University of Trento - Italy

Department of Information Engineering
and Computer Science

Bachelor's Degree in Computer Science

Final Dissertation

A pentesting tool for OAuth and OIDC deployments

Supervisor

Silvio Ranise

Student

Claudio Grisenti

Co-supervisors

Andrea Bisegna

Roberto Carbone

Academic Year 2019/2020

Acknowledgements

I would like to thank my parents,
for always supporting me and allowing me to cultivate my passion for IT.

Contents

1 Introduction **5**
1.1 Problem statement . . . 5
1.2 Contributions . . . 6
1.3 Structure of the thesis . . . 6

2 Background **7**
2.1 Terminology . . . 7
2.2 SSO . . . 8
2.3 OAuth 2.0 . . . 8
2.3.1 OAuth grant types . . . 11
2.3.2 PKCE Extension . . . 12
2.4 OpenID Connect . . . 12
2.4.1 JSON Web Token . . . 13

3 Preliminary Analysis **14**
3.1 Sources . . . 14
3.2 Vulnerabilities . . . 15
3.3 Software and Plugins . . . 16

4 Micro-Id-Gym: pentesting tool for OAuth/OIDC **19**
4.1 User Interface . . . 20
4.2 Plugin Architecture . . . 21
4.3 Plugin flow . . . 22
4.3.1 Authentication Trace . . . 22
4.3.2 Passive and active checks . . . 23

5 Experimental analysis **24**

6 Conclusions and future work **26**

Bibliography **27**

Abstract

In the modern landscape of Information and Communication Technologies, Cybersecurity is more important than ever. In recent years we have seen an exponential growth in services available to every computer, allowing every person to please every kind of need, from a basic one like e-commerce to keeping in touch with your friends on social medias. However, this led to an incumbent advancement in security measures, given the manipulation of users' sensitive data over the Internet, a notorious place for its wide-spread presence of insecure applications and users.

Access to sensitive data must be granted after a thorough inspection of the provided credentials, reason why it is crucial to create and implement the right technologies to authenticate and authorize every user online, two faces of the same coin. This lead to many services or applications delegating the management of authentication to a third party, which in some cases makes available a type of service called Single Sign-On (SSO). This particular authentication scheme gives the possibility to the user to access multiple web applications with one set of credentials. OAuth 2.0 (hereafter OAuth) is an open standard for access delegation, used mainly for granting clients the authorization to server resources on behalf of a resource owner and widely available on web browsers due to its use of the Hypertext Transfer Protocol (HTTP). OpenID Connect (hereafter OIDC) provides an identity layer on top of OAuth, which guarantees authentication and specifies a RESTful HTTP API with JSON as data format.

OAuth implementations, despite their heavy use on well-known services and applications, are prone to vulnerabilities and if exploited by malicious users can lead to critical damage to entire systems. Unfortunately these flaws can be found in multiple entities involved in the protocol and result in different exploits by an attacker. An example can be the lack of use of an extension called Proof Key for Code Exchange (PKCE) during the initial phase of the protocol which handles the exchange of codes used to obtain the needed resources. The extension guarantees that the user that receives the code is the same that consumes it to obtain the token which allows access to the private resources. The absence of this extension can lead to replay attacks and malicious entities acquiring confidential data.

For the explained reasons, high skills in the security field and reliability are needed to implement and set up a proper application based on OAuth or OIDC that can ensure the security of private data of every user. Unfortunately it is not easy to have this set of skills, both for its cost and difficulty and if the developers choose to not integrate these protocols in their work, they can deliver insecure services. This rises the need for a guide to help developers to follow the current best practices, highlighting what extensions, conventions or parameters to include to ensure protection, and a tool capable of checking whether the implementation of a service is compliant with the introduced protocols or fails to follow all suggested security practices and notify the developers what has to be improved. Some tools are already available online but they do not deliver a thorough inspection, nor work with any kind of implementation.

Our contribution can be divided in 3 phases, the first one was focused on writing said guideline for developers, allowing them to have a compact and focused list of practices accessible during the entire production of a service or application. To gather this information we used research papers, security considerations and guidelines based on real-world scenarios. This allowed us to have details on the stated problem both from a theoretical point of view, defining what the essential extensions or parameters are and why these have been created to ensure security, and on a practical point of view the repercussions of improperly configured systems. The next step was a research of the already available tools online, focusing on their implementations and capabilities. We concentrated whether the tools were automated (they required no input from the user during execution) or not, the programming languages and technology used and if they were specific to a given implementation of the protocols, permitting us to define exactly what our tool needed or not. Both the guideline and the list of tools allowed us to move the next phase, the development of Micro-Id-Gym OAuth/OIDC, a tool capable of providing a solution to the stated problem.

1 Introduction

Recently we have witnessed an increase in the use of SSO protocols, to ease the use of multiple platforms that require authentication. One example can be the many government services available online that can be accessed at anytime and require just one set of credentials, or even the many service platforms delivered by private organizations, like Facebook or Google and its plethora of services, accessible with just one account.

The use of SSO protocols is almost necessary, to avoid password fatigue, i.e. the effect required to remember several credentials for the increasing number of services used. This huge growth in online services brought many inexperienced users which may find remembering a large number of credentials difficult, or even understand what credential to use given an application. On the other hand, this can be useful even to most expert users: why have one account per application used when you can simply sign in once and just continue with your work.

The protocols that will be explained in details in section 2 are the simplest solution to this situation and they have become successful in recent times because of their key features:

- *Identity Management delegation*: they do not require a centralized figure that acts as authentication/authorization provider but anyone who wishes to implement their version can act as provider;
- *Extensions*: this protocols have been developed with modularity in mind, with the possibility of adding new features as extensions;
- *Platform agnostic*: they do not require a specific platform to work, instead they have been developed using API which can be used by both Web applications and Mobile ones.

These basic features were chosen because of their versatility: they made it possible for private companies to use these protocols without relying on third-parties and implement them on any platform they preferred, web or mobile. Most importantly, if a developer for any reason needed a feature which was not available, they could simply add it.

1.1 Problem statement

Many authentication and authorization protocols exist and in this thesis OAuth and OIDC will be specifically taken in consideration. The deployment of these protocols can be flawed, due to the fact that many of the parameters used are not defined as mandatory and can be omitted if the programmer wishes to (or neglects to). This brings to the table many problems such as: incorrect generation, incorrect checks or validations for codes or tokens in the authorization/authentication flow. Furthermore, the issues stated become more relevant when we go deeper into the details of these protocols: they use codes and tokens which are described as bearer credentials [18]. This type of credential lays the foundations for many different protocols in many fields of Cyber Security because of the basic idea behind it: the bearer of the token or code, which is everyone in possession of it, is acknowledged as an authenticated and/or authorized user, which means he or she has complete access to any confidential information linked to the account. The approach just described has become successful for both its ease of use and how simple is to implement. The user just provides the required credential and can use the needed web services, but the provider does not check for the real identity of the user.

1.2 Contributions

The contributions of this thesis are the following:

1. Review of the state of the art regarding the vulnerabilities for OAuth/OIDC, focusing both on theoretical aspects and real-world scenarios, allowing us to have an overview of the best security practices;
2. Analysis of the existing tools created to carry out a security assessment for OAuth/OIDC implementations;
3. Designing and developing a tool that covers the missing practices not available in the tools found in step 2, allowing us to deliver one capable of testing any kind of implementation of the protocols in a thorough manner.

Our work can be used by users, who want to check whether the services they are using are safe, but also by developers who want to guarantee protection of the users' data. Moreover, this tool can be used in the early stages of the system development to have a list of security-driven practices, but also in the last stages to check whether the service soon to be available is secure. In particular our contribution is focused on:

- **Pen-testing:** testing already deployed services is a key aspect of delivering secure ones. Our contribution aims at solving this problem, giving the possibility to developers to check their application in a fast and precise manner.
- **Secure Deployment:** making sure that software in use is safe is very important, but also testing the one in development is even more so. Testing services before deployment is crucial for companies because it would avoid drastic consequences in case of a successful attack.

1.3 Structure of the thesis

The thesis is structured in this manner:

Section 2 presents the basic terminology for this thesis, protocols and frameworks used.

Section 3 presents the results of the state of the art search of already available tools on the web.

Section 4 presents the tool's architecture and the checks implemented.

Section 5 presents the results of the analysis made on well-known web applications.

Section 6 contains some conclusions and describe future works.

2 Background

2.1 Terminology

Following the specifications in RFC 4949 [23], we define the following concepts:

Authorization: An approval that is granted to a system entity to access a system resource.

Authentication: The process of verifying a claim that a system entity or system resource has a certain attribute value.

Authentication and authorization lay the foundations of every cyber security protocol, given the fact that one is intertwined with the other. That is, one user can be authorized only if authenticate first, and one user being authenticate leads to having an authorized used.

Credential: A data object that is a portable representation of the association between an identifier and a unit of authentication information, and that can be presented for use in verifying an identity claimed by an entity that attempts to access a system. Example: username and password;

Confidentiality: property of preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information;

Integrity: property of guarding against improper information modification or destruction, and includes ensuring information non-repudiation and authenticity;

Availability: property of ensuring timely and reliable access to and use of information;

Threat: A potential for violation of security, which exists when there is an entity, circumstance, capability, action, or event that could cause harm;

Vulnerability: weakness in an information system, system security procedures, internal controls, or implementation that could be exploited by a threat source;

Attack any kind of malicious activity that attempts to collect, disrupt, deny, degrade, or destroy information system resources or the information itself. A successful attack can be also called exploit;

Attacker a party who attacks a host, network, or other IT resource;

One aspect that has to be clarified is that the word exploit and attack refer to two different situations:

- An exploit indicates a successful security violation.
- An attack indicates an attempt at a security violation, but not its result.

2.2 SSO

Single Sign-On (SSO) is a property of access control applied to two or more independent software applications or systems allowing users to log in with one unique set of credential and have access to all services available on said platforms. This approach allows for users an easier and more relaxed experience given the fact that the issue of inserting credentials many times and their management is not necessary anymore.

The idea just briefly explained will surely make you soon realize that many advantages arises with this, even for the inexperienced user and when considering security:

- risks mitigation for 3rd-party sites;
- reduction of password fatigue from different IDs and passwords;
- cost decrease due to IDs or passwords reset requests;

Procedures and basic structures are needed for this method of authentication, one is obviously some entity entitled to the administration of the every credential of each consumer. The key role for this identity provider in this access control implementation is that every time an authentication is issued, the claimer is redirected to the provider which will check whether the assertion is genuine or not. Another needed feature is session management: this is a procedure which allows a steady usage and interaction between the user and the service, which is vital for every transaction (each having a specific identifier).

Different solutions are available to the public, one is OIDC, an OAuth extension which allow the underlying protocol to handle authentication besides authorization, and SAML SSO, an open standard based on XML markup language.

2.3 OAuth 2.0

OAuth [16] is a delegation framework which allows a third-party application to obtain a limited access to a private resource using the HTTP protocol, either after the authorization of the resource owner or allowing the application to acquire it on its own behalf.

For a clear explanation of this authorization flow we can use as example Google Docs, a service provided by Google which lets two or more users to share and modify online various types of documents. A typical scenario in which OAuth is used is when a web service, or a mobile application, needs access to a document or media file on the Google Docs service shared between multiple accounts. When the application requires this resource, the user is redirected directly to the Google Account server to verify the owner identity. If the authorization is successful, Google Account provides the resource owner an authorization code, which will be given to the application and exchanged for an access token that will allow access to the resource. This authentication flow is shown in Figure 2.3.1.

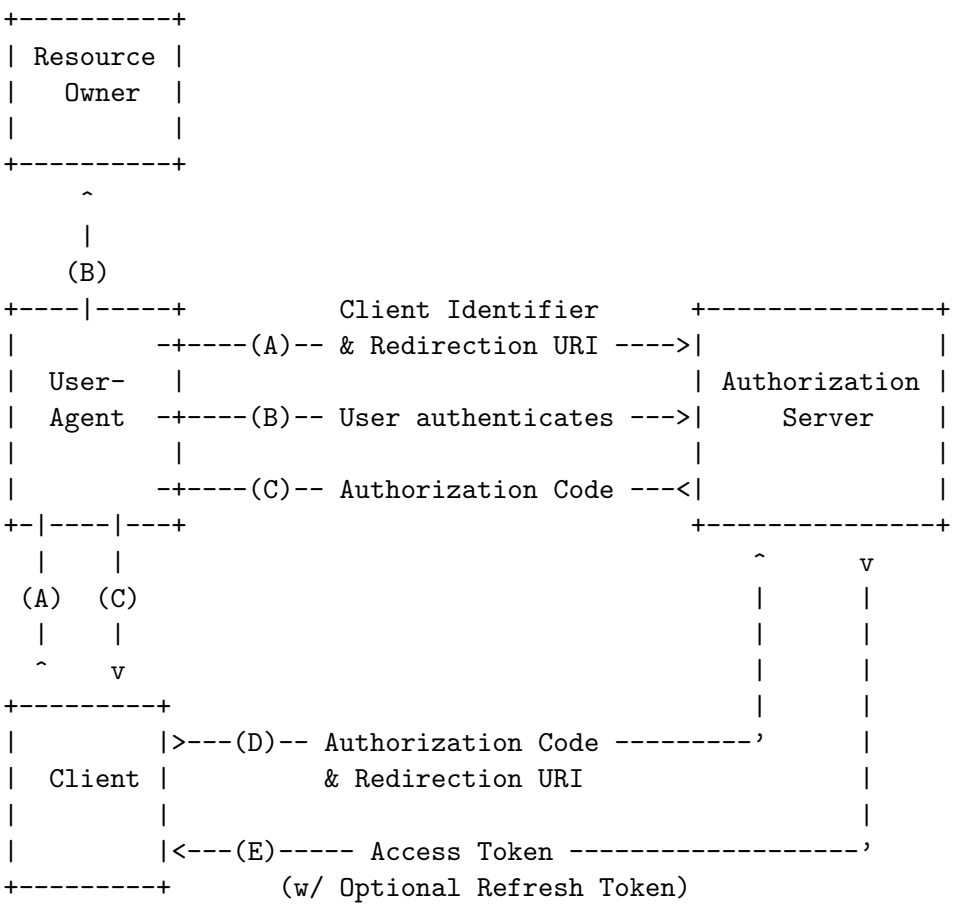

Figure 2.3.1: Authorization code Flow [17].

To have a thorough explanation of the standard, we will define the terminology as in [17] to define specific roles in the OAuth standard:

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 (e.g., whether the application executes on a server, a desktop computer, or other devices). There are 2 type, differentiated by the ability to authenticate securely with AS:

1. *public client:* Clients incapable of maintaining the confidentiality of their credentials;
2. *confidential client:* Clients capable of maintaining the confidentiality of their credentials;

Authorization Server (AS): The server issuing authorization codes/access tokens to clients after successfully authenticating the resource owner and obtaining authorization;

With regards to protocol specification [17], we define the following parameters:

Access Token : Access tokens are credentials used to access protected resources. An access token is a string representing an authorization issued to the client. The string is usually opaque to the client. Tokens represent specific scopes and durations of access, granted by the resource owner, and enforced by the resource server and authorization server;

Refresh Token : Refresh tokens are credentials used to obtain access tokens. Refresh tokens 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 (access tokens may have a shorter lifetime and fewer permissions than authorized by the resource owner);

Authorization Code : a bearer credential issued to the client that can be exchanged for an access token;

Client_id : identification code that identifies a client assigned during registration;

Redirect URI : URI used to redirect the user to the client when authentication phase succeeds. This value must be set at client registration time;

State : a random generated nonce used by the client to keep a state between a request and a response;

A *Grant Type* is the process followed by C to obtain a valid access token and can be defined in multiple ways, basing it both on the possibility of one entity acting as multiple ones (e.g. C and RO can act as one) and the degree of trust between them. At the time of writing 4 major schemes are famous among the different OAuth implementations, but the protocol also gives the possibility of creating custom ones using extensions improving its flexibility even further.

The *Authorization Code* grant type stands out because of its level of security and option to grant access to a client with no need to disclose RO's credential and it is also the scheme which we will take in consideration in the following of the thesis. The flow for this grant type is depicted in figure 2.3.1 and starts with RO and C exchanging messages through the use of an User-Agent. As first step we have C requesting access to a private resource which can only be granted explicitly by RO . Explicitly means that RO's agreement to the access of private data is valid only after some type of input, e.g. clicking a button. After this, RO is redirected to AS by C to obtain an authorization code after a valid authorization process. After C acquires said code by RO, the last phase of authentication takes places between C and AS, with the exchange of the authorization code for the access token. With this, C is able to access the required data directly hosted by RS by simply providing said token.

The remaining grant types are described in Section 2.3.1.

2.3.1 OAuth grant types

An authorization grant is the way which the Client gains access to the resource (in other word how C obtains the access token), more particularly on which credentials uses.

OAuth defines 4 types of grant which have different levels of security and trust for each role and parameters used. Obviously anyone can create new grant types with new extensions.

The four major grants are:

Authorization code: described in Section 2.3, it is mostly used because of its security and possibility of separating every entity described by the protocol. Furthermore, C requires no private information (aside from the resource hosted by RS) from RO that can lead to exploits;

Implicit: An access token is directly issued instead of an authorization code, improving performance, but with new security concerns;

Resource Owner password Credentials: RO's credentials are used directly to obtain an access token. This grant should be used only when there is a high degree of true between RO and C;

Client Credentials: this grant type is used when the client is acting on its own behalf (C and RO are the same entity) or C is requesting access to a previously granted resource;

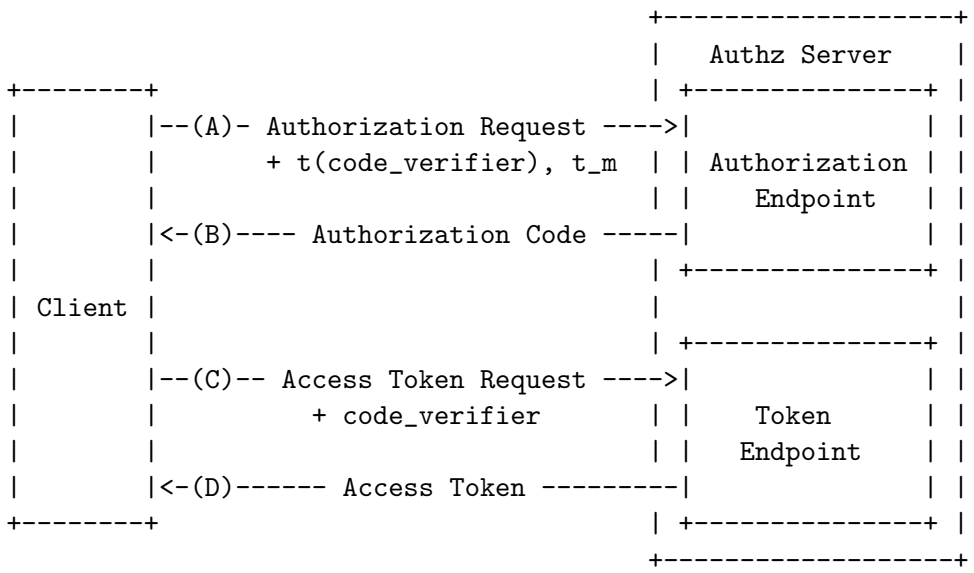

Figure 2.3.2: PKCE abstract protocol flow [20].

2.3.2 PKCE Extension

If OAuth is used with no encryption (usually TLS/SSL is an undergoing encryption layer), the authorization phase between RO and AS can be intercepted, which will lead to a sniffed authentication code and a replay attack with an attacker capable of obtaining a valid access token and RO's private data.

This vulnerability can be mitigated with an OAuth extension called **PKCE** (Proof Key for Code Exchange), defined in RFC 7636 [20], which uses 3 parameters *code_verifier*, *code_challenge* and *code_challenge_method*.

PKCE uses an unique randomly generated value (code_verifier) which is encrypted into a value called code_challenge using a security proofed hash function indicated by the value of code_challenge_method.

This extension does not change the usual authorization code flow, but instead adds these values to specific requests in the grant type to avoid replay attack. More precisely, code_challenge and code_challenge_method values are included when C requests for an authorization code to the AS and code_verifier is included when the authorization code is exchanged for an access token.

With all these values, the server can prove that the entity requesting the authorization code is the same exact one that used it to request an access token, given the security provided by the hash function.

An example of the flow is depicted in figure 2.3.2.

2.4 OpenID Connect

As described before, there is distinct difference between authorization and authentication and, for now, we described a protocol which only authorize an entity the use of a resource. AS in the OAuth protocol has sometimes been used also as authentication server to avoid including other protocols or extensions. This methodology can lead to vulnerabilities called *code substitution attack*: an attacker can create a malicious application acting as a valid one and request the victim to login and obtain a valid authorization code to exchange it for an access token.

OpenID Connect [7] has been developed to solve this problem. It is a OAuth extension which enriches the capability of the protocol giving the opportunity to add an extra layer on top of it which handles authentication. This is managed by introducing a new entity called **Identity Provider** (ID) that handles both the registration and authentication process and **Relying Parties** (RP) which are the applications in need of authentication and "rely" on IDs capabilities to identify or not genuine

users.

This extension is much similar to OAuth both for its features and for its logging process. It does not use a central authority (which would be bottle-neck for the entire process) and the use of a single set of credential can guarantee access to multiple services.

2.4.1 JSON Web Token

The success of OAuth and OpenID Connect is also due to the fact that they can be easily applied to many type of services, may them be web application or even mobile applications and it is possible because the used core technologies are adaptable and modular.

An example of this idea are **JSON Web Tokens** [19] (JWT), an identity token which encodes multiple user information inside a quite compact string. This is made possible by packaging all the needed details inside three separated *JSONs*: the first one (header) indicates what type of encryption function is used (hash function), the second (payload) asserts the user information and the last of is the digest (*signature*) of the hash of both the header and payload. A nonce is used in the hash function to ensure more security.

These three strings are encoded into a base 64 URL-safe string, for ease of use and read, and separated by a dot. An example of a JWT can found in Table 2.4.1. The most common parameters in the payload can be found in Table 2.4.2.

The use of a single string can deliver an incredible possibility to the application: the authenticated user can simply send his/hers JWT to asserts his/hers identity and the application can verify by simple calculating the verify signature. If they match the user is genuine, and if they do not match, the user who sent the request is not who is declaring.

Table 2.4.1: JWT example.

Plain text content
Header: "alg": "HS256", "typ": "JWT"
Payload: "sub": "1234567890", "name": "John Doe", "iat": 1516239022
Verify Signature:
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret)

JWT
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9. eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ. SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

Table 2.4.2: Information encapsulated inside a JWT.

Parameter	**Value**
sub	asserts identity of the user (subject)
iss	indicates the authority who issued the jwt (issuer)
aud	created for a specific client (audience)
nonce	may contain a nonce
auth_time	specifies when the user authenticated
exp_time	indicates when the jwt expires
iat	indicated who was issued at
name / email adress	additional information about the user

3 Preliminary Analysis

At the beginning of the project, two main rules were set that had to be followed during the entire process to deliver a better and well thought-out tool:

1. **The Tool must be implementation agnostic:** this directive aims at having a tool capable of working on well-known OAuth/OIDC implementation, as well as new ones and help every developer to know if his/her work lacks on some security concerns.

2. **The Tool must cover every best current practice:** OAuth and OIDC are standards, which means there are guidelines that *can* be followed. An example can be the security extension described in Section 2.3.2 or the State parameter, features developed to ensure security but not made mandatory in the protocol. This tool must check whether a given implementation does not follow any of them.

We payed attention on the first one during the entire development, making sure every check did not rely on any implementation-dependent information or setting. This was made possible by basing every part of the tool, both its flow and check architecture, solely on parameters required or always available in every implementation of the standard.

The second rule required an extra step before the actual development of the plugin: a search of the already available tools that perform automated checks on OAuth/OIDC web services. This step has been made in order to have an idea on the current situation of the acquirable tools by the public and to define perfectly what had to be implemented or not in the plugin. Specifically to the latter, this information was essential to avoid the implementation of checks already present in the found tools.

To gather the information needed, many platforms/methods have been used, given the fact that many applications/tools were spread on many of them:

Web search engines: search engines have been select because some findings were on specific websites hosted to describe the applications developed, or articles written on it.

Git public repositories: these platforms allow developers to both interact with each other to improve their work and also release their product in the easiest way possible.

Google Scholars in this specific version of the well-known search engine, searches about scholarly literature is made more simple and it helped us find articles about tools and their implementations.

3.1 Sources

To ensure completeness of the tool we needed a list of the best practices regarding security for OAuth and OIDC. We used different types of resources during our analysis, from research papers to security considerations and guidelines based on real-world scenarios.

We mainly used 3 documents during the first phase, **Assisting developers in securing OAuth 2.0 deployment** [21] by Davide Piva, **OAuth 2.0 Security Best Current Practice draft-ietf-oauth-security-topics-13** [26] and **OWASP Helsinki Chapter Meeting #30 October 11, 2016** [27].

The first document, Assisting developers in securing OAuth 2.0 deployment, is a thesis written by Davide Piva in his last year of his Bachelor's Degree in Computer Science at the Department of Information Engineering and Computer Science of the University of Trento. This dissertation focused on a formal description of the OAuth threats defined in the best practice documents and for each a clear identification of the corresponding countermeasure. This work focused on helping every developer to build a secure OAuth application, even the ones without advanced cyber security knowledge.

The second document, OAuth 2.0 Security Best Current Practice draft-ietf-oauth-security-topics-13, describes the current security practices for OAuth, updating and extending OAuth 2.0 Threat Model to incorporate practical experiences since it was published. Internet-Drafts are draft documents

valid for a maximum of six months and can be updated or replaced by new ones at any time. During our work we specifically worked on number 13, published on July 8th 2019. It contains a set of 13 practices which have to be followed to ensure security.

The third and last document, OWASP Helsinki Chapter Meeting #30 October 11, 2016, is a summary of a meeting held by OWASP [8], a community-driven website about web application security which provides articles, tools and corresponding documentation. The main topic discussed during this meeting is the security concerns regarding well-known and used SSO protocols, such as OAuth, OIDC and SAML SSO. This piece of writing describes the established vulnerabilities in the said protocols using both theoretical knowledge and real life cases of wrong implementations in recognized sites or applications.

3.2 Vulnerabilities

A comprehensive list of all the security best current practices can be found in Appendix A. This attachment contains a set of test cases, a formal description of every test that the tool has to include to deliver a thorough inspection of the implementation taken in consideration. Each test case contains the name of the check, a brief explanation of what practice is referring to and a description of each phase of the test, particularly the set up, how to perform it and what it should return.

Table 3.2.1 contains a summary of Appendix A, including a name of each test case and a brief description of what the practice taken in consideration is about. Some of the entries in this table show that many vulnerabilities can be avoided with simple good programming conventions, may them be limit codes and tokens lifetime or avoiding storing any information client-side.

Table 3.2.1: Test cases summary.

Test Case	Description
URI redirection validation	URI should be validated, more specifically for pointing to internal domains or bypass- ing checks with /../ .
Tokens stored as plain text in cookies	Tokens should not be stored as plain text in cookies because they would be accessible to at- tackers in browser's cache.
Tokens expiration and lifetime	Tokens' expiration time should be specified and its duration should be less than 10 minutes.
Authorization code lifetime	Authorization code lifetime should be short, usually less than 10 minutes.
JWT validation	JWT are used in OpenID Connect for user information and authentication session. Keys that must be validated: iss,aud, id token signature,exp,iat,auth time. JWT algorithm type need to be checked for "none" value
Correct use of refresh token	A refresh token, coupled with a short access token lifetime, can be used to grant longer access to resources without involving end-user authorization but it should be used only when needed.
Use of HTTP 307 status code	The use of HTTP 307 status code can lead to user's credential leak from the login page.
Incorporating external JavaScript/Components	Loading external JavaScript or components from CDNs can lead to use of malicious code.
Leaking authorization code through referrer head	Referrer header can be used to obtain code, access token or state value. This can be disclosed unintentionally by both Clients and A.S.'s web site
Credential/Sensitive data/ Secret data disclosure	If the network used for the protocol is not secure, data can be disclosed (TLS/SSL needed).

3.3 Software and Plugins

The research resulted in many wide variety of tools, both on a architectural and complexity point of view. One key aspect of the problem that we are facing is that OAuth and OIDC can be utilized on web services, may them be a simple website that prefers credentials management to be simple and relies on these two protocols, but also mobile applications installed on miscellaneous devices that can prefer the same approach to the same problem.

As the platforms differ from each other considerably, the implementations of the protocols do too. Two points which can describe easily the contrast in implementation for given platform are the Client role and the grant type. Usually Clients regarding the mobile platform are called **native apps** and following RFC 8252 [28] terminology it can be described as an application installed by the user to their device which is distinct from a web app that runs in the browser context only.

Usually, the best and most used grant type is *authorization code type* because it can deliver security and privacy for the user credentials and does not rely on trusting any third-party application, but native apps can be trusted due to their nature. In the recent year we have seen a extensive growth in **app stores**, centralized digital shops that distribute application directly to the user. Their best feature is that many of them perform checks on the available item that they hand out, ensuring no malicious-code, allowing the use of implicit grant type or Client Credential grant type. Unfortunately sometimes this is not the case, because app stores are not the only mean for obtaining applications as they can be downloaded from different web sites and not be genuine. This issue is the reason why the *authorization code* grant type and the use of external user-agent are still recommended in general.

This brief description can make you understand why every of the developers decided to focus either on web services or mobile applications for their tools and that is why the research gave a clear distinction between them. For example web services tools opted for browsers extension that can directly handle HTTP packets and mobile apps opted for emulators working on smartphones. Our work focused on web applications.

Tables 3.3.1 and 3.3.2 will describe the findings, splitting them between the targeted role in the checks. A brief description of the content of each column can be:

Covered Attacks/Vulnerabilities: a list of the vulnerabilities and attacks checked by the tool;

A/S/M: this column describes the degree of automation during the use of the tool, if it requires no user input (Automatic), if it requires some user input (Semi-Automatic) or if it is completely manual. For each Semi-Automatic tool a note is available which describes what input is required by the user;

Availability: specifies whether information regarding the source code or the implementation is available to the public or not. This data can be useful to new developers who want to expand the already available tools or create new ones;

License: information regarding the license applied to the tool;

Technology: programming language used to implement the tool;

Platform: describes which type of platform is tested, web or mobile. Given the differences between the two, every tool can focus only on one;

All the gathered information can be useful during the implementation for multiple reasons: to have an extensive insight of the state of the available pentesting tools but also for the possibility of including some of the found work in ours. The idea is that if both our tool and one of the discovered use the same technologies, they can be merged resulting in a more complete tool (if the license applied on the work allows it).

Unfortunately, not all the findings resulted in an open source product, specifically AuthScan's and AuthScope's developers have been contacted to acquire information regarding the possibility of an examination of the source code of their work but no answer has been received. Moreover, other entries in the table show tools without any license, that is because these were found on git repositories managers like GitLab and GitHub without any applied license.

Table 3.3.1: Collection of Tools.

Tool Name	**Covered Attacks/Vuln [27][25]**	**A/S/M**	**Availability**	**License**	**Technology**	**Platform**
Target: Any						
oauth2-redirector [5]	CSRF (state parameter)	S[1]	Yes	MIT	Ruby	Web
oauth2-client-tester [4]	Checks for correct 3-way handshake	A	Yes	-	Ruby	Web
Target: IdP Role						
MoSSOT [22]	Credential disclosure Previously unknown vuln. IdP	A	Yes	Apache2	Python	Mobile
LinkedIn Oauth Tester [2]	Checks for correct token creation on Linkedin	A	Yes	-	Ruby	Web

Legenda: "A": Automatic, "S": Semi-Automatic, "M": Manual, "-" : No Licence applied, "N/A" : Not available

[1]Login and authorization required.

Table 3.3.2: Collection of Tools.

Tool Name	Covered Attacks/Vuln [27][25]	A/S/M	Availability	License	Technology	Platform
Target: Client Role						
MoSSOT [22]	Profile Vulnerability App secret disclosure Access Token Replacement Augmented Access Token Replacement	A	Yes	Apache2	Python	Mobile
OAuth - Vuln. scanner [3]	Too big token lifetime Token transmitted using TLS Weak JWT sign method	S[2]	Yes	-	Python	Web
AuthScan [15]	Replay attack in BrowserId CSRF attack in BrowserID Secret Token Leak in Facebook Connect Non-secret Token in Using Windows Live ID Guessable Token in Standalone Sites	A	No	-	N/A	Web
Overscan [24]	X-CSRF Token attack CSRF Auth. Code phishing Clickjacking	A	Yes	-	Java	Web
AuthScope [29]	Sensitive and secret data leakage	A	No	-	N/A	Mobile
OAuthGuard [6]	Unsafe Token Transfer CSRF Impersonation Privacy Leaks OAuth flow misuse	A	Yes	GNU/GPL3	JavaScript	Web
PrOfESSOS [10]	Issuer confusion ID spoofing Reply Attacks Signiture Bypass IdP Confusion Malicious Endpoint	A	Yes	Apache2	Java	Web
WPSE [13]	Mix-IdP CSRF Privacy Leaks	A	Yes	-	JavaScript	Web
SSOScan [12]	Access Token misuse Signed request misuse Credential leakage via Referrer Secret Leakage Credential Leakage via page content	A	Yes	-	JavaScript	Web

Legenda: “A”: Automatic, “S”: Semi-Automatic, “M”: Manual, “-” : No Licence applied, “N/A” : Not available

[2]Authorization required.

4 Micro-Id-Gym: pentesting tool for OAuth/OIDC

In this section technical details will be discussed in order to explain how the tool works and why some decisions were made, describing each technology used in the implementation and its use. The core technology used for the implementation is **Burp Suite** [9] (hereafter Burp), a suite of security testing tools popular in the pentesting world.

We choose to use Burp for its feature of creating custom plugins, giving the developer the possibility of using the already available features to create new ones to expand the program. The languages possible to use to implement the plugin are java, ruby and python. Moreover Burp is an HTTP proxy, an application acting as intermediary between 2 other entities exchanging HTTP packets, functioning on behalf of the client, in our case the tool. This gives us the possibility of intercepting HTTP messages, meaning we are capable of displaying and modifying them, allowing us to develop any type of check on the service taken in consideration.

We used as programming language Java and the dependencies used in the plugin are composed only of Selenium WebDriver [11], a framework which can be used to automate user interaction with web browsers, and a simple library that allowed us to manipulate JSONs for OIDC's JWT.

Section 3.2 describes the review of the state of the art regarding the vulnerabilities and best practices for OAuth and OIDC which were the guidelines in our work for a complete list of checks needed in our plugin, but unfortunately some of them can not be included due to the rule set at the beginning of our project: the plugin has to work with as many web app as possible. Some of the test cases required an advanced knowledge of the implementation and the creation of custom sections for the plugin. This reason made impossible to implement the next vulnerabilities checks:

Tokens expiration and lifetime the lifetime of an access token should be short, not allowing attackers to use old ones much time after it was issued. This is difficult to test because of the wide variety of providers, each having a different approach to the use and naming of access tokens.

Authorization code lifetime similar to the previous one, authorization codes should be consumed shortly after its issue to obtain a valid access token. Testing whether a code can be used long after its issue can be troublesome due to the number of providers bringing the same problems stated in the previous test case.

Correct use of Refresh Token this tokens should be used shortly before the expiration of the access token and not abused. Testing whether a provider allows consuming a refresh token multiple times and long before is due is difficult due to the different API used by the different providers.

The implemented ones, based on the differences in their work-flow, can be divided into two separate groups:

Passive checks: checks which can operate in the background while the traffic passes through Burp and *DO NOT* need to change any HTTP message.

Active checks: checks which need to modify specific packets as they pass through Burp which means that each check needs to reiterate the entire authentication.

4.1 User Interface

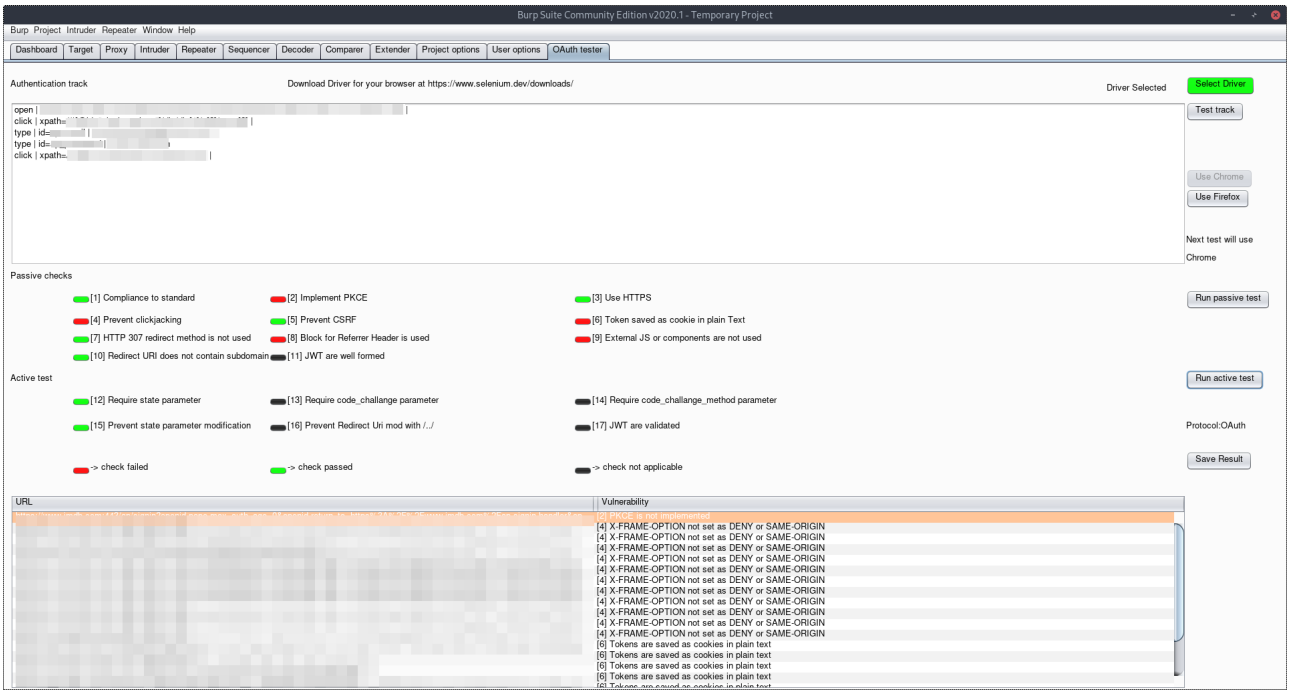

Figure 4.1.1: GUI - example after all checks run.

In Figure 4.1.1, it is depicted the plugin after a complete run of all the checks. The text area on the top is used to insert the authentication trace to guide the plugin trough the login phase and then on the landing page of the website. Below that portion you can find the GUI section which is used to show the user the result of the checks: both using buttons colored based on the tests outcome and a table containing a column for each **URL** which a vulnerability is found.

The available buttons are:

Select Driver button to select the binary file for Selenium WebDriver;

Test Track button to initialize a check on the track used to guide the plugin through the authentication process. It also check what protocol (OAuth/OIDC) is used in the web app taken in consideration;

Use Chrome/Use Firefox selects which web browser to use (Only Chrome/Chromium/Firefox compatible);

Run passive test it runs the passive checks;

Run active test it runs the active checks (some of them rely on the passive ones);

Save Result saves the result available in the bottom table as JSON file. If a file with the same name already exists, it is **not** overwritten and the save process is aborted;

The plugin can used in a few simple steps, which can be summarized in:

1. Choosing which compatible web browser to use.
2. Selecting the Selenium WebDriver binary with the *Select Driver* button.
3. Inserting the authentication trace.
4. Clicking the *Test Track* button to test the correctness of the track.
5. Clicking the *Run Passive Test*/*Run Active Test* button.
6. (Optional) Saving the result as JSON file.

In the center area of the plugin the user can find the name of each test that it can execute, each numbered from 1 to 17 and on its left a gray button. Each of them indicate the result of the test that has been run, at first gray signaling that it is yet to run. After a successful run of the checks, each button changes color matching the result of the check, green meaning it passed, red meaning it failed and black meaning it was not applicable (e.g. check 12 omits the use of the state parameter, if this parameter is not available in the considered implementation it can not be omitted so the check can not run).

4.2 Plugin Architecture

At the beginning of this project we decided to expand an already available Burp plugin, which was featured in **Micro-Id-Gym** [14], a training environment in which users develop hands-on experience on how Identity Management solutions work and the underlying security issues. This both helped our development phase, which meant including some simple features in the new plugin and implement the missing checks, but also the testing phase. Because Micro-Id-Gym included an implementation of both OAuth and OIDC, the testing phase was simply running the plugin on them, reducing the time required for this phase. The core structure of the base plugin has been maintained for its simplicity, modularity and possibility of creating new plugins destined to different protocols or standards.

During the flow of the plugin, what Selenium does is open a web browser, act on the commands received by the user (indicated by the authentication trace) and have Burp as HTTP proxy. This is done to ensure Burp manipulating the right traffic, which is the authentication and authorization of users using OAuth/ OIDC.

In the source code there are many Java files and many of them are simply interfaces required by Burp to include specific features, some examples can be the possibility of redirecting traffic or creating custom logs. The core files for the plugin to run are:

BurpExtender.java: the core class of the project. It handles all the methods needed from Burp and it interfaces it with all the other classes of the project. It also contains the section which manages the active tests.

ExtensionGUI.java this class handles the entire GUI of the plugin, including the result printing and the commands input. It also include all the code performed by Selenium WebDriver, both the track check and execution

TestTools.java a class container for all the utilities needed for both active and passive checks and application flow.

VulnItem.java a simple class which represent the information about a found vulnerability.

A complete list of all the source code files can be found in table 4.2.1.

Table 4.2.1: Source code Java files.

BurpExtender.java	IMessageEditor.java	**ExtensionGUI.java**
IMessageEditorTabFactory.java	IBurpCollaboratorClientContext.java	IMessageEditorTab.java
IBurpCollaboratorInteraction.java	IParameter.java	IBurpExtenderCallbacks.java
IProxyListener.java	IBurpExtender.java	IRequestInfo.java
IContextMenuFactory.java	IResponseInfo.java	IContextMenuInvocation.java
IResponseKeywords.java	ICookie.java	IResponseVariations.java
IExtensionHelpers.java	IScanIssue.java	IExtensionStateListener.java
IScannerCheck.java	IHttpListener.java	IScannerInsertionPoint.java
IHttpRequestResponse.java	IScannerInsertionPointProvider.java	IHttpRequestResponsePersisted.java
IScannerListener.java	IHttpRequestResponseWithMarkers.java	IScanQueueItem.java
IHttpService.java	IScopeChangeListener.java	IInterceptedProxyMessage.java
ISessionHandlingAction.java	IIntruderAttack.java	ITab.java
IIntruderPayloadGeneratorFactory.java	ITempFile.java	IIntruderPayloadGenerator.java
ITextEditor.java	IIntruderPayloadProcessor.java	**TestTools.java**
IMenuItemHandler.java	**VulnItem.java**	IMessageEditorController.java

4.3 Plugin flow

The plugin can be in 3 different states based on what button the user clicks: *Test track* when the track is tested for correctness, *Passive Test* when passive checks are run, *Active test* when active tests are run. From a implementation point of view, the plugin is relatively simple: the class `BurpExtender.java` interfaces with Burp to provide all the feature to read and manipulate HTTP packets. This class differentiates on the various cases in which the plugin can be (one excludes the others). This is possible due to having an instance of the `ExtensionGUI.java` class that contains a variable indicating what the current state is. This variable is a simple Integer that can denote when the correctness of the authentication trace is tested, passive checks during this type of test and each of the available active checks.

The approach described was chosen due to the active tests requiring to reiterate through the authentication trace multiple times and the possibility of adding new ones: it can be done by simply creating a new integer denoting the check and iterating one more time.

4.3.1 Authentication Trace

This test has been developed to check whether the authentication trace inserted by the user is written correctly, with the right commands and syntax. A description of what it can contain and its syntax can be found in table 4.3.1. This is performed by Selenium by reading the track line by line and executing the commands. If any type of error is encountered the action is stopped and the user is notified by appending an error signal on the text area.

This test also examine what of the two protocols the website is using by storing all the traffic. After that, every packet is reviewed for JWT, indicating the use of OIDC.

The authentication trace can be written manually, including every command to execute a successful login into the OAuth/ OIDC implementation, but this approach can be quite tedious and prone to errors. Luckily, a web browser extension called **Katalon** [1] has been developed that can easily create the authentication trace needed to run the plugin.

Table 4.3.1: Authentication Trace Description.

Commands available in the authentication trace	**Types of indicators**
open |http://www.example.com |	name
type |indicator=example |example	class
click |indicator=example |example	id
	xpath
	link
Authentication Trace Example	
open |http://www.example.com |	
click |xpath=//*[@id="signin-options"]/div/div[1]/a[2]/span[2] |	
type |id=ap_email |email	
type |id=ap_password |password	
click |name=continue |	

4.3.2 Passive and active checks

Regarding *passive checks*, the flow of the plugin is a simple execution of the authentication trace, scan of the traffic and decide whether the current packet should be or should be not stored to later perform tests. Obviously not all the traffic during an attempted login should be tested, there is a specific time frame in which HTTP packets are used to transport OAuth/OIDC data: `TestTools.java` includes all the methods used to recognize when this particular time in the protocols flow starts and ends. If some packets are recognized as useful, they are stored in a list of HTTPRequestResponse object in BurpExtender's instance of ExtensionGUI which will be scanned when the traffic is stopped for each check implemented.

On the other hand, *active tests* do not need any storing of traffic, they check in every packet whether any parameter or content in the body's message has to be modified or deleted and they differentiate between checks on the variable indicating the plugin state.

The way this plugin prints the result to the user is managed by the ExtensionGUI.java class and it works similarly between passive and active tests:

Passive tests: there is a mapping between each check and a series of methods which receives from them a list of `VulnItem.java` and based on its size changes the GUI (changing color to the specific button and adding the vulnerability found in the result table)

Active tests: also here there is a mapping between check and methods but the difference is that they need a boolean value, set by Selenium when the traffic stops based on if the url of the current page contains errors

5 Experimental analysis

A key aspect of the implementation stage was the experimental analysis: we had to decide how and on what platform we would check the newly implemented features. We decided to focus on a single web app which offers the possibility of logging in with multiple IDs. This allowed us to concentrate purely on the reliability of checks and not on the structure itself. The web app taken in consideration is present in the Top 100 websites list made by Alexa, which is calculated using a combination of average daily visitors and pageviews over 1 month. We decided not to disclose the web app and the identity providers names to ensure their information systems security. This is customary in the penetration testing world, the tester usually contacts the developers in case of a found vulnerability to avoid it being revealed to the public and possibly exploited.

Table 5.0.1 provides a graphical representation of the results of a complete run of the plugin, which gives an idea of what well-known providers is lacking from a security point of view. Every table row represents a check available in the plugin, everyone numbered from 1 to 17 matching their number to the ones in the tool. The first column indicates the name of the check and the others represent which ID is taken in consideration.

The result of the analysis showed in table 5.0.1 provided very important information regarding the deployments of renowned services:

0% prevents clickjacking: including the non-standard header *x-frame-option* set to deny or same-origin can avoid the clickjacking vulnerability. This can be exploited when an attacker creates a malicious site containing an iframe element in its HTML code covering a dummy button. This button can be clicked unintentionally by the user while clicking the iframe granting ;

0% blocks for referrer header: using links can ease the implementation of the protocols and the use of the service by the users. However, sometimes urls can contain user's codes or tokens because of faulty implementation, leading to possible replay attacks. The block of the referrer header is usually used to hide the origin of the HTTP request, but in this case can be used to delete the wrongfully inserted credentials;

0% avoids use of external JS or components: using Content Delivery Networks (CDN) can result in a faster service and implementation, but it also requires additional network use which can lead to an exploit;

0% implements PKCE; described in section 2.3.2, PKCE was developed specifically to avoid replay attacks;

33% do not use the state parameter: similar to PKCE, it has been created to avoid Cross-site Request Forgery (CSRF).

Attachment B contains pictures of the plugin after the tests described in this section.

Table 5.0.1: Table recap for analysis result.

Web App provider number	**1**	**2**	**3**
Passive Tests			
[1] Compliant to standard	🟢	🟢	🟢
[2] Implements PKCE	🔴	🔴	🔴
[3] Use HTTPS	🟢	🟢	🟢
[4] Prevent ClickJacking	🔴	🔴	🔴
[5] Prevent CSRF	🟢	🔴	🟢
[6] Token saved as cookie in plain text	🔴	🔴	🔴
[7] HTTP 307 redirect method is not used	🟢	🟢	🟢
[8] Block for Referrer Header is used	🔴	🔴	🔴
[9] External JS or component are not used	🔴	🔴	🔴
[10] Redirect URI does not contain subdomain	🟢	🟢	🟢
[11] JWT are well formed	⚫	🟢	⚫
Active Tests			
[12] Require state parameter	🟢	⚫	🟢
[13] Require code_challenge parameter	⚫	⚫	⚫
[14] Require code_challenge_method parameter	⚫	⚫	⚫
[15] Prevent state parameter modification	🟢	⚫	🟢
[16] Prevent Redirect URI mod with /../	⚫	⚫	⚫
[17] JWT are validated	⚫	🔴	⚫

🟢 check passed - 🔴 check failed - ⚫ check not applicable

6 Conclusions and future work

In this thesis we described what OAuth and OIDC goals are and why they are so important in the modern landscape of Information technology: they offer a simple solution to the wide variety of applications that need validated access to private resources. The growth of this type of applications in the recent years helped the adoption of these protocols, in both web services and mobile applications. Unfortunately they also dismiss the use of some of the security extensions/features that have been created to avoid the most common vulnerabilities.

Our contribution is first defining what every developer who wants to create a service or application using OAuth or OIDC needs to know, a set of current security practices to ensure a safe use of private data. This task has been completed by using different sources, to guarantee a comprehensive guideline based on both theoretical concepts and real-world scenarios. After that, we developed a tool able to detect in an automated way if the best current practices are included in the implementation taken in consideration. This can help both a user interested in the security of the available applications or services, but also to the developer in need for a way to test whether his/her deployed application or service is safe to use.

As future work, the scope of the plugin can be improved and extended to additional architectures. During our work we focused on web applications, accessible from web browsers, allowing their use from a wide variety of devices. We made this choice because of their heavy use and their availability on both desktop computers and mobile devices. Unfortunately this left out a great portion of OAuth and OIDC based applications, developed to work purely on mobile devices. Many of these are downloadable on famous app-store which make a lot of user vulnerable and making our plugin compatible with these applications will help this issue. Furthermore, an improvement to the ease of use of the plugin can be the porting of the checks implemented in it to a web browser extension, making it possible to surf the web and have immediate information regarding the security of the web sites in use.

References

[1] Katalon - an all-in-one test automation solution. https://www.katalon.com/?pk_abe=AB_testing_Homepage&pk_abv=layout1. accessed: may 2020.

[2] Linkedin oauth tester. https://github.com/evertrue/linkedin_oauth_tester. accessed: may 2020.

[3] Oauth - vuln scanner. https://gitlab.com/projet-2a/OAuth_vulnerabilities_scanner. accessed: may 2020.

[4] Oauth2-client-tester. https://github.com/tcannonfodder/oauth2-client-tester. accessed: may 2020.

[5] oauth2-redirector. https://github.com/tam7t/oauth2-redirector. accessed: may 2020.

[6] Oauthguard. https://github.com/oauthguard/OAuthGuard. accessed: may 2020.

[7] Openid foundation. https://openid.net/. accessed: may 2020.

[8] Owasp foundation - web application security. https://owasp.org/.

[9] Portswigger web security - burp. https://portswigger.net/burp. accessed: may 2020.

[10] Professos. https://github.com/RUB-NDS/PrOfESSOS. accessed; may 2020.

[11] Selenium - selenium automates browsers. https://www.selenium.dev/. accessed: may 2020.

[12] Ssoscan. http://ssoscan.org/. accessed: may 2020.

[13] Wpse. https://sites.google.com/site/wpseproject/. accessed: may 2020.

[14] Ivan Martini Valentina Odorizzi Giulio Pellizzari Silvio Ranise Andrea Bisegna, Roberto Carbone. Micro-id-gym: Identity management workouts with container-based microservices.

[15] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. Authscan: Automatic extraction of web authentication protocols from implementations. In *NDSS*, 2013.

[16] Ed. D. Hardt. https://tools.ietf.org/html/rfc6749, 2012.

[17] Ed D. Hardt. The oauth 2.0 authorization framework. https://tools.ietf.org/html/rfc6749, 2012.

[18] D. Hardt Independent M. Jones Microsoft. The oauth 2.0 authorization framework: Bearer token usage. https://tools.ietf.org/html/rfc6750, October 2012.

[19] N. Sakimura NRI M. Jones Microsoft, J. Bradley Ping Identity. Json web token. https://tools.ietf.org/html/rfc7519, 2015. accessed: may 2020.

[20] J. Bradley Ping Identity N. Agarwal Google N. Sakimura, Ed. Nomura Research Institute. Proof key for code exchange by oauth public clients. https://tools.ietf.org/html/rfc7636, 2015.

[21] Davide Piva. Assisting developers in securing oauth 2.0 deployment - demystifying threats and protection techniques for bearer credentials.

[22] Shangcheng Shi, Xianbo Wang, and Wing Cheong Lau. Mossot: An automated blackbox tester for single sign-on vulnerabilities in mobile applications. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 269–282, 2019.

[23] R. Shirey. Internet security glossary, version 2. https://tools.ietf.org/html/rfc4949, 2007.

[24] Kasidit Siriporn. Overscan: Oauth 2.0 scanner for missing parameters. In *Network and System Security: 13th International Conference, NSS 2019, Sapporo, Japan, December 15–18, 2019, Proceedings*, page 221. Springer Nature.

[25] Andrey Labunets Daniel Fett T. Lodderstedt, John Bradley. Oauth 2.0 security best current practice draft-ietf-oauth-security-topics-13. `https://tools.ietf.org/html/draft-ietf-oauth-security-topics-13`, 2019.

[26] A. Labunets Facebook D. Fett Facebook T. Lodderstedt yes.com, J. Bradley Yubico. Oauth 2.0 security best current practice draft-ietf-oauth-security-topics-13. `https://tools.ietf.org/html/draft-ietf-oauth-security-topics-13`, July 8, 2019.

[27] CSSLP / Nixu Corporation Teemu Kääriäinen. Owasp helsinki chapter meeting #30 october 11, 2016. `https://www.owasp.org/images/9/99/Helsinki_meeting_30_-_Threats_and_Vulnerabilities_in_Federation_Protocols_and_Products.pdf`, 2016.

[28] J. Bradley Ping Identity W. Denniss Google. Oauth 2.0 for native apps. `https://tools.ietf.org/html/rfc8252`, 2017.

[29] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. Authscope: Towards automatic discovery of vulnerable authorizations in online services. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 799–813, 2017.

Attachment A

Test Cases

URI redirection validation

Name	**URI redirection validation**
Description	Redirect URI should be validated, more specifically for pointing to internal domains or bypassing checks with /../ .
Test	1. R.O. starts OAuth flow 2. Client redirects R.O. to A.S. with redirect URI 3. Check whether A.S. accepts redirect URIs containing /../ or internal domains

Tokens stored as plain text in cookies

Name	**Tokens stored as plain text in cookies**
Description	Tokens should not be stored as plain text in cookies because they would be accessible to attackers in browser's cache.
Test	1. R.O. starts OAuth flow 2. Client redirects R.O. to A.S. with redirect URI 3. R.O. receives authentication code from A.S. 4. R.O. passes authentication token to Client 5. Client exchanges authentication token for access token with A.S. 6. Check whether Client stores access token as cookie in plain text on R.O.

Tokens expiration and lifetime

Name	**Tokens expiration and lifetime**
Description	Tokens' expiration time should be specified and its duration should be less than 10 minutes.
Test	1. R.O receives authorization code 2. Client receives authorization code from R.O. 3. Client exchanges authorization code for access token 4. Check wheter Client can use access token for more than 10 minutes

Authorization code lifetime

Name	**Authorization code lifetime**
Description	Authorization code lifetime should be short, usually less than 10 minutes.
Test	1. R.O. receives authorization code 2. Client receives authorization code from R.O. 3. Check whether Client can obtain access token 10 minutes later than when authorization code was issued

JWT validation

Name	**JWT validation**
Description	JWT are used in OpenID Connect for user information and authentication session. Keys that must be validated: *iss*,*aud*, *id token signature*,*exp*,*iat*,*auth_time*. JWT algorithm type need to be checked for "none" value
Test	1st check: Check every key mentioned when Client receives JWT 2nd check: Check wheter A.S. accepts JWT with algoritmh type "none" as value

Correct use of refresh Token

Name	**Correct use of refresh Token**
Description	A refresh token, coupled with a short access token lifetime, can be used to grant longer access to resources without involving end-user authorization but it should be used only when needed.
Test	1. R.O. starts Oauth flow and obtains authorization code and sends it to Client 2. Client obtains access token 3. Check whether Client can repeatedly request new access token using refresh token even if the one in use is not expired

Use of HTTP 307 status code

Name	**Usage of HTTP 307 status code**
Description	The use of HTTP 307 status code can lead to user's credential leak from the login page.
Test	Check if any HTTP redirect code is 307

Incorporating external JavaScript/components

Name	**Incorporating external JavaScript/components**
Description	Loading external JavaScript or components from CDNs can lead to use of malicious code.
Test	Check for use of external JavaScript or components in any page

Leaking authorization code through referrer head

Name	**Leaking authorization code through referrer head**
Description	Referrer header can be used to obtain code, access token or state value. This can be disclosed unintentionally by both Clients and A.S.'s web site
Test	Check for the setting of referrer header in Client's or A.S.'s pages

Credential / Sensitive data / Secret data disclosure

Name	**Credential / Sensitive data / Secret data disclosure**
Description	If the network used for the protocol is not secure, data can be disclosed (TLS/SSL needed).
Test	Check whether any role of the protocol does not use TLS or SSL, if so they are susceptible to data/credential leakage.

Attachment B

Pictures of the plugin after a complete execution on different IDs

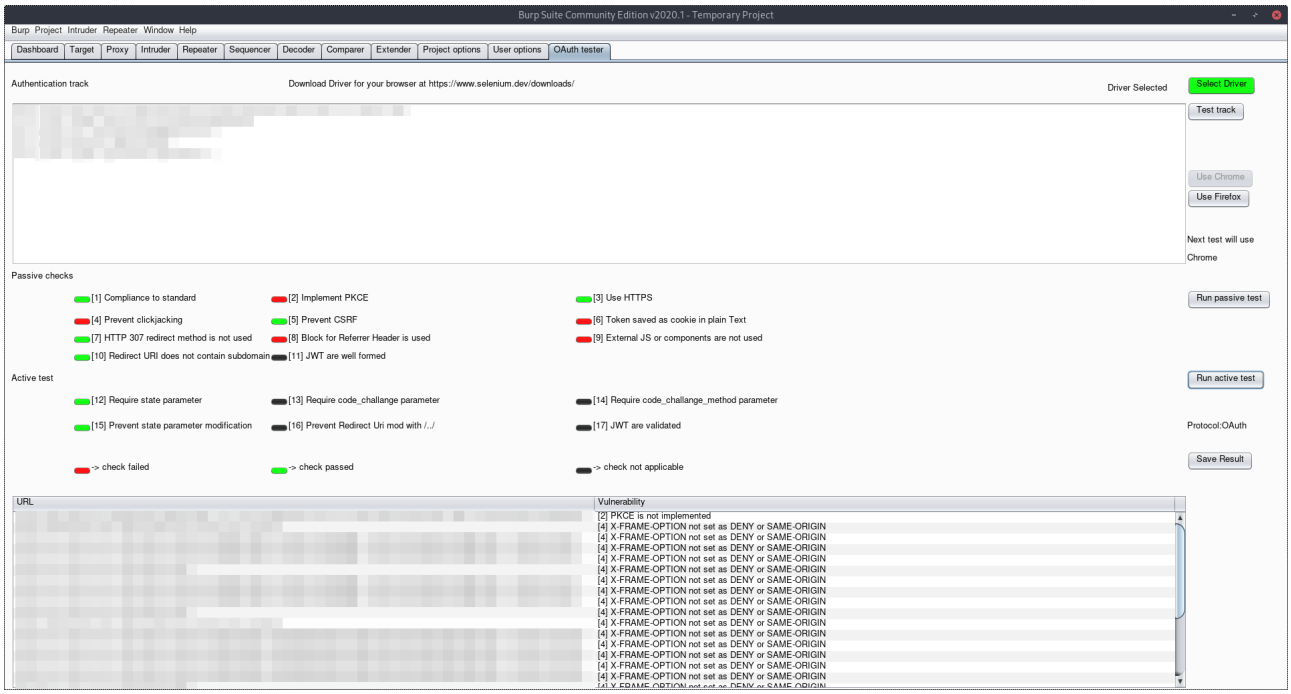

Figure 6.0.1: Checks result web app - provider 1.

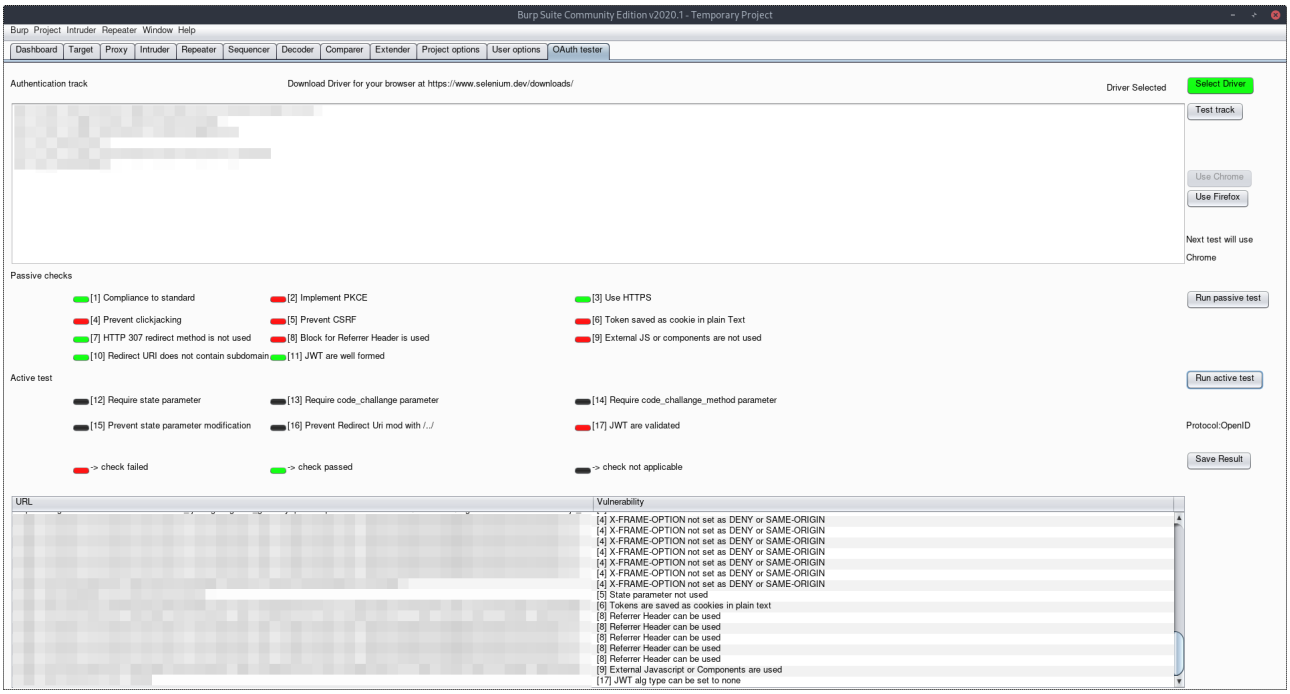

Figure 6.0.2: Checks result web app provider 2.

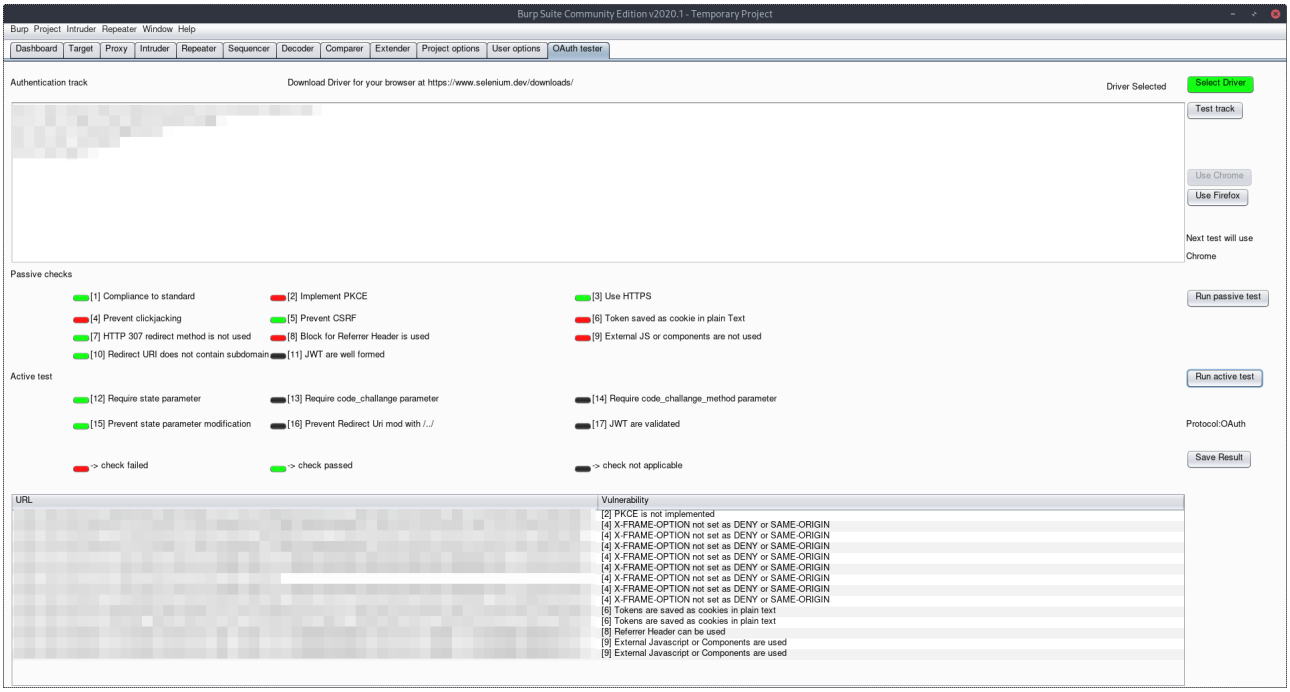

Figure 6.0.3: Checks result web app - provider 3.