Departement of Information Engineering and Computer Science

Bachelor's Degree in
Computer Science

Final dissertation

Declarative Specification of Pentesting Strategies for Browser-based Security Protocols: the Case Studies of SAML and OAuth/OIDC

Supervisor

Silvio Ranise

Student

Matteo Bitussi

Co-Supervisors

Andrea Bisegna

Roberto Carbone

Academic year 2021/2022

Acknowledgements

Thanks to Professor Ranise for the opportunity of working in the cybersecurity field with a real-world problem to solve.

Thanks to Andrea and Roberto for the help and the advice during all this work.

Thanks to my family for always introducing me to new things since I was young, for always promoting my ideas and creativity, and for supporting me and my studies.

Thanks to Alice for the help and support during the long exam sessions and the writing of this thesis.

Thanks to all the E-Agle TRT Racing Team, for the amazing experience, the infinite amount of knowledge it gave to me, the opportunity to work with expensive racing devices, the fun and emotions of building a car from scratch and seeing it racing, and for the lateness in my graduation.

Contents

Abstract

This thesis deals with the work started in my internship at Fondazione Bruno Kessler (FBK) in the context of browser-based protocols (based on HTTP) security testing. In the last years, there has been a constantly rising number of services transitioning from physical to virtual, such as health care and government services, also due to the Covid-19 pandemics. This was done to simplify some procedures and make them faster to be accomplished. However, some security and privacy concerns arise from this transition. A high quantity of sensitive data could be at risk of unwanted access. Software and security engineers have to implement the service's software and manage the data that the software is working with. They will rely on browser-based security protocols to ensure that security requirements such as user identity, the connection between the service and the user, and the integrity of data are met.

To ensure that all these security requirements are respected during the implementation of the service, there is the need to test the implemented services against the well-known vulnerabilities of the protocols used. Usually, to do that, a software engineer or a security tester, which will be called tester below, will act like an attacker trying to see, edit, and deny access to the data in the services. The tester will use a software to intercept the HTTP messages (from now on messages) between the service and the tester's browser. It will be possible for the tester to edit or remove the messages, simulate an attack, and see if the service is vulnerable. The tester will have to gather all the well-known vulnerabilities related to the used protocols and test them manually. The pentesting is time-consuming and could be done improperly if the tester is not qualified to do it. A software is needed to automatically test the services against the well-known vulnerabilities, which could give a result of whether the services are vulnerable.

The work in this thesis is about the design and implementation of the automatic HTTP pentesting software (from now on tool) and its relative declarative specification language. The tool addresses the challenges highlighted above, such as the automation of the pentesting phase to improve its precision and the design of a language for the specification of the tests. Other software has been evaluated before the start of this work, some of them missed the automation part, others were limited in terms of the possible security tests (from now on tests) to be done. For these reasons, a tool and language have been designed and implemented. First, a declarative specification of pentesting strategies has been designed, simultaneously with the design of an approach to execute the pentesting strategies. Next, the implementation of a tool to execute the pentesting strategies has been done. To test the correctness of the implemented tool and the expressiveness of the proposed specification language, a test definition for OAuth/OIDC, a web security protocol, has been written in the implemented language and used during development.

The implemented tool and language have proved to be working in two different use cases, with OAuth/OIDC and SAML. The results obtained by the tool have also been confirmed by comparison with the results of similar software; they have proved to be correct. Lastly, the objectives fixed at the beginning of the work were accomplished, the software and language created are valuable tools to help the tester discover vulnerabilities in services. The definition of an OAuth/OIDC and SAML test suite covering the well-known vulnerabilities is useful as the tester does not need to write them every time to test them. Depending on the service to test, what has to be changed are the automation actions to be executed during testing, as the web pages may be different, so the actions to be accomplished have to be different. Future work has been planned to extend the tool with other functionalities and support other protocols.

Glossary

Burp	Burp Suite Community Edition, a web security testing software. 6, 8, 18, 21, 23, 27
OAuth	OAuth, is a SSO protocol. 3, 5, 8, 9, 11, 21, 25, 27, 28
OIDC	Open ID Connect, is a SSO protocol. 21, 25, 27, 28
PKCE	Proof Key for Code Exchange, an extension of OAuth. 11, 12, 14, 15, 18–20
SAML	Security Assertion Markup Language, is an SSO protocol. 5, 8–10, 23, 25
session track	The list of action the browser has to follow. 19, 20, 23, 24, 28

1 Introduction

In the last years, we have seen a constant transition from physical to virtual. This is the case of bank transactions, government documents, health care data, and almost anything that can be virtualized. This is a significant step forward to access services and resources anytime and anywhere. However, there are also some concerns about the security of services and data, that being virtual, can be accessed by some authentication protocols instead of a physical identification of the subject accessing the physical documents. This means that it has to be checked whether the person accessing the service or data is the one that is claiming to be. The three basic principles over which data security is based are confidentiality, integrity, and data availability. This principle is called the CIA triad [6]. If those requirements are not met, sensitive data could be made inaccessible or at the risk of access from unauthorized parties. As a result, critical services for health care could be inaccessible, and this is what happened for example in Lazio [13, 7], Italy, in August 2021, where, because of a RansomWare infecting the internal computer network of the region and keeping it down for multiple days, the covid vaccines and other health-care services could not be booked. Moreover, many data in the systems have been encrypted and denied access. The attack was possible due to a stolen set of privileged credentials used to log into the system. It is not clear how the credentials were stolen, maybe with a phishing attack or in another way. Even if this particular attack was probably not due to a vulnerability exploit, this is an excellent example of what the consequences could be.

In this thesis, the considered services are web services (from now on service) that rely on browser-based security protocols. Browser-based protocols are those used to secure web services communications, authentications, and identification. The development has been done with a particular focus over Identity Management (IdM) protocols such as SAML and OAuth/OIDC but without limiting the possible applications to these protocols. To ensure that the CIA triad requirements are satisfied, there is a need to test all the implementations of the protocols used by the service that is liable for the security of the data. The services that provide access to the data have to be appropriately designed and tested by a software engineer or a security tester (from now on tester) at least w.r.t. a set of well-known vulnerabilities. These are vulnerabilities of the protocols that have been discovered over time and that are known to exist. The service should be built to avoid them and should be tested against them. However, there could arise some problems, for instance, the tester should be qualified to test the protocols and should know how they work. This is not always the case, as software engineers are sometimes responsible for that. Another problem is that due to the fast-evolving nature of protocols, new vulnerabilities will eventually be found, and so, new tests will have to be defined, or old tests will have to be edited. This is much work to be done by the tester. The result is that some vulnerabilities could remain undiscovered, representing a weakness in the system. This thesis proposes an automated testing software and a language to help the tester to test the web protocols used in the services mentioned above.

Some software to test specific protocols and vulnerabilities already exist [2, 11], but they are usually very specific and hard-coded in a way that the tester could not easily edit the tests. Other tools exist, but they are not usually automated. With this work, I wanted to create a software that could test any browser-based protocol and abstractly specify the tests, allowing editing over time. To do this, a language will be used to specify the tests, and a software will be implemented to execute them. The test results will be evaluated by an oracle, which is composed of a series of components that are responsible for telling if the test is passed or not.

1.1 Contributions

The contributions of this thesis are the following:

- *Design of a declarative specification language for pentesting strategies*: The pentesting strategies

are the set of actions done on messages exchanged by protocols to verify if the service implementing the protocol is vulnerable to known vulnerabilities. The objective was to design a language that could define the necessary pentesting strategies. The language supports all the possible actions needed by a pentester to edit the messages, for instance, remove, add, and edit to be done on parts of the message. To verify if the service is vulnerable, there are also dedicated actions in the language to check the content of the messages. For example, to verify that the service is not vulnerable to a tested vulnerability, there could be the need to check the value or the presence of a specific parameter in the content of a specific message. This is done using a specific action in the language. The language also manages the browsers, making it possible to work on multiple sessions at once and, for example, to reply messages from one browser session to another. The previously introduced language was then implemented using JSON as a base language. The goal was to have a software that could automatically execute the specified tests. To do that, a series of actions (among those specified by the language) would have to be specified and then executed in a browser. The messages exchanged by the browser needs to be intercepted, and the tests have to be executed over these intercepted messages to give a result. For executing the actions on the browser, there was an already working solution [2, 11] that has been taken as a starting point. My contribution was to design the test execution and how the tests would interact with the browser and to extend the action execution to manage multiple sessions and new actions.

- *Design and Implementation of a tool to execute the pentesting strategies*: The previously introduced language to execute tests against browser-based protocols is implemented. The Burp Suite (from now on, Burp), a security testing software for web security testing, has been chosen as the platform to be used as the ground for the implementation; it is based on a browser and a proxy that intercepts the traffic from it. It admits the execution of plugins written in Java that use a set of APIs to interact with the proxy. The tool will then be integrated as a plugin for Burp. A dedicated interpreter has been built in the tool to parse the tests written in the language. The parsed test is executed with the primitive actions in the browser. The tool is able to intercept the traffic from the browser using the Burp's APIs. Then, based on the language actions, it manipulates the messages before sending them back to the proxy. The last step is the execution of the checks specified in the tests; an oracle has been implemented using the checks defined in every test to evaluate the tests' correctness and give a result.

- *Specification of a test suite for OAuth and OIDC protocols*: A test suite containing the well-known vulnerabilities of OAuth and OIDC protocols have been defined using the language. All of the OAuth/OIDC tests defined in [11, 2] were considered and defined with the language, and new tests were also introduced, given the new possibilities introduced with the language and the tool. Overall, 12 tests were translated, and 6 were added. This was done to validate the expressivity of the language and its functionality. The new defined tests were by prevalence actives, they could be added because of the new features introduced by the language, such as the new multi-session environment and the possibility of saving the values of parameters in variables.

- *Experimental analysis of the OAuth and OIDC test suite*: The defined OAuth and OIDC test suite analysis is accomplished. It has been used during the development of the tool to validate its functioning, comparing the obtained results with the results of different tools and manually checking the results over multiple protocol implementations to check its validity. The experimental analysis has been conducted over a test implementation of the protocol and some real-world implementations.

1.2 Structure of the thesis

The thesis is structured as follows:

- **Chapter 2**: A brief introduction over the concepts and software used in the rest of the thesis is provided.

- **Chapter 3**: The design of the tool and the testing language is given.

- **Chapter 4**: The implementation of the tool and the testing language is described.
- **Chapter 5**: Some examples of application of the software and the language described in this thesis are illustrated.
- **Chapter 6**: Other works related to the one of this thesis are discussed.
- **Chapter 7**: Conclusions, results, and works that could be done in the future are briefly commented.

2 Background

This chapter will discuss the subjects needed to comprehend the rest of the thesis. The most common subjects will be ignored, focusing on the more specific and less common ones.

2.1 Burp Suite community edition

Burp Suite Community Edition (Burp) is one of the most used application security testing software for web security testing. It works by using a proxy server over which a browser redirects the traffic. The proxy does like a Man In The Middle attack, taking the input traffic from the browser and replying the HTTP messages to the target service, also giving transparency over the TSL (Transport Layer Security) or SSL (Secure Socket Layer) encryption. Burp has access to the proxy; it can sniff HTTP packets and can edit them before they are forwarded to the browser or the target service. Burp also gives the possibility of creating custom plugins giving the developers access to the java API. This is precisely how the plugin (from now on tool) will be implemented, using Burp as a base over which to develop the software that will execute the tests. The tool will be able to intercept, read and edit messages that pass through the Burp's proxy by the use of Burp's API.

2.2 JSON

As stated in [4]: "JavaScript Object Notation (JSON) is a text format for the serialization of structured data. It is derived from the object literals of JavaScript. JSON can represent four primitive types (strings, numbers, booleans, and null) and two structured types (objects and arrays). An object is an unordered collection of zero or more name/value pairs, where a name is a string and a value is a string, number, boolean, null, object, or array. An array is an ordered sequence of zero or more values". In this thesis, the word "name" identifying a name/value pair will not be used and will be substituted with "tag".

2.3 Regex

Regex stands for regular expression, it is a sequence of symbols that define a group of strings that can be matched. There is a list of symbols that can specify which characters to match. For example, if the value of the "Host" header in a message has to be found, a regex like this one has to be defined: `Host:\s?.*` This regex will search for the string "Host:" followed by 0 or 1 whitespace, and all the characters that follow until "`\n`" is found. This is a complete explanation of the symbols used in this example:

- `\s` is the whitespace character.
- **?** is an operator that tells that the preceding symbol will be matched 0 or 1 times.
- ***** is an operator that tells that the preceding symbol will be matched 0 or more times.
- **.** matches any character except line breaks.

2.4 IdM protocols: SAML and OAuth

Identity Management (IdM) protocols are protocols that deals with identity management. In this thesis, SAML (Security Assertion Markup Language) and OAuth 2.0 (from now on OAuth) will be introduced.

Both SAML and OAuth are Single Sign-On (SSO) protocols, they follow an authentication scheme that allows a user to log in with a single ID and password to any of several related, yet independent, software systems, a complete explanation can be found in [11].

2.4.1 OAuth

As stated in [12], the OAuth authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its behalf.

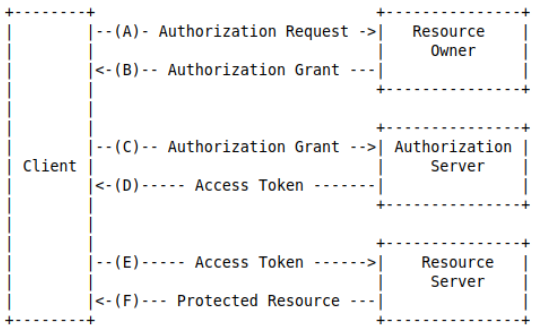

Figure 2.1: OAuth abstract protocol flow source [12]

A series of messages must be exchanged between the two parties to authenticate a resource owner who wants to access reserved data in a service. As described in [12], the flow illustrated in Figure 2.1 describes the interaction between the four roles and includes the following steps:

(A) The client requests authorization from the resource owner. The authorization request can be made directly to the resource owner (as shown) or preferably indirectly via the authorization server as an intermediary.

(B) The client receives an authorization grant, which is a credential representing the resource owner's authorization, expressed using one of four grant types defined in this specification or using an extension grant type. The authorization grant type depends on the method used by the client to request authorization and the types supported by the authorization server.

(C) The client requests an access token by authenticating with the authorization server and presenting the authorization grant.

(D) The authorization server authenticates the client validates the authorization grant and if valid, issues an access token.

(E) The client requests the protected resource from the resource server and authenticates by presenting the access token.

(F) The resource server validates the access token and, if valid, serves the request.

PKCE

As stated in [14]: OAuth 2.0 public clients utilizing the Authorization Code Grant are susceptible to the authorization code interception attack. A technique to mitigate against the threat is the use of Proof Key for Code Exchange (PKCE, pronounced "pixy"). The PKCE extension utilizes a dynamically created cryptographically random key called `code_verifier`. A unique `code_verifier` is created for every authorization request, and its transformed value, called `code_challenge`, is sent to the authorization server to obtain the authorization code. The authorization code obtained is then sent to the token endpoint with the `code_verifier`, and the server compares it with the previously received request code so that it can perform the proof of possession of the `code_verifier` by the client. This works as the mitigation since the attacker would not know this one-time key since it is sent over TLS and cannot be intercepted. The PKCE specification adds additional parameters to the OAuth 2.0 Authorization and Access Token Requests:

(A) The client creates and records a secret named the `code_verifier` and derives a transformed version t(`code_verifier`) (referred to as the `code_challenge`), which is sent in the OAuth 2.0 Authorization Request along with the transformation method "t_m".

(B) The Authorization Endpoint responds as usual but records t(`code_verifier`) and the transformation method.

(C) The client then sends the authorization code in the Access Token Request as usual but includes the `code_verifier` secret generated at (A).

(D) The authorization server transforms `code_verifier` and compares it to t(`code_verifier`) from (B). Access is denied if they are not equal.

An attacker who intercepts the authorization code at (B) is unable to redeem it for an access token, as they do not own the `code_verifier` secret.

2.4.2 SAML - Security Assertion Markup Language

As stated in [5], "(SAML) 2.0 is an XML-based framework that allows identity and security information to be shared across security domains. The Assertion, an XML security token, is a fundamental construct of SAML that is often adopted for use in other protocols and specifications. An Assertion is generally issued by an Identity Provider and consumed by a Service Provider that relies on its content to identify the Assertion's subject for security-related purposes."

3 Design of the testing language

This chapter introduces how the testing software (from now on tool) and the language introduced in Chapter 1 have been designed. The testing language specifies what a test does. A test is a series of verifications or modifications done on one or more HTTP messages (from now on messages) exchanged by the client and the service using the browser-based protocol. After the execution of a test, a check of the expected conditions has to be done, and the result has to be given to tell if the test has been successfully passed or not. The language will be defined based on all the possible actions that a security tester would need to do on the messages. One of the objectives was to think of a language that could specify the tests once and make it possible to test them against multiple web services. For example, a series of tests to verify the well-known vulnerabilities of a specific protocol could be defined and then used on any web service.

3.1 Choosing a language standard

A standard to follow while writing the tests had to be found; the first option was to define a formal language with its dedicated parser, this option was discarded, as it was challenging to implement and other already existing alternatives were available. The one that has been chosen is JSON (JavaScript Object Notation), which allows specifying the hierarchical structure of the tests using easy syntax writing. Also, its Object-oriented Notation is very useful, as tests and its components will be defined as Objects.

3.2 Example: PKCE Downgrade Test

I want to introduce the language with an example. Due to its complexity, having a concrete example before explaining all its components could be helpful to understand its function. The scenario of this test is based on the presence of a web service hosted in a server (client), the victim, which is using a browser to connect to the client, the Authorization Server (AS) and the tester, that is doing a Man-In-The-Middle attack between the web service and the victim, and between the victim and the AS.

The PKCE Downgrade test has as objective to test an OAuth vulnerability were removing the parameter `code_challenge` from the url of the authorization request message will be downgrading the authentication process in a way that PKCE will not be used by the AS if the AS is vulnerable [14]. This will allow the attacker to steal the authorization code from the AS. The message sequence chart of the attack can be seen in Figure 3.1.

To test this, the browser will execute the victim's actions, doing a login on the client. The authorization request message has to be intercepted, the `code_challenge` parameter has to be removed from it, and then the message has to be forwarded. The test is passed if the AS does not admit an exchange without PKCE; for instance, it can return an error page, which means that the AS is not vulnerable. Otherwise, if the AS is vulnerable, the exchange would continue without any error pages or messages. The complete test can be found in Attachment A

3.3 Language structure

Each part of the language is an Object, which is then written as a JSON Object in the language that contains a series of name/value couples. The hierarchical structure of the language can be seen in Figure 3.2. An example of a Test Suite Object containing other Objects can be seen in Figure 3.3. In this section, all the Objects composing the language will be discussed.

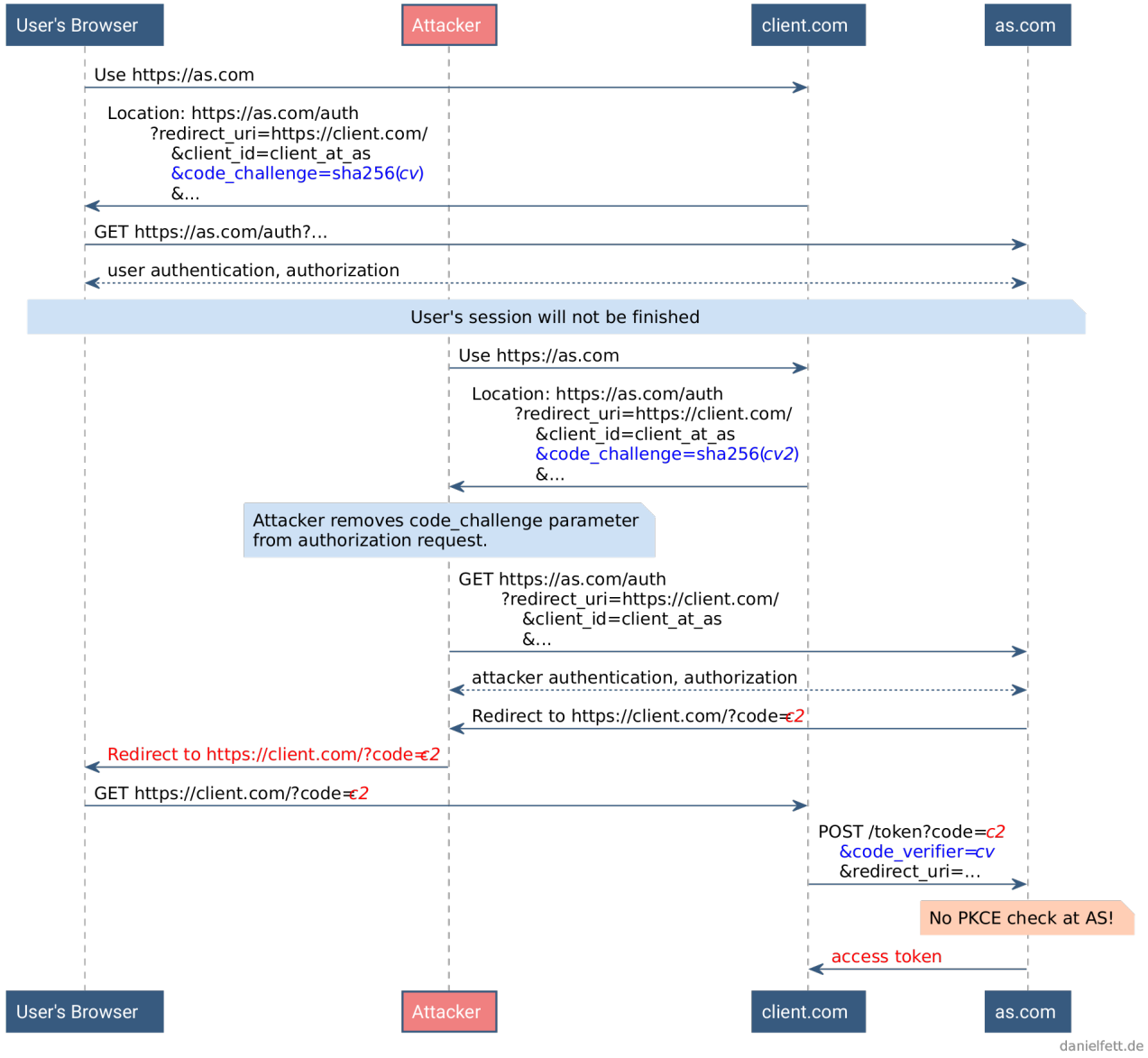

Figure 3.1: PKCE Downgrade. Source: Daniel Fett [9]

3.3.1 Test Suite

The Test Suite is the main Object which contains all the other ones, it has these tags:

- **name**, the name of the test suite.
- **description**, the description of the test suite.
- **tests**, which is a list containing the tests to be executed.

To implement the PKCE test example above, the Test Suite can be defined as shown in Listing 3.1. The Test Suite Object is a JSON object having the tag **test suite**, with name and description, and a tag *tests* which will contain a list of Test Objects.

```
1    {
2        "test suite": {
3            "name": "OAuth active tests",
4            "description": "A test suite containing a OAuth test"
5        },
6        "tests": [
7            {
```

```
8                   // A test
9               }
10          ]
11      }
```

Listing 3.1: Test Suite definition

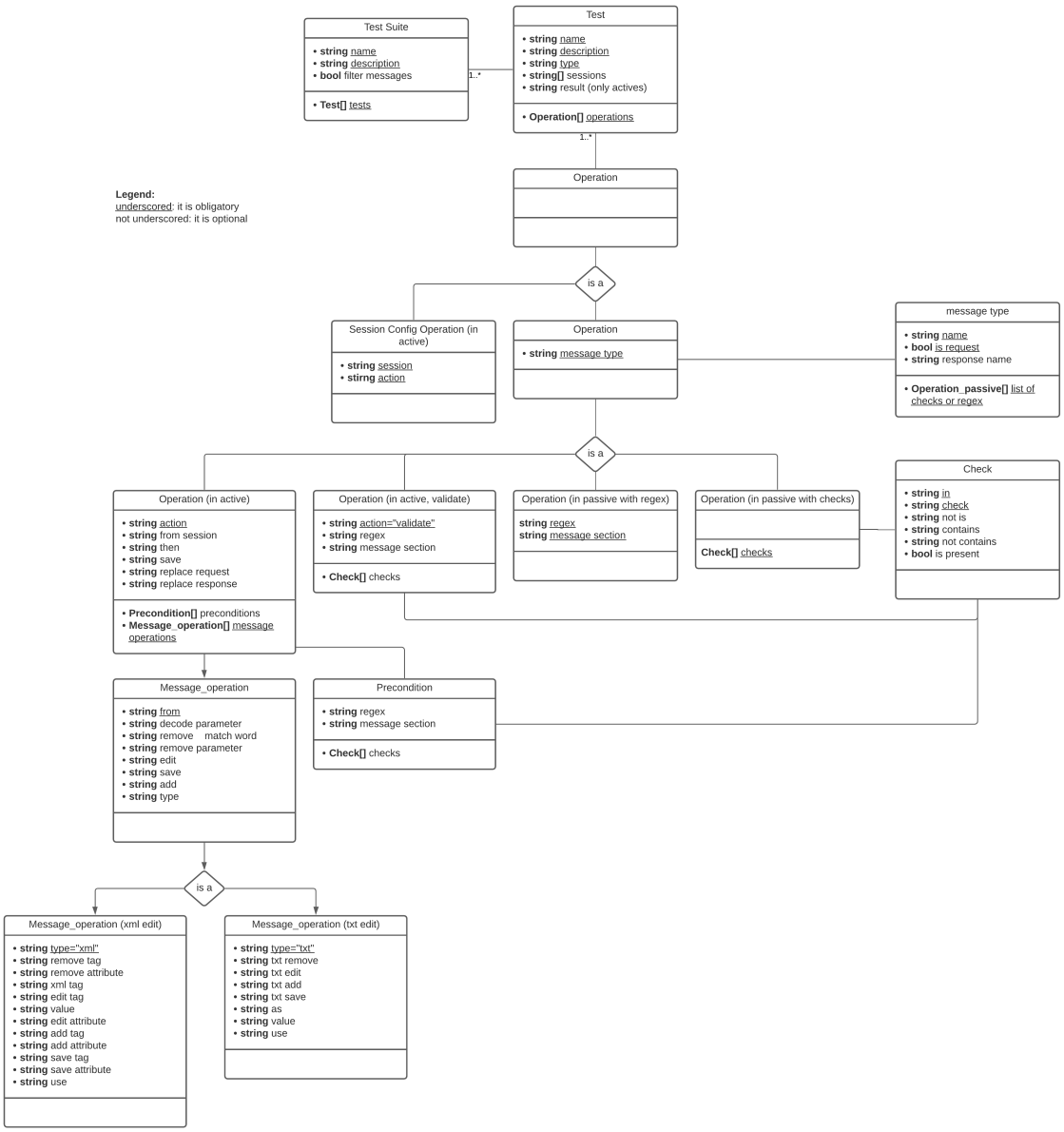

Figure 3.2: Language structure

3.3.2 Test

Following the hierarchical order, the Test object is the one that actually defines a test. As said earlier, a test is contained in a Test Suite, and contains various tags to be defined:

- **name**.
- **description**.
- **type**, it can be *active* or *passive*.
- **sessions**, which is a list of the sessions which are needed in this test.

- **result**, (only for actives) it defines the conditions over which the test is considered passed or not.
- **operations**, a list of Operation objects which will be executed.

A test can be defined either as active or passive depending on the type of actions it has to do on the intercepted messages. If a test does not need to manipulate the flow or the content of the messages, it is considered passive; otherwise, it is considered active. The list of Operation Objects contained in a Test is executed iteratively one after the other, so only after an Operation is completed the next one starts.

Following the PKCE example introduced in Section 3.2, an active Test is defined:

```
{
    "test": {
        "name": "PKCE Downgrade",
        "description": "Tries to remove code_challenge parameter",
        "type": "active",
        "sessions": [
            "s1"
        ],
        "operations": [
            // list of Operation Objects
        ],
        "result": "incorrect flow s1"
    }
}
```

Listing 3.2: Active test definition

As can be seen in Listing 3.2, the test is specified, writing its type, which is active, as it has to edit a message removing a parameter from the url. Moreover, a session and a list of Operations to be executed are added. The result of the test is also specified.

3.3.3 Operation

The Operation Object defines the actions that a test has to do. As shown in the language structure in Figure 3.2, an operation could be either a **standard operation** or a **session config operation**, the latter is used to manage the sessions for the active tests (i.e. start, stop, pause them). Depending on the type of test in which an Operation is defined, the standard Operation becomes active or passive. In both cases, an operation has to contain the **message type** which defines the type of message to be intercepted in that particular Operation (more info in the dedicated paragraph).

A **passive** operation has as objective to verify the presence (or absence) of some text or parameters in the intercepted message. To do this, it should contain one of the following options:

- A list of **Check** Objects, which are then executed to check the presence (or absence) of some text or parameter.
- A **regex** inspection, which executes an inspection considering the intercepted message as plain text and executing a regex over it; if the regex has a match, the Operation is considered passed, otherwise failed. Note that when a regex is used, it has to be also specified the **message section** over which to execute it (body, head, url).

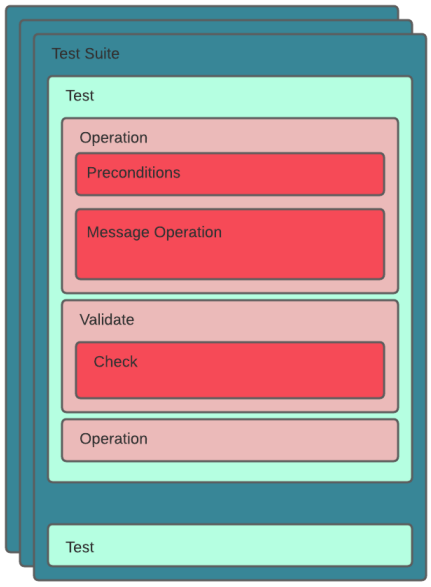

Figure 3.3: language structure

If the Test Object where the operations are defined is an **active** test, also, if the intercepted messages need to be manipulated, an active Operation has to be defined. It is composed by:

- **action**, the action it has to do (*intercept*, *validate*).
- **from session**, from which session to expect the message to be intercepted.
- **then**, the action to do after the receiving and manipulation of the message (forward or drop).
- **replace request (or response)**, specify a previously saved message in order to replace it to the intercepted one.
- **preconditions**, a list of Precondition objects.
- **message operations**, a list of Message Operation objects, which will do the actual manipulation of the intercepted message.

If the action is set to *validate*, the Operation is still active. However, it becomes like a passive Operation because its objective is just to verify that some messages meet some requirements. It will contain a regex or a list of checks to be done. The Validate Operation Object is part of the oracle, which is the component that decides whether the tests should be considered passed or not. More details will be discussed in the dedicated section.

In the PKCE example seen in Section 3.2, an Operation Object for an active test is specified. This Operation has to intercept the *authorization request* message from session *s1*. It has to check some Preconditions over it and then do some Message Operations. In the end, the message is forwarded to its original destination.

```
{
    "action": "intercept",
    "from session": "s1",
    "then": "forward",
    "message type": "authorization request",
    "preconditions": [
        // Precondition list
    ],
    "message operations": [
        // Message Operation list
    ]
}
```

Listing 3.3: Operation definition

3.3.4 Message section

The message section specifies in what part of the message to execute the given action. According to [10] the parts that compose an HTTP message are:

- **url**, a start line describing the message.
- **head**, a block of headers containing attributes.
- **body**, an optional body containing data.

3.3.5 Check Object

The Check Object is used in Operations Objects and in other Objects to verify that, in a message, some circumstances are satisfied. For instance, it can be used to check that a specific parameter in the url of a message should be equal to a specific string. The Check Object is defined by:

- **in**, where to search the given parameter. The possible value is *head*, *body* or *url*.
- **check param**, specifies the parameter name to be searched.

Also, the actual checks on the parameter value have to be defined: (if none of these is defined, the Check will only check if the given parameter is present or not in the given section)

- **is**, checks that the parameter value is exactly what is passed to this tag.
- **not is**, checks that the parameter value is not what is passed to this tag.
- **contains**, checks that the parameter value contains what is passed to this tag.
- **not contains**, checks that the parameter value does not contain what is passed to this tag.
- **is present**, used to explicitly tell to just check the presence of the parameter, it accepts true or false, depending on if, respectively, the presence or the absence of the parameter has to be checked.

Note that by how the logic of the language has been thought, to consider a test passed, all the check Objects has to be evaluated to true. For instance, if a parameter should not be present in a given message, the Check has to verify that the parameter is **not** present. If a Check is evaluated to false, the test will fail.

3.3.6 Precondition and Validate Objects

In active tests, Check Objects will not work as in passive tests, there is another way of using them: using them in a Precondition, which is a list of the Check Objects, or by using the validate option in an Operation Object, which will make possible to use Checks to validate a given message. This difference of Check Objects between active and passive tests has been made to allow the Oracle to work, differentiating between a precondition and a validation. This way, Validate Objects can be used to define the oracle, and Precondition Objects can be used to impose the criteria to allow the execution of the test.

Precondition Object Preconditions are used in an Operation of an active test to check that the intercepted message complies with specific criteria fixed by the tester before executing the Message Operations. If the Check objects in the precondition are evaluated to false, the test is considered unsupported, not failed. More precisely, the preconditions will be a list of Check Objects. This can be useful in case it is not known if a given test is compatible with a given web service. For example, if the service does not use some protocol, this can be assured by using preconditions checking for the presence of common parameters of the protocol. Preconditions can also be a regex.

Validate Object In an Operation Object, the only way to use Check objects is by setting the **action** tag to *validate*. This will transform the Operation into a Validate Operation. The Oracle will use this Validate Operation to decide whether the test should be considered passed or not. Validate Operation has to be used when a given part of a message should be in a specific way. If it is not, the test result will be considered failed. In order to do this, the Validate Operation with a list of checks or a regex (exactly like in passive tests) can be used.

3.3.7 Save

A message or a string can be saved by the use of the tag **save**. This can be used both in an Operation to save an entire message or in a Message Operation to save the value of a found parameter. So a variable can be a message-type variable containing an entire saved message or a string-type containing a string. There are two ways of using the value of a variable which depends on its type:

- Using a **message-type** variable: it can be used in an Operation with the tag **action** set to intercept. There is the possibility of using **replace request** (or **replace response**) tag giving the name of the variable. This will replace the intercepted message's request (or response) with the message saved in the variable.
- Using a **string-type** variable: can be used in Message Operations, where a parameter has to be edited, writing **use** tag specifying the name of the variable to use. This will use the value in that variable in the way specified by the other tags (i.e., tags **edit** or **add**).

3.3.8 Message Operation

The Message Operation is the Object that actually does the manipulations on the intercepted messages. It is composed by these tags:

- **from**, the message section to work on.
- **decode parameter** (optional) it indicates which parameter's value or string to be decoded before it can be processed.
- **encodings** (optional) the list of encodings to be applied to the parameter or text to be decoded. The supported encodings are base64, deflate, url.
- **remove match word** (optional), it accepts a regex, everything matched by that regex in the specified message section is removed.
- **remove parameter**, it accepts a parameter name, it removes both the name and the value of that parameter from the specified section.
- **edit**, it accepts a parameter name, and edits its value with the new value specified with **in** tag.
- **save**, (optional) it accepts a parameter name, the value of that parameter is saved in a variable, it is necessary to specify the name of the variable using **as** tag.
- **add**, (optional) it accepts a parameter name, it appends to the parameter value the string passed with **this** tag.
- **type** (optional) specify how the decoded parameter should be interpreted (txt or xml).

In a Message Operation, there is the possibility to specify a parameter or some text to be decoded before manipulation. To do that, specify with **decode parameter** the parameter to be decoded and with **encodings** the encodings necessary to decode the parameter. The order of definition of the **encodings** will be followed during decoding. The parameter (or text) decoded will be encoded again automatically at the end of the Message operation before forwarding it. The decoded parameter can be manipulated using the **type** tag. There is the possibility to interpret the decoded parameter by two means:

Type txt Associated with this interpretation, it is possible to use a list of actions over the plain text:

- txt remove: removes the matched string from the decoded parameter.
- txt edit: edits the matched string with a custom string (specified with the **value** tag).
- txt add: after the matched string adds a string specified with the **value** tag.
- txt save: saves the matched string in a variable with the name specified in the **as** tag.

All the previous tags accept a regex, and anything that regex matches will be edited, added, or saved based on the specified tag.

Type xml Another possibility is to interpret the decoded text as XML. To do this, the type tag has to be set to *xml*. This way the various possible operations to be done on the decoded xml are:

- **remove tag**, removes the specified tag.
- **remove attribute**, removes the specified attribute associated with the xml tag specified using the **xml tag**.
- **edit tag**, edits the specified tag with the value contained in **value** tag.

- **edit attribute**, edits the specified attribute associated with the xml tag specified using the **xml tag**.
- **add tag**, adds the specified tag, having the value specified with tag **value**, and also the name of the parent xml node to add the new node to, has to be specified using the **xml tag**.
- **add attribute**, adds the specified attribute associated with the xml tag specified using the **xml tag**.
- **save tag**, saves the specified tag value.
- **save attribute**, saves the specified attribute value associated with the xml tag specified using the **xml tag**.

In the PKCE example found in Section 3.2, a simple Message Operation is present, which has to remove the parameter `code_challenge` from the url of the message, so the resulting Message Operation will be:

```
{
    "from": "url",
    "remove parameter": "code_challenge"
}
```

Listing 3.4: Message Operation definition

Note for body section in message operations If the *body* section is chosen, the meaning of the following tags becomes different:

- **remove parameter** will work like **remove match word**, in a way that the value of the tag is treated as a regex which will be matching against the entire body section, having all the matches removed from it.
- **edit** is treated as a regex, substituting everything that matches that regex with the text specified by the **in** tag.
- **save** is treated as a regex, saving what will be matched by the regex, the name of the variable in which the value will be saved is specified with the **as** tag.
- **add** is associated with a regex. It will add at the end of the matched text the value specified by **this** tag.

This also changes for the **decode parameter** tag, in a way that if the message section is 'body', the **decode param** will accept a regex, and everything matched by that regex will be considered to be decoded.
An example of a regex to match a parameter in the body could be "`(?<=parameter_name=)[^$\n& ]*`" that will search for the text "parameter_name", taking everything after the "=" until end of line or "&" or whitespace is found.

This difference in the tag meaning is due to the difficulty of identifying parameters in the body section in contrast to the head section. While the head section is based on the HTTP standard, having all parameters defined in a clear and well-defined way like "Name: value" the body section could contain any type of content. To manage this variety of contents, the decision to use regex instead of parameter names for the body section has been chosen.

3.3.9 Message Type definition

The message type definition is needed to define some types of messages that will be later used in the language to intercept them. The message type definition is not part of the language, but it is stored in a file in the Burp folder. The definition of the type of messages uses the same Objects as the language. A Message Type Object is defined using these tags:

- **name**, the name that will be used in the language to refer to this message type.
- **is request**, if set to true if the searched message is a request, false otherwise.
- **response name**, if the searched message is a request message, this tag can be used to associate a name to the response of that message. This is useful when only the request message can be identified, allowing to intercept the correlated response message.
- **checks**, a list of Check objects used to identify the message. If evaluated to true, the message is considered found.

Following the example from Section 3.2 the definition of the *authorization request* message used in the test, and can be found in the Listing 3.5.

```
{
    "message_types": [
        {
            "name": "authorization request",
            "is request": true,
            "response name": "authorization response",
            "checks": [
                {
                    "in": "url",
                    "check param": "response_type",
                    "is present": "true"
                }
            ]
        },
    ]
}
```

Listing 3.5: Message Types definition

When used, this message type will search for the parameter `response_type` in the url of every request message. When a match is found, the Operation in which this Message Type is used will be executed.

3.4 The oracle

The set of all parts of the language that decide the result of the tests is called Oracle, which decides whether a test should be considered passed or failed (or not applicable). I decided to build the oracle so that the user can almost fully customize it. The oracle is based on three main components:

- Evaluation of the complete (or incomplete) execution of the session track.
- Evaluation of the Precondition objects.
- Evaluation of the Validate objects.

If all the above conditions are met, the test is considered passed. Otherwise, it is considered failed (or not applicable). The oracle can be built, for example, by using Validate objects to verify that some intercepted messages satisfy some conditions like having a particular parameter or string in them.

To build the oracle for the PKCE example mentioned in Section 3.2, both the result of the test and the precondition has been used, specifically, the precondition part is shown in Listing 3.6

```
"preconditions": [
    {
        "in": "url",
        "check param": "code_challenge",
        "is present": true
    }
],
```

Listing 3.6: Precondition definition

This precondition is used to consider the test "not applicable" if the parameter `code_challenge` is not found in the authorization request message. This means it is not possible to execute the given test over the actual intercepted message if the preconditions are not satisfied.

The **result** tag of the Test in the PKCE example in Section 3.2 is set to:

```
"result": "incorrect flow s1"
```

This means that the oracle will consider the test passed, if and only if the execution of the session track of the session named *s1* will be incorrect. The execution is considered incorrect when the execution of the session track fails. This can be, for instance, because an unexpected page is displayed or an element that should be clicked is not present on the page. In the case of the PKCE example in Section 3.2 the test is passed if the removal of the parameter `code_challenge` is not admitted by the AS; usually, if this is the case, an error page will be displayed.

3.5 Sessions

A session is a browser executing a session track, a session track is a list of user actions that the browser will simulate automatically during execution of the Tests. There is the possibility of defining and using more than one session so that, e.g., reply tests can be executed. Different sessions can have different session tracks.

As said in Section 3.3.3, a **from session** tag can be specified in the Operation, this will specify in which session to search the message. To define the session track the idea used in [11, 8], has been used, adding some options like **wait** and **clear cookies** functionalities. The syntax of the session track is based on the plain text export of Katalon Recorder [1]. An example of a session track that does the login at Unitn website is this:

```
open | https://www.google.com/ |
click | id=L2AGLb |
click | link=Accedi |
click | id=identifierId |
type | id=identifierId | matteo.bitussi@studenti.unitn.it
click | id=identifierNext |
click | id=clid |
type | id=clid | matteo.bitussi@unitn.it
click | id=inputPassword |
type | id=inputPassword | password
click | id=btnAccedi |
click | link=Gmail |
```

Listing 3.7: Session track Unitn login

This session track will do the login on the Unitn website using some credentials and password. The supported actions are:

- **open** | **url** |, to open an url.
- **click** | **id=, link=, xpath=** |, to click on a http object with the given id, link or xpath.
- **type** | **id=** | **text**, to write on a given http element the given text.
- **wait** | **milliseconds**, to make the execution of the session wait for a given time.
- **clear cookies** |, to make the browser of the session clear all the cookies in it.

4 Implementation of the tool

This chapter will discuss the implementation of the language and the tool, the problems faced, and the adopted solutions.

4.1 General overview of the tool

The components of the final tool can be seen in Figure 4.1. Burp Suite is composed of its proxy and related APIs; the tool will get all the messages from the proxy through the API, it will process them, and return them to the proxy. This way, all the messages will pass through the tool and make it possible to check and edit them. Every browser, one for each session, will be using a dedicated proxy, which will act like a Man-In-The-Middle attack from the browser to the server, establishing a secure connection only on the last part of the communication to the server, making it possible to see plain HTTP communications on the browser side. Each browser will be supplied with the user actions taken from the session tracks specified beforehand. It is also possible to do manual user actions on the browser in case (for example) a captcha has to be resolved. Also, the session actions taken from the tests defined by the language can be supplied to the browser (for example, to pause or stop it). An essential component of the tool is the Test Suite defined with the language, which is supplied to the plugin, and executed with the sessions, eventually giving a result.

4.2 The tool

To implement the tool, I have decided to start from a work done by my colleague Wendy Barreto in her bachelor thesis [2], which realized a similar tool for OIDC and OAuth SSO protocols. This was a good base to start with the implementation. The interface of [2] has been taken and adapted to fit the needs of this work. The tool code is written in Java, and the Burp's interface classes have been used to interact with it. The standard usage of Burp is based on the execution of a browser that connects to the Burp's proxy, in a way that all the packets can be intercepted, viewed or edited and forwarded or dropped from the Burp interface. The tester would do some actions on the browser and watch the message flow in Burp and then check them or edit them. With the tool, the idea is the same, but the operation is done on the browser, and the checks or edits on the messages are made automatically, in a way that the tester does not have to do them by itself.

4.2.1 User Interface of the tool

In Figure 4.2 the interface of the tool is shown. Starting from the top left, there is the session track input space, where a different track can be specified for each session. Following on the top right, a series of buttons that allow various configurations can be found:

- **Use Chrome** or **Use Firefox** the browser to be used can be selected.
- **Select driver** the driver used to automate the actions on the browser can be selected.
- **Record** the record button can be used to record the flowing messages.
- **Load messages** the load messages button can be used to load the previously saved messages to be tested offline.
- **Offline mode** the offline mode button to test the loaded messages instead of the live ones.
- **Execute track** used to execute the session track without executing the tests, useful when the unaltered messages have to be saved.
- **Test track** used to test the session track without saving or doing any test.

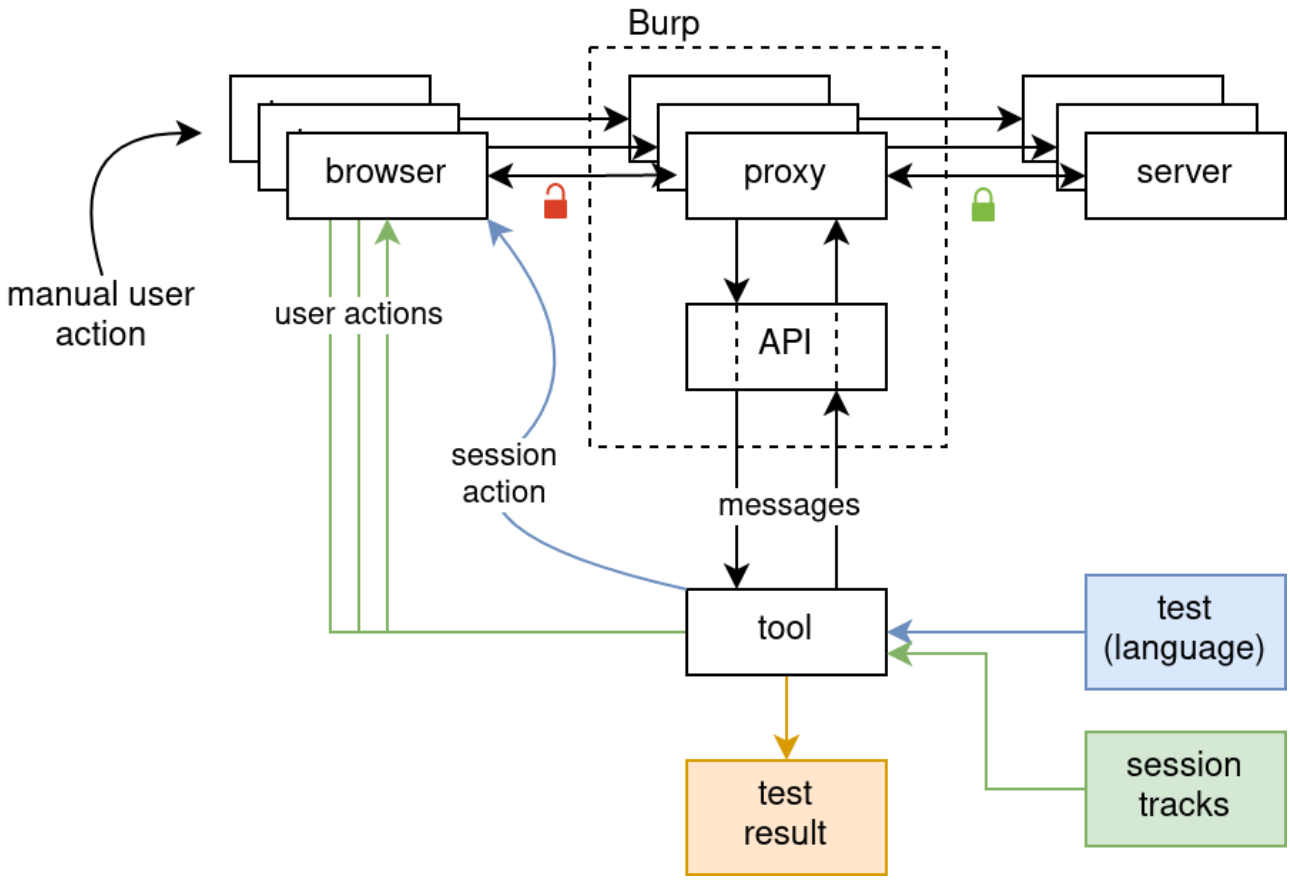

Figure 4.1: General schema

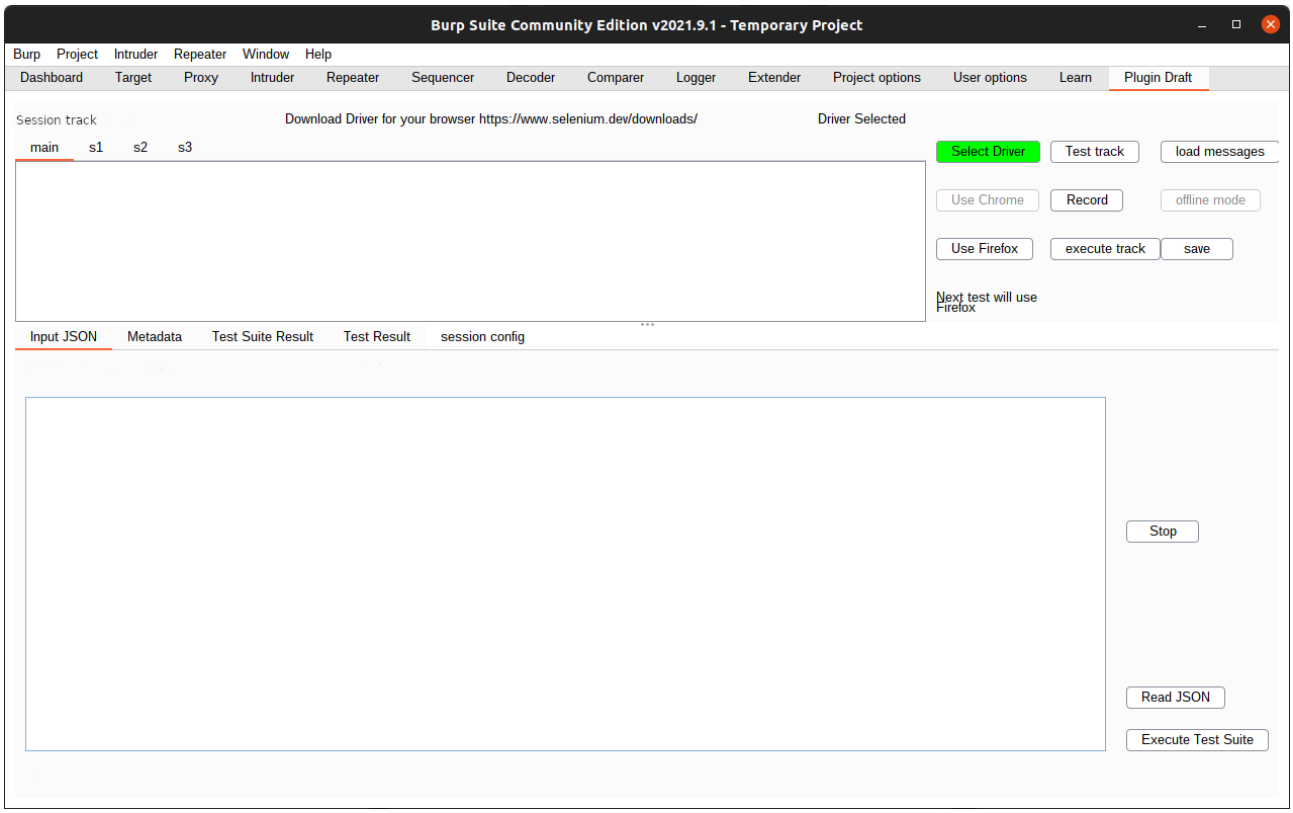

Figure 4.2: Tool interface

In the bottom part, multiple tabs can be found:

- **Input JSON** tab is used to load the tests written in the language into the tool, and with the use of two buttons, the language can be parsed, and the tests can be executed.
- **Test suite result** is the tab containing all the results of the executed tests.
- **Test result** is the tab used to see the specific test result, with all the intercepted messages related to it.
- **Session config** is used to configure the ports of the sessions that will be used in the tests.

In the bottom right part, when the **Input JSON** tab is selected, three buttons are available:

- **Stop** used to stop the current execution.
- **Read JSON** used to read the written Tests.
- **Execute Test Suite** used to execute the written tests.

4.2.2 Session managing

The sessions are managed independently, and each session is basically a browser that is launched when a session is started. Each session can follow a different session track defined in the apposite tabs. Every session is run in a separated thread to make parallelism between every session possible. Using specific commands in the language, it is possible to do some actions on each session, like stop it, pause it, or clear its cookies. Each browser uses a different proxy port so that it is possible to know from which session the messages come from, and so, being able to specific sessions in the tests.

4.2.3 Test execution

The test execution differs from passive to active, as passive tests do not need the edit of the messages, the execution of the session track is done once the messages are saved, and the tests are executed on the saved messages. The possibility of exporting the saved messages to a file has also been added so that they can be imported into the tool and tested again. On the other hand, active tests need to edit the messages, so the execution of the track has to be repeated for each test.

4.2.4 Decoding & encoding of parameters

As said in Chapter 3, the decoding and encoding of parameters are possible. To do that, a list of encodings to be done on the parameter has to be provided, e.g., url, base64, deflate. Once the specified message is intercepted, the parameter is taken and decoded following the order of the provided encodings list. To do that, part of the code of SAML Raider [15] has been used. SAML Raider is a Burp's plugin used to manage SAML certificates. The part of the code that deals with encoding and decoding the parameters has been taken and edited to fit the tool.

4.2.5 SAML certificate managing

In SAML Requests and responses, there is sometimes the need to remove or edit the certificate associated with that request or response, so, to speed up the process, a specific tag in the language has been added to remove or edit the certificate signature. There is still the possibility of doing the same removal by editing the SAML request or response with a regex, but with the use of the tag, this becomes more convenient. To do this, a part of the code of SAML Raider [15] has been used and edited to fit the needs of the tool.

4.3 Problems and limitations encountered

During the implementation and the testing of the tool, multiple problems have been encountered, the majority of them have been solved, but some are still present. The most relevant ones will be discussed next:

4.3.1 Automation problems

One of the tool's limitations is the session track actions automation. Some captchas are often encountered during execution, making it impossible to proceed. Moreover, the track execution is limited, there is only a possible flow of actions (the one defined), and there is not the possibility of inserting "if then else" constructs that could help to differentiate the actions based on the actual page or popup. For example, it could happen that a "limited time offer" popup could appear in a website only in a particular time, the execution of the session track could be compromised by that, making impossible to distinguish whether the test failed because of the tested vulnerability or the actual popup. Another problem in the automation part of the tool is that it is sometimes limited because the session track has to be defined over a specific website, doing a set of actions that are directly correlated to the website. Whenever the website is changed somehow, for example, the IDs or the button's position to be clicked change, the execution will fail because the track could not continue. This is still not resolved, as no methods to make the track more dynamic have been found yet. This also means that every different web service that has to be tested will need a different session track to be defined, making it time-consuming to define.

These Automation problems mainly occur when the service to be tested is not under the tester's control. Otherwise, captcha and popups could be easily deactivated for testing purposes. This means that this problem happens only in specific use case scenarios, so it is not a big deal.

4.3.2 Oracle is sometimes ambiguous

There is still a problem with the Oracle, where sometimes false positives or negatives arise if the tool's execution is interrupted for any reason. This is a problem because the interruption of the execution is a term of valuation for the Oracle. This means that the oracle will give a result also based on the correctness or incorrectness of the execution of the session track. This makes it impossible to distinguish if the session track has failed because of an error on the definition on it or because of an expected reason (like after a message modification).

4.3.3 Interface and user feedbacks

The tool's interface is a bit raw, and it is not very user-friendly. The user experience could be improved. I did not focus my attention on these topics during my work, but they are essential as the tool is not so easy to use. Also, the feedbacks of the errors encountered by the tool, such as execution errors or others, are not all shown to the user. This indeed has to be fixed, making more apparent to the end-user what is going wrong.

5 Case Studies and Language Validation

Once the language and the approach to execute the tests have been defined and implemented, the tool has been applied to two different scenarios of IdM protocols, OAuth and SAML. In the end, two test suite has been defined, one for each protocol, that are ready to be used. This chapter will describe how this was done.

5.1 OAuth & OIDC Use-Case

To test the tool and the language during development, there was the need to specify a test suite to evaluate the correctness of their results. This was done starting from two works [11, 2] that aimed at defining a set of tests to be done to test OAuth and OIDC protocols in a way that well-known vulnerabilities would be avoided. These two works have been chosen because the tests they present result from a deep examination. The OAuth and OIDC tests defined and used in [11, 2] has then be re-defined in the language. Due to the newly available functionalities introduced with the language, new tests have been specified, especially active ones. In total, 7 active tests and 11 passive tests built the OAuth/OIDC test suite. These tests aim at testing the services against the well-known vulnerabilities to find out if the services are vulnerable. One of them can be found in Section 3.2. The complete list of tests with a brief description can be found in Table 5.1.

Test	Type	Description
CSRF protection (remove state)	Active	Tries to remove state parameter
CSRF protection (edit state)	Active	Tries to edit state parameter
PKCE Downgrade	Active	Tries to remove code_challenge parameter
PKCE plain supported	Active	Tries to remove code_challenge_method parameter
Redirect uri subdomain	Active	Tries to edit the redirect uri with subdomain
Redirect uri external domain	Active	Tries to edit the redirect uri with external domain
State parameter replay	Active	Replay the parameter state from a session to another
Parameter state is used	Passive	Check for the presence of the parameter state in
Compliance to Standard	Passive	Checks the presence of client_id and response_type
PKCE is implemented	Passive	Check for common params presence
PKCE method is not plain	Passive	Check if param code_challenge_method=plain
Prevent clickjacking	Passive	Check the headers values
Open redirectors: HTTP 307	Passive	Check for 307 redirect
Block for Referrer header is used	Passive	Checks for common parameters presence
Token saved as cookie in plain	Passive	Check for the header cookie and search for token in it
Client_id is present	Passive	Check for client_id in all requests url
Implicit grant not used	Passive	Check for response_type parameter
Password credential grant not used	Passive	Check for response_type parameter

Table 5.1: OAuth/OIDC test suite

5.2 SAML Use-Case

During the last stages of the development and testing of the tool, a strict collaboration between my colleague Sofia Zanrosso and me has started. Her objective was to create a SAML Test Suite to facilitate automatic penetration testing over SAML [17]. The tool defined in this thesis was compared

with other ones and was used to define and execute the tests. During the progress of both our works, a lot of feedback and bugs have been reported to me, speeding up the testing phase of the tool. At the same time, we found that my tool was in some respects better compared to the other alternatives. For example:

- Other plugins were giving false positives on some tests.
- "the previously employed transition times between tools have been greatly reduced".
- "making it possible to analyze almost completely the vulnerabilities of the tested subjects".

In the end, it has been possible to define all the necessary tests and to execute them to test the services.

5.3 Expirience during the use of the language

Experience during the use of the language. Many tests have been translated or defined into the language during the two preceding case studies. This has not been difficult to do. In total, 18 tests for OAuth and 86 tests for SAML have been defined. The process of defining or translating a test involves the identification of the messages to be intercepted, the actions to be done on them, how many sessions are needed, and the parts of the messages to be checked to give a result of the test. Once these four main components have been identified, the next step is to declare everything from the Message Type to the checks. The most difficult part is probably the definition of a Message Type to intercept a message, as it is sometimes difficult to filter the messages.

6 Related work

This chapter will describe software and tools related to the work in this thesis.

6.1 Domain Specific Language

The language introduced in this thesis used to specify tests can be viewed as a Domain-Specific Language, which is a computer language specialized to a particular application domain [16]. More specifically, it falls under the class of Embedded Domain-Specific Modelling Languages. Embedded (or Internal) Domain-Specific Languages are typically implemented within a host language as a library and tend to be limited to the syntax of the host language, though this depends on host language capabilities [16]. The language in this work is hosted by JSON, having all the tests encoded in it. The encoded JSON is then parsed, and the fields are interpreted as Burp's API calls.

6.2 Micro-Id-Gym

Micro ID Gym (MIG) "aims to assist system administrators and testers in the deployment and pentesting of IdM protocol instances" - [3], inside this tool, two pentesting tools can be found:

- MIG - OAuth/OIDC [11].
- MIG - SAML SSO [8].

They are both plugins for Burp, which have as objective to test the two different protocols. These plugins execute a series of actions on a browser, check the messages in the background, and then provide a result. These two plugins are related to mine, but they are specifically created and defined to test SSO protocols only, and the tests they used are fixed and cannot be easily edited. If a new test has to be implemented, the plugin must be recompiled.

6.3 SSO Testing language and Plugin

The preceding two tools of MIG found in Section 6.2 have been improved by the work done by my colleague Wendy Barreto [2] in her bachelor thesis at Università di Trento to test OAuth and OIDC SSO protocols with a custom test definition pattern. Her work aimed to fix the problem of hard-coded tests in the plugins for SSO protocols testing. The previous MIG plugin had been improved by removing the staticity of the test, adding the possibility to define all the tests with the use of a JSON language. The available test actions worked well, but there were some limitations on the possible actions, especially in active tests. For example:

- Limited oracle for the verification of active tests, having just the verification of the correct execution of the operation and a check for the string "error" on the last page of the browser.
- The filtering of the message to check or edit for static tests is limited, only *Authorization grant message*, *Response messages*, *Request messages* and *All messages* are available.
- Only regex are supported to search what is needed in a message.
- Impossibility to work over encoded parameters.
- Impossibility of doing multiple operations on a single message.
- Impossibility of saving a parameter and using it somewhere else.
- Impossibility of using multiple sessions in a test.

Some of these limitations were stated as future works in [2]. A complete list of differences between the two languages and tools can be found in Attachment B.

Previously in this thesis, it has been said that part of this work has been taken as a base to start with the implementation of the tool. The idea of a language that could be used for any test over browser-based protocols was born when I used her plugin, which was limited to OAuth and OIDC tests. I wanted to enlarge the possible tests to be defined without a restriction on a specific protocol. One of the things that have been used is the interface of the plugin, which has been modified, adding buttons and tabs to deal with multiple session tracks and other added functionalities. Also, the automation of the session track was taken and edited. This part was already used in [11, 8].

7 Conclusions and Future Work

This work aimed to design and implement an automated tool that could execute security tests specified with a language, allowing the tester to use an already defined test suite, saving time, and ensuring that all the tests for a specific protocol are done. Firstly, a formal declarative specification for the pentesting strategies has been designed, searching for all the possible actions and checks that the tester could need during pentesting. Once this has been done, the implementation of the declarative specification has started, forming a language to declare all the tests. The next step was to design and implement a tool that could be used to execute the tests. The tool was built as a plugin for Burp. It was able to communicate with its proxy to manage sessions and manipulate messages. The definition of an OAuth/OIDC test suite written in the language has started during this phase. This was done to test the tool during development and check its results' correctness. In the end, this test suite was a complete set of tests to test the OAuth/OIDC protocol.

Lastly, the tool and language are working, as can be seen in [17]. In this case, all the necessary tests have been specified in the language and executed, resulting in the expected result. The tool admitted to specify and execute tests that were not possible with other software. Also, the tool worked during the OAuth/OIDC testing phase, giving the same result as other compared software in the same tests executed. These two use cases give testers a good set of tests to test OAuth/OIDC and SAML protocols in an automated manner. The initial objective of having an automated tool to help the testers to identify vulnerabilities in services implementations can be considered reached.

The software has still to be tested more deeply. It was used just by me, and in [17], this means that there could be some stability problems and some bugs not yet discovered. This could be fixed in future works, with the problems and the limitations introduced in Section 4.3. Some enhancements that could be added to the work could be the extension of the message filtering part of the language with the use of machine learning, making it possible to define a more abstract filter that does not solely rely on the search of parameters or exact strings. The message saving part of the software could be extended having different file formats, such as ".cap", this way the intercepted messages could then be imported into other software. Another thing is that the test definition language could be extended even for other uses, like for low-level networking protocols such as routing protocols, electronic trading protocols, IP protocols, and more; Allowing to search for known vulnerabilities in real-time or just by analyzing saved packets files. For example, a plugin for Wireshark software could be developed.

Bibliography

[1] Katalon recorder sample plugin. https://github.com/katalon-studio/katalon-recorder-sample-plugin. Last accessed 19/11/2021.

[2] Wendy Barreto. Design and implementation of an attack pattern language for the automated pentesting of OAuth/OIDC deployments, 2019/2020. Bachelor thesis, DISI, University of Trento.

[3] Andrea Bisegna, Roberto Carbone, Ivan Martini, Valentina Odorizzi, Giulio Pellizzari, and Silvio Ranise. Micro-id-gym: identity management workouts with container-based microservices. *Int. J. Inf. Secur. Cybercrime*, 8(1):45–50, 2019.

[4] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017.

[5] Brian Campbell, Chuck Mortimore, and M Jones. Security Assertion Markup Language (SAML) 2.0 profile for OAuth 2.0 client authentication and authorization grants. *Internet Engineering Task Force (IETF)*, 2015. Page 2, Last accessed 29/11/2021.

[6] Yulia Cherdantseva and Jeremy Hilton. A reference model of information assurance & security. In *2013 International Conference on Availability, Reliability and Security*, page 547. IEEE, 2013.

[7] Milena Gabanelli e Simona Ravizza. Attacchi hacker, dati sanitari in pericolo: la lista segreta dei 35 ospedali colpiti. https://www.corriere.it/dataroom-milena-gabanelli/attacchi-hacker-dati-sanitari-pericolo-lista-segreta-35-ospedali-colpiti/1abf2704-2079-11ec-924f-1ddd15bf71fa-va.shtml, 2021. Last accessed 29/11/2021.

[8] Stefano Facchini. Design and implementation of an automated tool for checking SAML SSO vulnerabilities and SPID compliance, 2019/2020. Bachelor thesis, DISI, University of Trento.

[9] Daniel Fett. Downgrade attack on PKCE. https://danielfett.de/2020/05/16/pkce-vs-nonce-equivalent-or-not/. Last visited 16/02/2022.

[10] David Gourley, Brian Totty, Marjorie Sayer, Anshu Aggarwal, and Sailu Reddy. *HTTP: the definitive guide*. O'Reilly Media, Inc., 2002.

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

[12] Dick Hardt et al. The OAuth 2.0 authorization framework. https://datatracker.ietf.org/doc/html/rfc6749.html, 2012. Page 1, Last accessed 29/11/2021.

[13] Il Fatto Quotidiano. Attacco hacker lazio, al poliambulatorio santa caterina cittadini rimandati indietro: "impossibile fissare visite o cambiare medico". https://www.ilfattoquotidiano.it/2021/08/03/attacco-hacker-lazio-al-poliambulatorio-santa-caterina-cittadini-rimandati-indietro-impossibile-fissare-visite-o-cambiare-medico/6282342/, 2021. Last accessed 29/11/2021.

[14] Nat Sakimura, John Bradley, and Naveen Agarwal. Proof Key for Code Exchange by OAuth Public Clients. RFC 7636, September 2015.

[15] Compass Security. SAML raider plugin. https://github.com/CompassSecurity/SAMLRaider. Last accessed 23/11/2021.

[16] Wikipedia contributors. Domain-specific language — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Domain-specific_language&oldid=1072061142, 2022. [Online; accessed 4-March-2022].

[17] Sofia Zanrosso. Enlarging the Pen Test Coverage of SAML Single Sign-On Solutions with Cyber Threat Intelligence, 2020/2021. Bachelor thesis, DISI, University of Trento.

Attachment A PKCE complete test Example

```
{
    "test suite": {
        "name": "OAuth Active tests",
        "description": "A series of tests to test OAuth's well-known
    ↪ vulnerabilities",
        "filter messages": true
    },
    "tests": [
        {
            "test": {
                "name": "PKCE Downgrade",
                "description": "Tries to remove code_challenge parameter",
                "type": "active",
                "sessions": [
                    "s1"
                ],
                "operations": [
                    {
                        "session": "s1",
                        "action": "start"
                    },
                    {
                        "action": "intercept",
                        "from session": "s1",
                        "then": "forward",
                        "message type": "authorization request",
                        "preconditions": [
                            {
                                "in": "url",
                                "check param": "code_challenge",
                                "is present": true
                            }
                        ],
                        "message operations": [
                            {
                                "from": "url",
                                "remove parameter": "code_challenge"
                            }
                        ]
                    }
                ],
                "result": "incorrect flow s1"
            }
        }
    ]
}
```

Attachment B Language comparison

Action	Old language	New language
Custom message filtering	only on active tests	supported
Edit string	only by regex	supported with regex and check
Remove string	only by regex	supported with regex and check
Add string	not supported	supported
Check parameter	only with regex	with regex and check construct
Multiple operations in single message	not supported	supported
Saving and reusing of values and messages	not supported	supported
Multiple sessions in single test	not supported	supported
Custom oracle definition	not supported	by using regex and checks