this section we are going to show and explain how passive and active tests can be specified using the language. We are going to focus on the Operations required by the test, as the other input values are not crucial for the execution of the test but important for the presentation of the result of the test. As a consequence the input values for names, descriptions and headers' values of the Test Result table will not presented in this section, but they should be specified by the user in order to provide a valuable display of the tests' results.

Every test will require a number of Operations, which needs an "Action", a "Keyword", a "Message Type" and "Message Section" to execute the same amount of searches. As mentioned in Section 4.1 complex tests require to combine different regular expressions, the regex inserted in "Keyword" for every test would be provided in this section. Passive tests require to set "Action" to Passive, as after the regex pattern is used to search inside the traffic of messages exchanged, no action has to be performed. Active tests are executed in two steps: First it is necessary to identify the message to alter through one or more pattern searches; After that a modification of the messages identified will be executed. The alteration will require to add more values inside the Operation object.

In the following sections every test will be briefly introduced and a corresponding table with the values to insert in the Operation's fields will be presented.

6.1.1 Passive tests

In the language designed passive tests are defined as set of Operations, where the field "Action" is set to Passive. Every Operation executes a search pattern reading the "Keyword" value as regex. The search is executed in the group of messages defined by "Message Type" and the section of the message defined by "Message Section".

Verify compliance to Standard: The test verifies if the SSO process is compliant to the standard of OAuth/OIDC. Using the language, the test checks if the body of response message to the authorization request, where the authorization grant is given, does not contain the string `response_type=code` or `client_id`. If the test succeeds then the process does not comply to the standard. In Table 6.1 the values to insert inside the Operation are presented.

Table 6.1: Verify Compliance to Standard

Operation	**Input values**
"Action"	Passive
"Keyword"	`^((?!response_type=code).)*$\|^((?!client_id).)*$`
"Message Type"	Authorization grant message
"Message Section"	body

Proof Key for Code Exchange (PKCE) is implemented: The implementation of PKCE can protect the protocol from sniffing attacks [22]. PKCE uses three different parameters code_verfier, code_challenge and code_challenge_method. The last two parameters need to be included in the authorization grant message. The aim of the test is to verify that PKCE is implemented.

In the new language, the test checks if the body of response message to the authorization request, where the authorization grant is given, does not contain the string `code_challenge` or `code_challenge_method`. The definition of the Operation that executes this check is shown in Table 6.2. The test fails if it does not find any of these strings inside the body of the message specified.

Table 6.2: PKCE is implemented

Operation	**Input values**
"Action"	Passive
"Keyword"	`code_challenge|code_challenge_method`
"Message Type"	Authorization grant message
"Message Section"	body

Using HTTPS: The test verifies that all the messages exchanged are using the Hyper Text Transfer Protocol Secure (HTTPS). This protocol provides a security mechanism to encrypt the data, which the HTTP protocol cannot provide [2]. As shown in Table 6.3 the test using the language checks if the URLs of all the messages exchanged start with `https:`.

Table 6.3: Using Https

Operation	**Input values**
"Action"	Passive
"Keyword"	`^https:`
"Message Type"	All messages
"Message Section"	url

Prevent clickjacking: Including the header X-Frame-Options set to deny or origin can prevent from clickjacking, an interface-based technique used to trick the user to click into something different from what the user can see [16].

The test checks into the headers of the request messages exchanged, if X-Frame-Options parameter is set as `deny` or `sameorigin`. The test succeeds if it finds a request message that matches the regex and fails if the header is not set to `deny` or `sameorigin`. The specification of Operation is presented in Table 6.4

Table 6.4: Prevent clickjacking

Operation	**Input values**
"Action"	Passive
"Keyword"	`(?=.*X-Frame-Options)(?=.*(deny|sameorigin))`
"Message Type"	Request messages
"Message Section"	header

Prevent Cross-site Request Forgery (CSRF): CSRF is "an attack that forces an end-user to execute unwanted actions on a web application in which they are currently authenticated". The attack can be prevented by exploiting the `state` parameter [3]. In our language the test checks into the parameter list of the request messages looking for parameters `state`, `code` and `code_verifier`.

If the regex is able to find `state`, but not one among `code` and `code_verifier`, the process does not prevent CSRF. Table 6.5 will show the input values for the Operation of the test.

Table 6.5: CSRF is not prevented

Operation	**Input values**
"Action"	Passive
"Keyword"	`(?=.*code)(?=.*(^((?!code_verifier).)*$))(?=.*(^((?!state).)*$))`
"Message Type"	Request messages
"Message Section"	parameters

Tokens are saved as cookies in plain text: "Tokens should not be stored as plain text in cookies because they would be accessible to attackers in browser's cache" [7]. The test checks if any header of the response messages has a string `cookie` and `token` or `code` in plain text. The test is successful if the Operation finds a message with those values in plain text. The regex used to execute the search is presented in Table 6.6

Table 6.6: Tokens are saved as cookies in plain text

Operation	**Input values**
"Action"	Passive
"Keyword"	`(?=.*Cookie)(?=.*(token|code))`
"Message Type"	Response messages
"Message Section"	header

HTTP 307 redirect: In [3], the use of 307 for redirection is strongly discouraged. If the status code 307 is used for redirection, the user-agent will forward user credentials to the Client via HTTP POST, so through a data form. This will lead to a problem if the relying party is malicious, as the user credentials can be stolen and used to impersonate the user at AS. The test checks if any response message has 307 as the value for status code. As shown in Table 6.7, in our language the regex is a simple string applied to the status code field.

Table 6.7: HTTP 307 redirect

Operation	**Input values**
"Action"	Passive
"Keyword"	`307`
"Message Type"	Response messages
"Message Section"	status code

Block for Referrer header is used: It is possible to leak state or code parameters from Client or AS through Referrer headers [14]. It is possible to prevent that using an attribute `rel=noreferrer` which suppress the referrer header. As shown in Table 6.8, the test checks if any header of the response messages does not have the string `Referrer-Policy: no-referrer` and contains the string `rel=noreferrer` or `</a>`. The test succeeds if Block for Referrer is used.

Table 6.8: Block for Referrer header is used

Operation	Input values
"Action"	Passive
"Keyword"	`(?=.*Referrer-Policy: no-referrer)(?=.*(rel=noreferrer|</a>))`
"Message Type"	Response messages
"Message Section"	header

Leaks through the Browser History: The document OAuth 2.0 Security Best Current Practice explained that "Authorization codes and access tokens can end up in the browser's history of visited URLs" [14]. Hence if an attacker has access to the user's browser history, can use the value and replay it. The test described in Table 6.9 checks if strings `code=` or `access_token=` are saved in the URL in plain text.

Table 6.9: Leaks through the Browser History

Operation	Input values
"Action"	Passive
"Keyword"	`code=|access_token=`
"Message Type"	All messages
"Message Section"	url

External JavaScript or Components are used: "Loading external JavaScript or components from Content Delivery Networks can lead to use of malicious code" [7]. As presented in Table 6.10, the test contains two Operations and checks for script tags strings inside the response body. The test also verifies if there are scripts containing domains different from the domains involved in the login process. In Table 6.10 the domains involved are defined as `domain 1` and `domain 2`.

Table 6.10: External JavaScript or Components are used

Operation	Input values
Operation #1	
"Action"	Passive
"Keyword"	`Content-Type: text/html`
"Message Type"	Response messages
"Message Section"	body
Operation #2	
"Action"	Passive
"Keyword"	`<script(.*)src=\"[^/](?!.*domain 1|.*domain 2).*\"></script>`
"Message Type"	Response messages
"Message Section"	body

Redirect URI contains subdomain: "Redirect URI should be validated, more specifically for pointing to internal domains" [7]. As shown in Table 6.11 the test first checks if the status code is in the range of 300 to 399, as these status code indicate redirection. Then it looks into the headers of response messages for strings that indicate that redirect URI contains subdomain.

Table 6.11: Redirect URI contains subdomain

Operation	**Input values**
Operation #1	
"Action"	Passive
"Keyword"	`(3[0-8][0-9]|39[0-9])`
"Message Type"	Response messages
"Message Section"	status code
Operation #2	
"Action"	Passive
"Keyword"	`(?=.*code)(?=.*openid|oauth)`
"Message Type"	Response messages
"Message Section"	body
Operation #3	
"Action"	Passive
"Keyword"	`Location: https://(.*)\\.(.*)\\.(.*)\\.(.*)/(.*)/`
"Message Type"	Response messages
"Message Section"	header

6.1.2 Active tests

In our language, active tests contain Operations that execute the search of the pattern found in "Keyword" and perform an action defined in the field "Action". In the following sections the different Actions defined in our implementation would be presented.

Prevent state parameter modification: The test aims to alter the `state` parameter, as it is used to prevent CSRF attacks and maintain state between the request and response. The test is possible only if the `state` parameter is used.

The specification using our language is presented in Table 6.12. The test phases are:

- Looks for a request message with a parameter named `code` and with a parameter named `state`.
- Modifies the parameter `state` from the request message with a string defined by the user

Oracle: This test is considered successful if we manage to log in even if the `state` parameter is modified with a random string.

Table 6.12: Modify state Parameter

Operation	Input values
"Action"	Find And Replace Parameter
"Keyword"	`(?=.*state)(?=.*code)`
"Message Type"	Request messages
"Message Section"	parameters
"Update"	state
"Set"	random

Require state parameter: Similarly to the test that alters the parameter `state`, in this case the test attempts to delete the `state` parameter. The test is possible only if the `state` parameter is used.

The test is presented in detail in Table 6.13 and follows the following steps:

- Looks for a request message with a parameter named `response_type` and with a parameter named `state`.
- Removes the parameter `state` from the request message.

Oracle: This test is considered successful if we manage to log in even if the `state` parameter is deleted.

Table 6.13: Delete state Parameter

Operation	Input values
"Action"	Find And Delete Parameter
"Keyword"	`(?=.*state) (?=.*response_type)`
"Message Type"	Request messages
"Message Section"	parameters
"Delete"	state

Require code_challenge parameter: PKCE prevents from sniffing attacks, so this test tries to delete the `code_challenge` parameter. It is possible to execute the test only if PKCE is implemented, if it is not implemented the test does not alter any message. The Table 6.14 the parameters inserted in Operation for this test.

The phases of the test:

- Look for a request message with a parameter named `response_type` and with a parameter named `code_challenge`.
- Remove the parameter `code_challenge` from the request message.

Oracle: This test is considered successful if we manage to log in even if the `code_challenge` parameter is deleted.

Table 6.14: Delete code_challenge Parameter

Operation	Input values
"Action"	Find And Delete Parameter
"Keyword"	`(?=.*code_challenge)(?=.*response_type)`
"Message Type"	Request messages
"Message Section"	parameters
"Delete"	`code_challenge`

Require code_challenge_method parameter: As mentioned PKCE prevents from sniffing attacks, so this test attempts to delete the `code_challenge_method` parameter. It is possible to execute the test only if PKCE is implemented, if it is not implemented the test does not alter any message. Table 6.15 shows the input values inserted in Operation.

The phases of the test:

- Look for a request message with a parameter named `response_type` and with a parameter named `code_challenge_method`.
- Remove the parameter `code_challenge_method` from the request message.

Oracle: This test is considered successful if we manage to log in even if the `code_challenge_method` parameter is deleted.

Table 6.15: Delete code_challenge_method Parameter

Operation	Input values
"Action"	Find And Delete Parameter
"Keyword"	`(?=.*code_challenge_method)(?=.*response_type)`
"Message Type"	Request messages
"Message Section"	header
"Delete"	`code_challenge_method`

6.2 Assessment tests

After specifying the tests using the new language, we put all the tests together, passive and active, inside a Test Suite. The Test Suite is written using the JSON format and is a container of all the tests studied until now. For every single test, the plugin will show the result and eventually also the messages involved in every test.

This Test Suite was exploited for a vulnerability assessment of a group of web applications. The services analysed were four, all of them were different internal services of the same society. Following the principle of responsible disclosure, the services tested in this phase have not been revealed. They require an authentication using a Google account. Google APIs use OAuth for authorization and OIDC for authentication.

For each service, a trace to sign-in with Google was defined and a vulnerability assessment on the four different services was run inserting the trace and the Test Suite inside the plugin for Burp.

The Table 6.16 shows the results given by the plugin. Three services gave the same results to the different tests, therefore we are going to group them in (2).

For passive tests:

- ✓stands for the “process implements secure protocols or mechanisms”.
- ✗stands for “the process does not implement mechanisms to prevent the attack”.

For active tests:

- The first sign indicates if the messages searched was found:
  - ✓stands for “the message was found and action was performed”.
  - ✗stands for “no message was found”.
- The second sign indicates if the page reached was an error page
  - ✓the last page is not an error page
  - ✗the last page is an error page.

Table 6.16: Assessment of services

Tests inside the Test Suite	①	②
Passive tests		
Compliance to Standard	✓	✓
Prevent sniffing attacks using PKCE	✗	✗
Using HTTPS	✓	✓
Prevent clickjacking	✗	✗
Prevent CSRF	✗	✓
Tokens are not saved as cookies in plain text	✓	✓
HTTP 307 redirect is not used	✓	✓
Prevent leaks through Referrer header	✗	✗
Third party JS is not used	✓	✓
Redirect URI does not contain Subdomain	✓	✓
Prevent Leaks trough browser history	✗	✗
Active tests		
Modify state parameter	✗–✗	✓–✓
Delete state Parameter	✗–✗	✓–✓
Delete code_challenge parameter	✗–✗	✗–✗
Delete code_challenge_method parameter	✗–✗	✗–✗
Modify redirect URI	✗–✗	✗–✗

7 Conclusion and future work

As the use of digital identity is becoming every day more and more popular, especially to provide services that involve sensitive data, it is necessary to secure the protocols of identity management. In this thesis we focused on OAuth, in order to find the vulnerabilities through a set of tests. The tests allow us to identify vulnerabilities and secure the mechanisms.

The main focus of this thesis was the design of a language that provides the flexibility that tests need but also the ability to express complex tests in a simple but efficient way. This language uses JSON as the format to define tests and exploits regular expressions.

In the current implementation the language does not explore all the regex possibilities. Regular expression are a strong tool for search pattern and can be used to build more complex tests. Using a single regular expression it is possible to define a search that after finding the pattern replace it inside the text, regex can also be used to extract strings between two special characters. This is possible using capturing groups, so we can define a pattern, apply it to a text, define a capturing group and extract this information. In a test, the data extracted could be values or parameters to use and exploit in a following test. Regular expressions provide all this possibilities and they can be studied more in deep.

As future work, more complex tests could be written using the language or a Test Suite able to execute a sequence of tests could be implemented. The language could be also used to define tests for different identity management protocols, like the Security Assertion Markup Language (SAML), as the language provides a level of generalization and the parameters can be entirely defined by the user.

Bibliography

[1] Elmer Hernandez , Cofense Phishing Defense Center. MFA bypass phish caught: OAuth2 grants access to user data without a password. https://cofense.com/mfa-bypass-phish-caught-oauth2-grants-access-user-data-without-password/. last access on 25/02/2021.

[2] Cloudflare. Why is HTTP not secure? — HTTP vs. HTTPS. https://www.cloudflare.com/it-it/learning/ssl/why-is-http-not-secure/. last access on 25/02/2021.

[3] Auth0 Docs. Prevent attacks and redirect users with OAuth 2.0 state parameters. https://auth0.com/docs/protocols/state-parameters. last access on 25/02/2021.

[4] Mozilla Developer Network Web Docs. HTTP messages. https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages. last access on 25/02/2021.

[5] RFC Editor. https://www.rfc-editor.org. last access on 22/02/2021.

[6] Michael Yoshitaka Erlewine. http://web.mit.edu/hackl/www/lab/turkshop/slides/regex-cheatsheet.pdf. last access on 28/01/2021.

[7] Claudio Grisenti. A pentesting tool for OAuth and OIDC deployments. Bachelor thesis, DISI, University of Trento, 2019/2020.

[8] The Open Group. The open group base specifications issue 7. https://pubs.opengroup.org/onlinepubs/9699919799/. last access on 28/01/2021.

[9] D. Hardt. The OAuth 2.0 authorization framework. https://www.rfc-editor.org/info/rfc6749. last access on 19/01/2021.

[10] Ecma International. The JSON data interchange syntax. https://www.ecma-international.org/publications-and-standards/standards/ecma-404/. last access on 17/01/2021.

[11] Medium. Regex tutorial. https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285. last access on 28/01/2021.

[12] Elizabeth Montalbano. Clever phishing attack bypasses MFA to nab Microsoft Office 365 credentials. https://threatpost.com/phishing-campaign-allows-for-mfa-bypass-on-office-365/155864/. last access on 25/02/2021.

[13] University of Gueph. SSO benefits. https://www.uoguelph.ca/ccs/security/internet/single-sign-sso/benefits. last access on 28/01/2021.

[14] Internet-Drafts of the Internet Engineering Task Force (IETF). OAuth 2.0 security best current practice. https://tools.ietf.org/id/draft-ietf-oauth-security-topics-08.html. last access on 25/02/2021.

[15] Oracle. Package java.util.regex. https://docs.oracle.com/javase/7/docs/api/java/util/regex/package-summary.html. last access on 22/02/2021.

[16] Gustav Rydstedt , Open Web Application Security Project OWASP. Clickjacking. https://owasp.org/www-community/attacks/Clickjacking. last access on 25/02/2021.

[17] PortSwigger. Using Burp Proxy. https://portswigger.net/burp/documentation/desktop/images/proxy-using-intercept.jpg. last access on 27/01/2021.

[18] PortSwigger. What is Burp Suite? https://portswigger.net/burp. last access on 27/01/2021.

[19] Andrea Bisegna, Roberto Carbone, Ivan Martini, Valentina Odorizzi, Giulio Pellizzari, Silvio Ranise. Micro-Id-Gym: Identity management workouts with container-based microservices. https://stfbk.github.io/tools/Micro-Id-Gym. last access on 25/02/2021.

[20] R. Fielding , J. Reschke. Hypertext transfer protocol (HTTP/1.1): Message syntax and routing. http://www.rfc-editor.org/info/rfc7230. last access on 17/01/2021.

[21] Margaret Rouse. TechTarget's IT encyclopedia, Single Sign-On (SSO) and federated identity. https://searchsecurity.techtarget.com/definition/single-sign-on. last access on 19/01/2021.

[22] N. Agarwal , J. Bradley , N. Sakimura. Proof key for code exchange by OAuth public clients. https://www.rfc-editor.org/info/rfc7636. last access on 19/01/2021.

[23] R. Shirey. Internet security glossary, version 2. https://www.rfc-editor.org/info/rfc4949. last access on 19/01/2021.

[24] Selenium WebDriver. https://www.selenium.dev. last access on 28/01/2021.