Dipartimento di Ingegneria e Scienza dell'Informazione
Departement of Information Engineering ans Computer Science

Bachelor's Degree in
Computer Science

Final dissertation

HTTP STL - Security testing language and implementation

Sottotitolo (alcune volte lungo - opzionale)

Supervisor
Silvio Ranise

Student
Matteo Bitussi

Co-Supervisors
Andrea Bisegna
Roberto Carbone

Academic year 2021/2022

Ringraziamenti

...thanks to... TODO (in italiano)

Contents

Abstract **3**

1 Introduction **4**

2 Background **4**
2.1 Burp Suite . . . 4
2.2 JSON . . . 5
2.3 Regex . . . 5
2.4 HTTP protocol . . . 5
2.5 IdM protocols: SAML and OAuth . . . 5
2.5.1 OAuth . . . 5
2.5.2 SAML . . . 5

3 Design of the testing language **5**
3.1 Test example: PKCE Downgrade . . . 5
3.2 Language structure . . . 6
3.2.1 Test suite . . . 6
3.2.2 Test . . . 6
3.2.3 Operation . . . 8
3.2.4 Message Operation . . . 9
3.2.5 Message type definition . . . 10
3.3 The oracle . . . 11
3.4 Sessions . . . 11

4 Implementation **12**
4.1 The plugin . . . 12
4.1.1 The interface . . . 12
4.1.2 Test execution . . . 13
4.1.3 Decoding & encoding of parameters . . . 13
4.1.4 SAML certificate managing . . . 13
4.1.5 Oracle . . . 14
4.1.6 Session managing . . . 14

5 Uses cases **14**
5.1 SAML Use-Case . . . 14
5.2 OAuth & OIDC Use-Case . . . 14

6 Related work **14**
6.1 SSO Testing language and Plugin . . . 14
6.2 Micro ID Gym . . . 14
6.3 Last plugin version . . . 15

7 Conclusions **15**

Bibliografia **15**

A Language compartison 1 **17**

B PKCE test Example complete **18**

Abstract

This thesis covers the work started in my internship at FBK in the context of Single-Sign-On (SSO) protocols testing. SSO protocols are becoming more and more poular these days, and are being used in very sensitive applications such as SPID or CIE. Seeing the vast number of different implementations that are being used from whatever type of service, there is the need to test them in order to ensure that (at least) the known most common vulnerabilities are avoided.
To avoid having to manually test each implementation, an automaic tool is necessary. Moreover, a standard language to define these test suites has to be defined. This is what will be shown in this paper.

1 Introduction

In the last years, we are seeing 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 great step forward, the storage is optimized and more easily accessible, facilitating the cataloging of the document. But there are also some big concerns about security of that data, as it is virtual, the access to it is regulated by some authentication protocols instead of a physical identification of the subject accessing the physical documents. This means that we have to be sure that the person accessing the data is the one that is claiming to be. The three basic principles over which data security is based are confidentiality, integrity and availability of data, this is called the CIA triad [4]. If those requirements are not met, sensitive data could be inaccessible or even at risk of access from unauthorized parties, as a result, critical services for health care could be inaccessible, this is what happend for example in Lazio [3] [6], Italy on 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, a lot of data in the systems has been encrypted, and so, denied of any access. It has to be said that preventing RansomWares infections is not the scope of this thesis, but this is a good example of what the consequences of exploiting a vulnerability could be. To ensure the this requirements are satisfied, there is a need to test all the implementations of the protocols and system that are liable of the security of the data. 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 would have to be edited. This is a lot of work to be done by a security tester, and the problem is that not always the person testing the implementation is qualified to do it. The result is that some vulnerabilities could be remain undiscovered, representing a weakness in the system.

An automated tool to test implementations of security protocols using a well-known list of vulnerabilities to be tested is needed, this is what the work described in this thesis is all about. Some tools to test specific protocols and vulnerabilities already exist [5][8], but they are usually very specific and hardcoded in a way that the tests could not be easily edited. With this work, I wanted to create a tool that could be used to test any type of protocol that ran over HTTP, and that could be define the tests in a very abstract way, allowing edit and addings. In order to do this, a new language will be used to define the tests, and a software will be implemented to executhe those tests.

In this thesis the design and implementation of this new tool will be discussed.

2 Background

In this chapter the subjects needed to comprehend the rest of the thesis will be discussed. The most common subjects will be ignored, giving a focus on the more specific and less common ones.

2.1 Burp Suite

Burp is one of the most used application security testing software for web security testing. It works by the use of a proxy server over which a browser redirect the traffic to. The proxy does like a Man In The Middle attack, taking the input traffick from the browser and replying the messages to the target service, giving also trasparency over the TSL or SSL 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 to the developers access to the java API. This is exactly how the tool will be implemented, using burp as a base over which develop the software that will be executing the tests. The tool (from now on plugin) will be able to intercept, read and edit messages that pass trough the burp's proxy by the use of Burp's API.

2.2 JSON

2.3 Regex

2.4 HTTP protocol

2.5 IdM protocols: SAML and OAuth

The so-called IdM protocols are protocols that deals with identity management. For the scope of this thesis, SAML and OAuth will be discussed, as they are used in examples and in the related works.

Both SAML and OAuth are Single Sign-On (SSO) protocols, that "is 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" - [12]. The difference between SAML (Security Assertion Markup Language) and OAuth is how they work, but their objective is the same, to certificate the identity of a given person.

2.5.1 OAuth

As stated in [9], The OAuth 2.0 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 own behalf. A series of messages has to be exchanged between the two parties in order to authenticate a resource owner that wants to access some reserved data in a service.

2.5.2 SAML

As stated in [10], The Security Assertion Markup Language (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

In this chapter it will be introduced how the Plugin and the testing language had been designed. The idea was to think of a language that could implement all the possible actions which a security tester would be wanting to do on the messages. One of the objective was to think of a language that could define tests that could then be tested over multiple webservices. For example, a series of tests to verify the well-known vulnerabilities of a specific protocol could be defined and then used on any type of website. I had to decide how to write and define the tests, I thought I could define a proper language with a dedicated parser, but it was not worth the effort, as there were already some well-tested alternatives available. one of them is JSON, which has been used as a base over which write the tests. JSON is a convinient way of defining gerarchical sturctures like tests are. The idea behind this language is that a specific message can be intercepted and checked or edited in some way. The gerarchical structure and the details of the language will be discussed in this chapter.

3.1 Test example: PKCE Downgrade

I want to introduce the language with an example. Due to its complexity, having a real example before the explanation of all its components could be helpful to understand their use. The PKCE Downgrade test has as objective to test an OAuth vulnerability where removing the parameter "code_challenge"

from the url of an authorization request message will be downgrading the authentication proces in a way that PKCE will not be used if the service is vulnerable. To test this, we have to intercept the authorization request message, remove the parameter "code_challenge" from it, and then forward it.

3.2 Language structure

Each object of the language is a JSON object which can contain different possible tags based on its type.

3.2.1 Test suite

The test suite is the main Object which contains all the other one, 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

Reffering to the test example above, the definition of the test suite for it is:

```
{
    "test suite": {
        "name": "OAuth active tests",
        "description": "A test suite containing a OAuth test"
    },
    "tests": [
        {
            // A test
        }
    ]
}
```

As we can see, 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.

3.2.2 Test

Following the gerarchical order, the Test object is the one that actually defines a test. As said earlier, a test is contained in a Test Suite, and has 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 doesn't need to manipulate the flow or the content of the messages is considered passive, otherwise it is considered active. The list of Operation Objects contained in a Test is executed iteratively one after the other. Only after the actual Operation has finished the execution the next is started.

Following the PKCE example above, we define an active Test:

```
{
    "test": {
        "name": "PKCE Downgrade",
        "description": "Tries to remove code_challenge parameter",
```

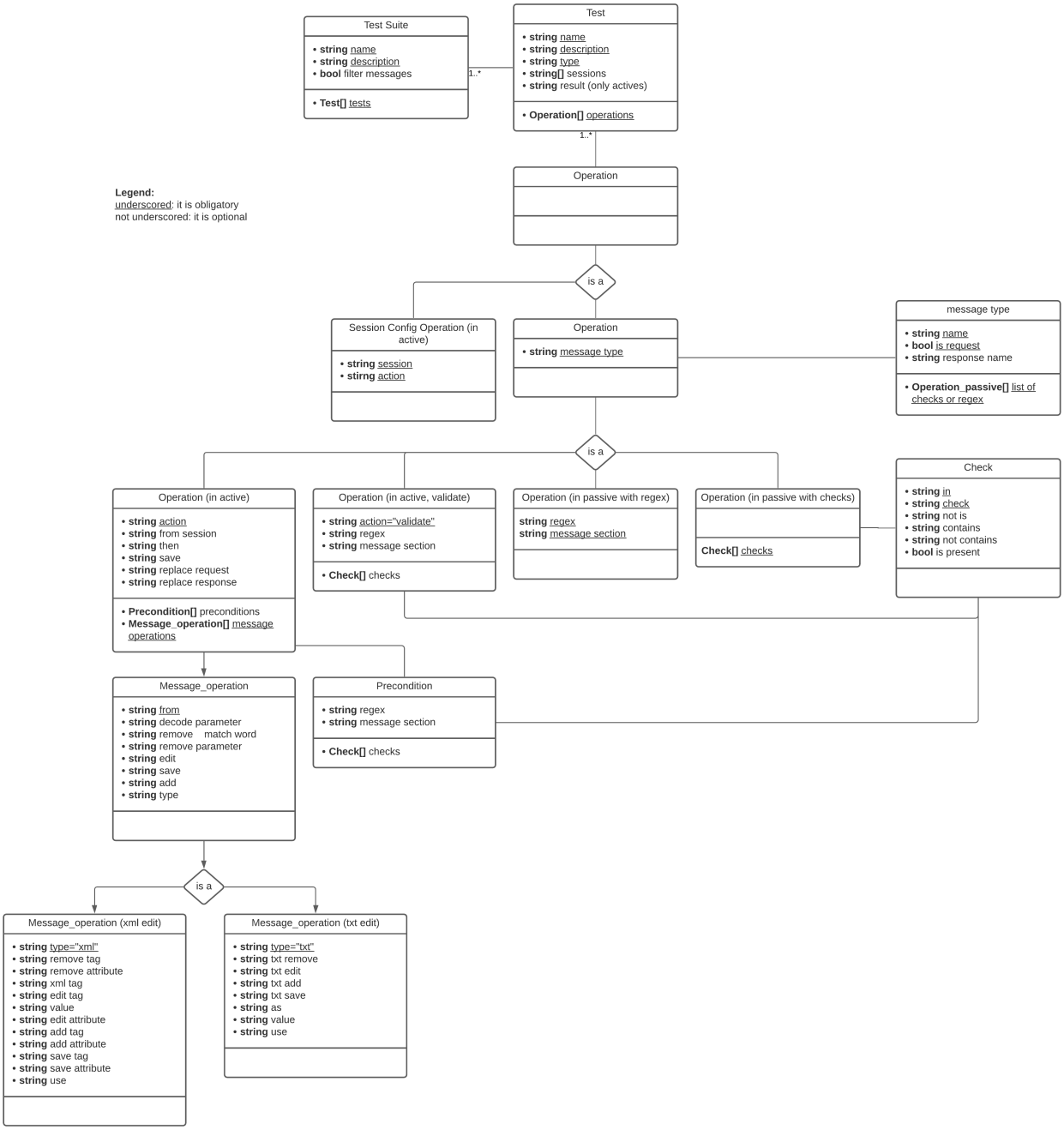

Figure 3.1: language structure

```
        "type": "active",
        "sessions": [
            "s1"
        ],
        "operations": [
            // list of Operation Objects
        ],
        "result": "incorrect flow s1"
    }
}
```

We define the test, specifying its type, which is active, as it has to edit a message removing a parameter from the url, we define a session to be used (more on that later in this chapter) and we add a list of Operation to be executed. The result of the test is also specified, this will also be discussed later in the chapter.

3.2.3 Operation

The Operation object is the component that defines what a test actually does. As shown in the language structure image 3.1, 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 which an Operation is defined into, 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 specified also the message section over which to execute it (boy, head, url)

If the Test where the operations are defined is an **active** test, also if the intercepted messages need to be manipulated in some way, 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 to 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 becomes like a passive operation, because its objective is just to verify that some messages are as wanted. It will contain or a regex or a list of checks to be done. The Validate Operation is part of the Oracle, that is the component that decides whether the tests should be considered passed or not, more details on Oracle section.

In the PKCE example above, we define an Operation Object for an active test. 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. At the end the message is forwarded. More details on preconditions and Message operations in the next sections.

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

3.2.4 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 or string to be decoded before 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), remove text from te specified section in the matched message, it uses a regex
- edit, edit the matched text
- save, (optional) used to save an entire message in a variable in a way it can be used in future operations
- add, (optional) add some text after the matched text
- type (optional) specify the type of edit you want to do over a decoded parameter

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 parameter (or text) decoded, at the end of the Message operation will be encoded again automatically. The decoded parameter can be manipulated by means of the "**type**" tag, there is the possibility to intepreter the decoded parameter as plain text, and to edit it using some actions:

- txt remove
- txt edit
- txt add
- txt save

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

Another possibility is to interpret the decoded text as xml, assigning the type tag "xml". This way we have various possible operations to be done on the xml:

- remove tag
- remove attribute

- edit tag
- edit attribute
- add tag
- add attribute
- save tag
- save attribute

In the PKCE example above, we have just a simple Message Operation which has to remove the parameter "code_challenge" frome the url of the message, so the resulting Message Operation will be:

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

3.2.5 Message type definition

The message type definition is needed in order to define some types of message that will be later used in the language to intercept them. The message type definition is not actually part of the language, but it is stored in a file in the burp folder. Anyway, 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, the name that will be used in the language to refer to the response of the searched message
- checks, a list of Check objects used to identify the message. If evaluated to true, the message is considered found

This is an example that defines the SAML request and the SAML response messages

```
{
    "message_types": [
        {
            "name": "saml request",
            "is request": true,
            "checks": [
                {
                    "in": "url",
                    "check param": "SAMLRequest",
                    "is present": true
                }
            ]
        },
        {
            "name": "saml response",
            "is request": true,
            "checks": [
                {
                    "in": "body",
                    "check param": "SAMLResponse",
                    "is present": true
                }
            ]
        }
    ]
}
```

The SAML request message has in its url the parameter "SAMLRequest", the SAML response message instead, has the "SAMLResponse" parameter in the body. Then, if a saml request has to be intercepted in a test, it has to be defined in the message types, and then used in the Operation. As a result, if "saml request" is used in an Operation, the message having the parameter "SAMLRequest" in his url will be intercepted and processed by the Operation.

3.3 The oracle

The ensemble of all parts of the language that decide the result of the tests is called Oracle, the oracle decides whether a test should be considered passed or failed (or not applicable). I decided to build the oracle in a way that can be almost fully customized by the user. 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. 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.

From the PKCE example above, both the result of the test and the precondition as been used:

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

With this precondition, the test has to be considered "not applicable" if the parameter code_challenge is not found in the authorization request message. This means that 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 is set to:

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

this means that the oracle will evaluate the test as passed if and only if the execution of the session track of the session s1 will be incorrect, this means that if the browser will encounter some type of page which was not meant to encounter, the test will be considered passed. With "was not meant to encounter" I mean that the actions in the session track cannot be done, because the objects that should be pressed in the page are not present, this happens for example if an error page is displayed, which tells us the service is not vulnerable to the Test, so the test is passed.

3.4 Sessions

A session is a browser executing a session track, which is a list of user actions that the browser will execute automatically during execution of the Tests. There is the possibility of defining and using more than one session, in a way that (i.e.) reply tests can be executed. As said in the previous sections, a "from session" tag can be specified in the Operation, this will tell in which of the available session search the desired message. To define the session track I have used and extended the idea used in [11][8][7], 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[2]. An example of a session track that does the login at unitn.it:

```
    open | https://www.google.com/ |
    click | id=L2AGLb |
    click | link=Accedi |
    click | id=identifierId |
```

```
5     type | id=identifierId | matteo.bitussi@studenti.unitn.it
6     click | id=identifierNext |
7     click | id=clid |
8     type | id=clid | matteo.bitussi@unitn.it
9     click | id=inputPassword |
10    type | id=inputPassword | password
11    click | id=btnAccedi |
12    click | link=Gmail |
```

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 of the cookies in it

4 Implementation

In this chapter I will describe the implementation of the language and the plugin, the problems faced and the solutions adopted.

4.1 The plugin

For the implementation of the Burp's plugin introduced earlier, I have decided to start from a work done by my colleague Wendy Barreto [5] in her bachelor thesis, which realized a similar plugin for OIDC and OAuth SSO protocols, this was a good base to start with my implementation. The interface of [5] has been taken and adapted to fit the needs of this work. The plugin code is written in Java, I used the Burp's interface classes to interact with it. The standard usage of Burp Suite is based on the execution of a browser which 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 flowing packets in Burp and then check them or edit them. With the plugin the idea is the same, but the operation done on the browser and the checks or edits on the messages are made automatically, in a way that the tester doesn't have to do them by itself.

4.1.1 The interface

In figure 4.1 we can see the interface of the plugin, starting from top left we have the session track input space, where it can be specified a different track for each session. Following on the top right, wee have a series of buttons that allow various configurations:

- the browser to be used can be selected
- the driver to automate the actions on the browser can be selected
- the record button can be used to record the passing messages
- the load messages button can be used to load the previously saved messages to be tested "offline"
- the offline mode button to test the loaded messages instead of the live ones

In the bottom we have multiple tabs:

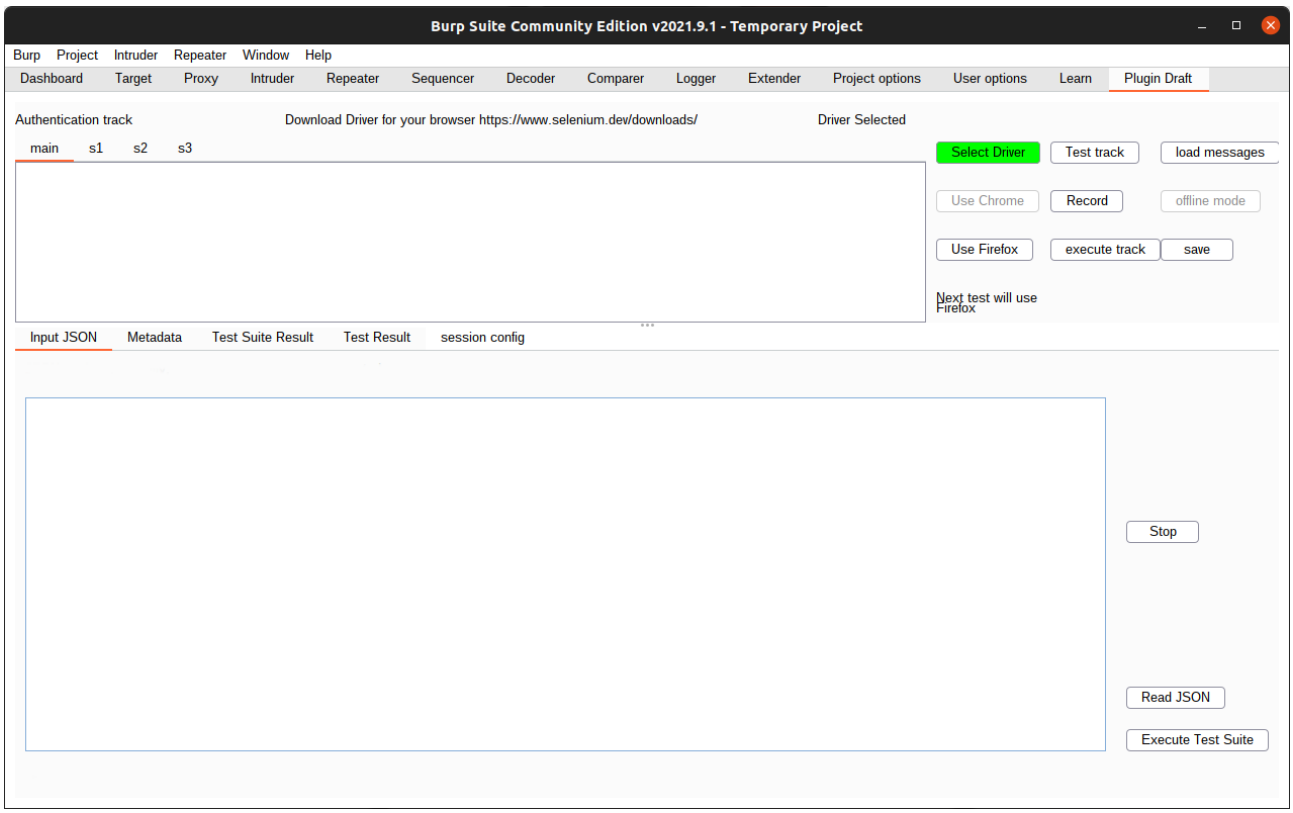

Figure 4.1: plugin interface

- "Input JSON" tab is used to load the tests written in the language into the plugin, and with the use of two buttons we can parse the language and execute the tests.
- "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.

4.1.2 Test execution

The test execution differs from static to dinamic, as static tests don't need the 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. I have also added the possibility of exporting the saved messages to a file, in a way that they can be imported in the plugin and tested again. On the other hand, active tests needs to edit the messages, so the execution of the track has to be repeated for each test.

4.1.3 Decoding & encoding of parameters

As said in the previous chapter, the encoding and decoding of parameters is possible. To do that, a list of encodings has to be provided, i.e. url, base64, deflate. Once the specified message is intercepted, the parameter is taken and decoded following the order of the provided encodings. To do that, i used part of the code of SAML Raider [1] which did the decoding of SAML Requests and responses parameters. I've taken that part of the code and edited it to fit the plugin. SAML Raider is a Burp's plugin used to manage SAML certificates.

4.1.4 SAML certificate managing

In SAML Requests and responses there is sometime the need to remove or edit the certificate associated to that request or response, so, to speed up the process I did add a specific tag in the language to remove or edit the certificate signature. There is still the possibility of doing it by editing the SAML

request or response with a regex, but this way is more convinient. To do this, i used a part of the code of SAML Raider [1], editing it to fit my needs.

4.1.5 Oracle

To identify abnormal pages like error pages the session track evaluation should be sufficient, because if some of the actions could not be executed means that the original "flow" of pages was not followed.

4.1.6 Session managing

The sessions are managed independently, 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 ran in a separated thread to make parallelism possible. By the use of specific commands in the language, 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.

5 Uses cases

In this chapter I will talk about some examples of use cases in which my language could be used.

5.1 SAML Use-Case

My work has been used by Sofia Zanrosso, for her bachelor thesis, her objective was to search for SAML vulnerabilities and to define a series of well-known test to verify them. She defined all the tests using my language and tested multiple SAML implementation with it.

5.2 OAuth & OIDC Use-Case

6 Related work

In this chapter I will talk about other tools and works that are related to mine.

6.1 SSO Testing language and Plugin

A work that is similar to mine, is the one done by my colleague Wendy Barreto [5] in her bachelor thesis at Università di Trento to test OAuth and OIDC SSO protocols with a custom test definition pattern. Previously in this thesis I said that I have taken part of her work as a base to start with mine, editing it to suit my needs. My idea of a language that could be used for any type of test over HTTP 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 restiction on a specific protocol. One of the things that I used is the interface of her plugin, adding some buttons and tabs to deal with multiple session tracks and added functionalities.

6.2 Micro ID Gym

Another plugin for burp that was developed by two my colleagues was Micro ID Gym, a tool to test OIDC and OAuth implementations. This plugin was used by Wendy Barreto to do her work. The old plugin of Stefano and Claudio was based on a track, which defined some actions to be done by the browser, which was a selenium istance. The plugin checks the messages and based on the test defined in the plugin tells if there is a vulnerability or not. Starting from the first version of Stefano, to the last of Wendy, the plugin was improved. In the first two versions of Stefano and Claudio the plugin

had its tests hard-coded, in a way that ony the supported tests could be executed, with little settings to change. If a new test had to be implemented, the plugin had to be recompiled. This version of the plugin worked well, but as said, the tests could not be customized or adapted by the user.

6.3 Last plugin version

Wendy improved the plugin by removing the staticity of the test, adding the possibility to customly define all of the tests with the use of a JSON language.
This is the last version of the plugin which i started working to. The plugin supported the definition of passive and active tests: passive tests are tests where the messages are not edited, active tests are tests where there could be an edit of one or more messages. The availabe test actions worked well, but there were some limitations on the possible actions, especially in the 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 which message to check or edit for static tests was limited: (only "Authorization grant message", "Response messages", "Request messages" and "All messages")
- Only regex were supported to search something in a message
- Unable to work over encoded parameters
- Impossibility of doing multiple operations on a single message
- Impossibility of saving a parameter and using it somewhere else

Some of which stated as future works in Wendy's thesis. Moreover, the language was thought to be used with tests for OIDC and OAuth, other SSO protocols such as SAML could not be tested, because of the fact that SAML parameters are encoded, so editing them or verifying them is not possible. This is the biggest limitation that made me decide to redesign the language.

7 Conclusions

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec sed nunc orci. Aliquam nec nisl vitae sapien pulvinar dictum quis non urna. Suspendisse at dui a erat aliquam vestibulum. Quisque ultrices pellentesque pellentesque. Pellentesque egestas quam sed blandit tempus. Sed congue nec risus posuere euismod. Maecenas ut lacus id mauris sagittis egestas a eu dui. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Pellentesque at ultrices tellus. Ut eu purus eget sem iaculis ultricies sed non lorem. Curabitur gravida dui eget ex vestibulum venenatis. Phasellus gravida tellus velit, non eleifend justo lobortis eget.

Bibliography

[1]

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

[3] 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.

[4] autore. Cia triad. link, 2019.

[5] Wendy Barreto. Design and implementation of an attack pattern language for the automated pentesting of oauth/oidc deployments. Master's thesis, Università degli Studi di Trento, 2021.

[6] 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.

[7] Stefano Faccini. Design and implementation of an automated tool for checking saml sso vulnerabilities and spid compliance. Master's thesis, Università degli Studi di Trento, 2019/2020.

[8] Claudio Grisenti. A pentesting tool for oauth and oidc deployments. Master's thesis, Università degli Studi di Trento, 2019/2020.

[9] IETF. The oauth 2.0 authorization framework. https://datatracker.ietf.org/doc/html/rfc6749.html, 2012. Page 1, Last accessed 29/11/2021.

[10] B. Campbell IETF. Security assertion markup language (saml) 2.0 profile for oauth 2.0 client authentication and authorization grants. https://datatracker.ietf.org/doc/html/rfc7522, 2015. Page 2, Last accessed 29/11/2021.

[11] Giulio Pellizzari. Master's thesis, Università degli Studi di Trento.

[12] Wikipedia. Single sign-on. https://en.wikipedia.org/wiki/Single_sign-on. Last accessed 29/11/2021.

Allegato A Language compartison 1

Action	Old language	New language
Custom message filtering	Only on active tests	supported
Edit string	only by regex	supported with regex and check construc
Remove string	only by regex	supported with regex and check construc
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

Allegato B PKCE test Example complete

This is the complete PKCE 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"
            }
        }
    ]
```

```
45 }
```