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

This thesis covers the work started in my internship at Fondazione Bruno Kessler (FBK) in the context of Single-Sign-On (SSO) protocols testing. SSO protocols are becoming more and more popular these days, and are being used in very sensitive applications such as SPID (Sistema Pubblico di Identità Digitale) an Italian identity management system, or CIE (Carta di Identità Elettronica) whcih is a digital ID card. Considering 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 most common known vulnerabilities are avoided.
To avoid having to manually test each implementation, an automatic tool is necessary. Moreover, a standard language to define these tests has to be defined. This is what will be shown in this thesis.

Glossary

Burp Burp Suite Community Edition, a web security testing software. 6, 15, 18, 20, 22

OAuth OAuth, is a SSO protocol. 6, 8, 18, 21, 22

OIDC Open ID Connect, is a SSO protocol. 18, 21, 22

SAML Security Assertion Markup Language, is an SSO protocol. 6, 7, 15, 20

session track The list of action the browser is told to do. 16, 17, 20, 22, 23

SSO Single Sign On. 3

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 documents. 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 principle 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 happened for example in Lazio [12, 5], 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 aim of this thesis, but this is a good example of what the consequences of exploiting a vulnerability could be. To ensure that these requirements are satisfied, there is a need to test all the implementations of the protocols and systems 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 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 [2, 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 runs over HTTP, and that could 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 execute them. The test results will be evaluated by an oracle, that is composed by a series of components that are responsible for telling if the test is passed or not.

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

1.1 Contributions

The contributions of this thesis are the following:

- A language to define tests for protocols over HTTP, that makes possible to customly define the tests based on the user needs
- A plugin for Burp Suite to execute the tests defined by the language

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 community edition

Burp Suite Community Edition (from now on 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 redirects the traffic to. The proxy does like a Man In The Middle attack, taking the input traffic from the browser and replying the messages to the target service, giving also 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 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 execute the tests. The plugin (from now on 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 Regex

Regex stands for regular expression, it is a sequence of symbols that define an ensemble of strings that can be matched by it. There is a list of symbols that can be used to specify which characters to match. For example, if we want to find the value of the "Host" header in an HTTP message, we want to define a regex like this one: `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

2.3 IdM protocols: SAML and OAuth

The so-called IdM protocols are protocols that deals with identity management. For the aim of this thesis, SAML (Security Assertion Markup Language) and OAuth 2.0 (from now on OAuth) will be discussed, as they are used in the examples and in related works.

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 more complete explanation can be found in [8]. The difference between SAML and OAuth is how they work, but their objective is the same: to certificate the identity of a given person in different circumstances.

2.3.1 OAuth

As stated in [9], 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 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.

2.3.2 SAML - Security Assertion Markup Language

As stated in [3], "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 how the tool and the testing language have been designed will be introduced. 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 objectives was to think of a language that could define tests that could then be tested over 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 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 (JavaScript Object Notation), which has been used as a basis over which write the tests. JSON is a convenient way of defining hierarchical structures such as tests are. The idea behind this language is that a specific message can be intercepted and checked or edited in some way. The hierarchical 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. To understand this test, a brief introduction of PKCE has to be done, as said in [10]: the Proof Key for Code Exchange (PKCE, pronounced pixie) extension describes a technique for public clients to mitigate the threat of having the authorization code intercepted. The technique involves the client first creating a secret, and then using that secret again when exchanging the authorization code for an access token. This way if the code is intercepted, it will not be useful, since the token request relies on the initial secret. 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 process in a way that PKCE will not be used if the service is vulnerable [13]. To test this, we have to intercept the authorization request message, remove the "code_challenge" parameter, and then forward it. The complete test can be found in Attachment B

3.2 Language structure

Each object of the language is a JSON Object which can contain different 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

To implement the PKCE test example above, the Test Suite can be defined in this way:

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

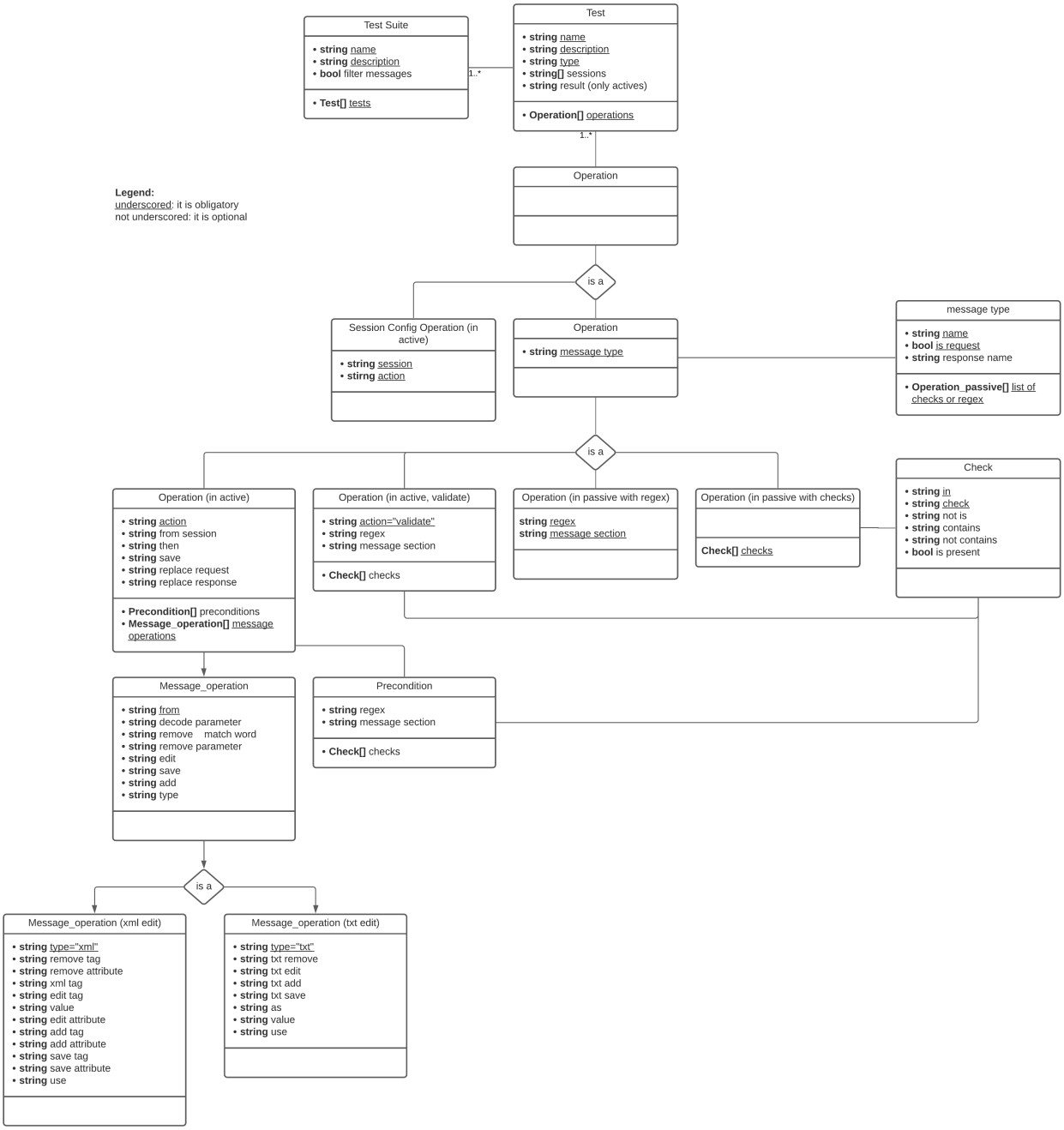

Figure 3.1: Language structure

```
    ]
}
```

Listing 3.1: Test Suite definition

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 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 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 for it:

```
{
    "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

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 this 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 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, but becomes like a passive operation, because its objective is just to verify that some messages meets some requirements. It will contain 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 the 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
    ]
}
```

Listing 3.3: Operation definition

3.2.4 Message section

The message section specifies in what part of the message execute the given action. The possible message sections are:

- **url**, only for request messages, is the message without the head and body parts
- **head**, it is the message without url and body parts
- **body**, it is the message without url and head parts

3.2.5 Check Object

The Check Object is used in Operations and in other Objects to verify that, in a message, some circumstances are satisfied. For example, 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 (head, body or url)
- **check param**, specifies the parameter name to be searched

Also the actual checks on the parameter value has to be defined: (if none of these are 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 we want to check the presence or absence of the parameter.

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 example, if a parameter should not be present in a given message, the check that has to be done is the one that verify that the parameter is **not** present. This means that the Check objects should not be thought like an "if then else" construct. If the Check is evaluated to false, the test will not pass.

3.2.6 Precondition and validate Objects

Check Objects in active tests will not work as in passive tests, there is another way of using them: using them in a Precondition, which is basically 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 done to allow the Oracle to work, differentiating between a precondition and a validation. This way we can define Validate Objects to define the oracle, and Precondition Objects to impose some criteria to allow the execution of the test.

Precondition Preconditions are used in an operation of an active test to check something in the intercepted message before the execution of 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 that it is not known if a given test is "compatible" with a given web service. For example if the service doesn't use some type of protocol, we can assure this by the use of preconditions, checking for the presence of common parameters of the protocol. Preconditions can be also regex.

Validate In an Operation Object the only way to use Check objects is by setting the "action" tag to "validate", this will transform the Operation in a Validate Operation. This Validate Operation will be used by the Oracle to decide whether the test should be considered passed or not. Validate Operation has to be used when we want to check that something in a message is how we expect, if it is not, the result of the test will be considered failed. We can use the Validate Operation with a list of checks or a regex (exactly like in passive tests).

3.2.7 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 it can be 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")

3.2.8 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, at the end of the Message operation will be encoded again automatically before forwarding it. The decoded parameter can be manipulated by means of the "**type**" tag, there is the possibility to interpreter the decoded parameter by two means:

Type txt Associated to 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 name specified in the **as** tag

All the previous tags accept a regex, and whatever that regex matches will be edited or 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 we have various possible operations to be done on the decoded xml:

- **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 above, we have just a simple Message Operation 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 changes also 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 "`(?<=SAMLResponse=)[^$\n& ]*`" that will search for the text "SAMLResponse", taking everything after the "=" until end of line or "&" or whitespace is found

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

3.2.9 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**, in the case that the searched message is a request message, we can use this tag to associate a name to the response of that message. This is useful when we can identify the request message but not the response 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
                }
            ]
        }
    ]
}
```

Listing 3.5: Message Types definition

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, which decides whether a test should be considered passed or failed (or not applicable). I decided to build the oracle in a way that it 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 (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 above, both the result of the test and the precondition has been used:

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

Listing 3.6: Precondition definition

With this precondition, we want the test 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 accomplished, because the objects that should be pressed in the page are not present, this happens for example if an error page is displayed, which can 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, 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, in a way that (i.e.) reply tests can be executed. Different sessions can have different session tracks.

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 the idea used in [11, 8, 6], 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.it 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

In this chapter the implementation of the language and the tool, the problems faced, and the solutions adopted will be discussed

4.1 General overview of the tool

All the components of the final tool can be seen in 4.1. Burp Suite is composed by its proxy and related APIs, the tool will get all the messages from the proxy through the API, and then, it will return them to the API and so, to the proxy. This way all the messages will pass through the tool, and will make possible to make checks 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 possible to see plain HTTP communications on the browser side. Each browser will be supplied with the user actions which will be 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). The most important component of the tool is the Test Suite defined with the language, which is supplied to the plugin, and executed in ensemble with the sessions, giving eventually a result.

4.2 The tool

For the implementation of the Burp's tool introduced earlier, 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 my implementation. The interface of [2] has been taken and adapted to fit the needs of this work. The tool code is written in Java, I used the Burp's interface classes to interact with it. The standard usage of Burp 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 tool 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.

4.2.1 Interface of the tool

In figure 4.2 we can see the interface of the tool, 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 part we have multiple tabs:

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

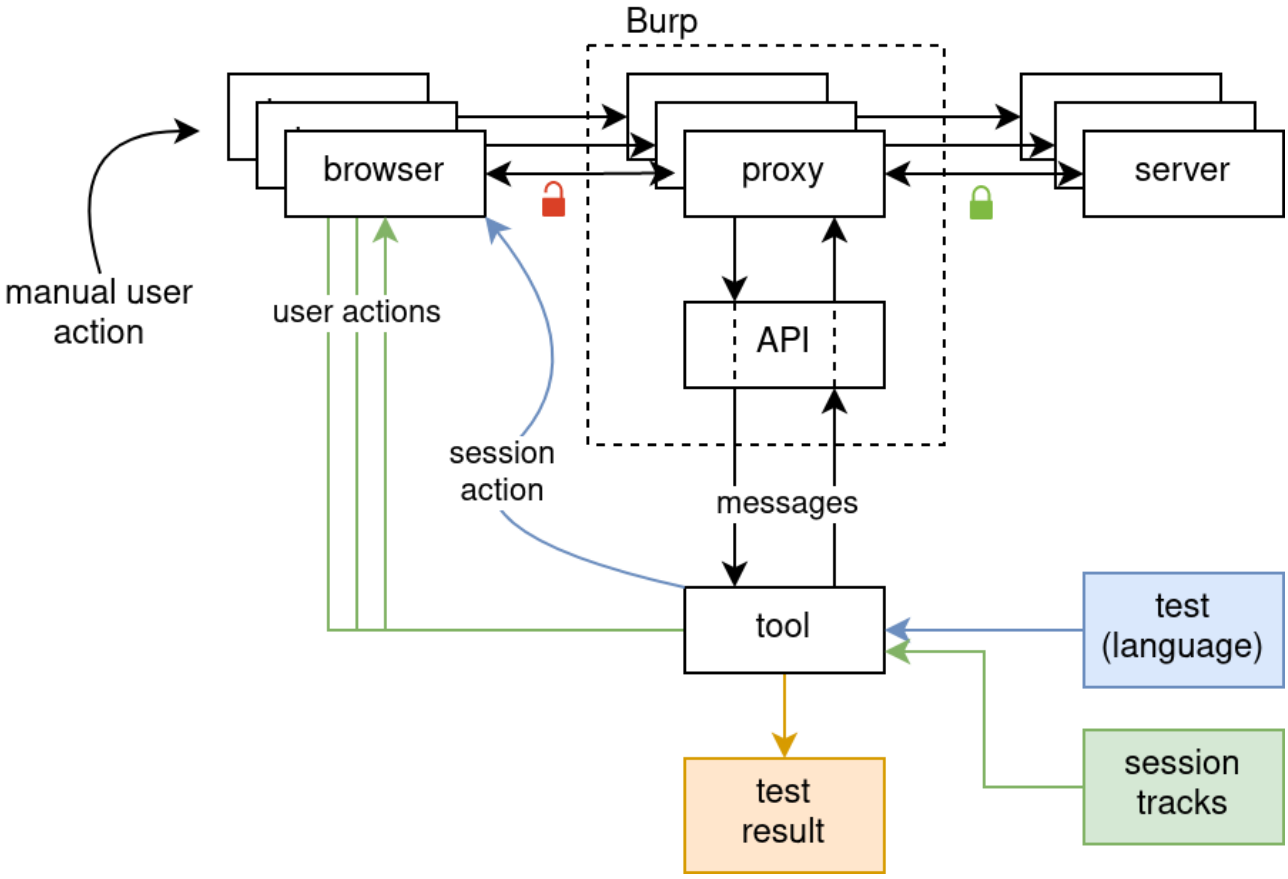

Figure 4.1: General schema

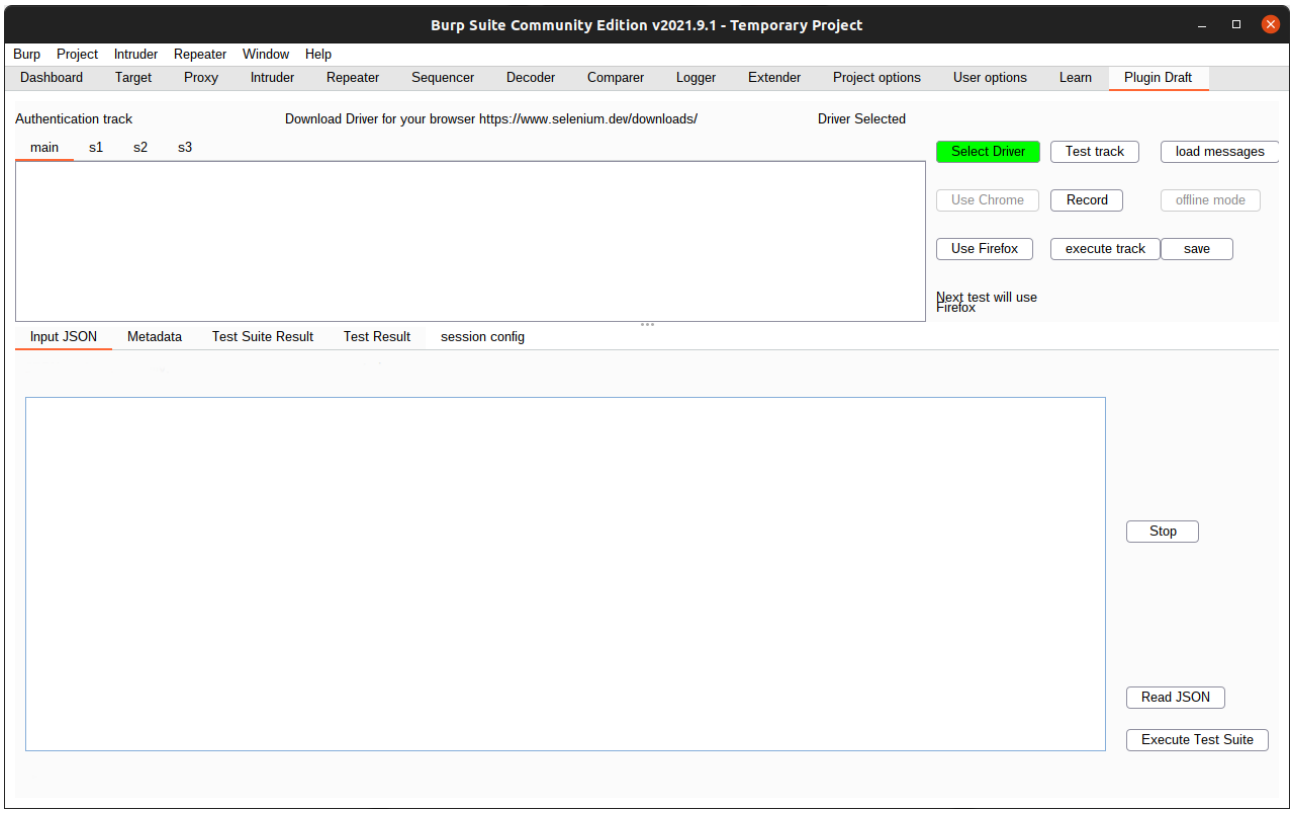

Figure 4.2: Tool interface

- "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.

4.2.2 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. I have also added the possibility of exporting the saved messages to a file, in a way that they can be imported in the tool 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.

4.2.3 Decoding & encoding of parameters

As said in the previous chapter, the decoding and encoding of parameters is possible. To do that, a list of encodings to be done on the parameter 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 list. To do that, part of the code of SAML Raider [14] which did the decoding of SAML Requests and responses parameters has been used. That part of the code has been taken and edited to fit the tool. SAML Raider is a Burp's plugin used to manage SAML certificates.

4.2.4 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 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. In order to do this, a part of the code of SAML Raider [14] has been used, edited to fit the needs.

4.2.5 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 run 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, and so, being able to specific sessions in the tests.

5 Uses cases

In this chapter some examples of use cases in which the work of this thesis has been used will be shown.

5.1 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 testsuite in order to facilitate automatic pentesting over SAML [15]. 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 feedbacks and bugs has been reported to me, speeding up the testing phase of the tool. At the same time, She found that my tool was by some aspect better w.r.t. other ones that She was using. For example:

- Other plugins were giving false positives on multiple tests, while mine was not
- "the previously employed transition times between tools have been greatly reduced"
- "making it possible to analyze almost completely the vulnerabilities of the tested subjects"

5.2 OAuth & OIDC Use-Case(Da mettere?)

The OAuth and OIDC tests defined and used in [8, 2] has been re-defined in the STL language and has been tested on some implementations of the protocols. todo: Da mettere??

6 Related work

In this chapter other tools and works that are related to mine will be discussed.

6.1 Micro ID Gym

Micro ID Gym (MIG) "aims to assist system administrators and testers in the deployment and pentesting of IdM protocol instances" - [7], inside this tool, two pentesting tools can be found:

- MIG - OAuth/OIDC [8]
- MIG - SAML SSO [6]

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, checks the messages in the background, and then, they tell a result. These two plugins are similar to mine, but they are specifically created and defined to test SSO protocols only, and the tests that they used are fixed and cannot be easily edited, if a new test has to be implemented, the plugin has to be recompiled.

6.2 SSO Testing language and Plugin

The preceding two tools of MIG have been improved by a work similar to mine, the one 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 at fixing 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 customly 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 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 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 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
- Impossibility of using multiple sessions in a test

Some of which stated as future works in Wendy's thesis. A more complete list of differences between the two languages and tools can be found in Attachment A.

Previously in this thesis I said that I have taken part of this work as a base to start with mine, editing it to suit the needs and objectives that had been defined. The 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 restriction on a specific protocol. One of the things that has been used is the interface of the plugin, that 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, it actually was already used in [8, 6].

7 Conclusions, Limitations and Future Works

In this chapter, the conclusion and limitations of the work of this thesis will be discussed.

7.1 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:

7.1.1 Automation problems

One of the biggest limitations that the tool has, is the session track actions automations, it often happens that some captcha is encountered during execution, making 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 is sometimes limited, because the session track has to been defined over a specific website, doing a set of action that is directly correlated to the website. Whenever the website is changed somehow, for example the IDs or the position of some button 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 has been found yet. This also means that every different web service which has to be tested will need a different session track to be defined. Making it a bit time-consuming to do.

7.1.2 Oracle is sometimes ambiguous

There still is a problem with the Oracle, where sometimes false positives or negatives arise if the execution of the tool 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 impossible to distinguish if the session track has failed because on an error on the definition on it, or because of an expected reason (like after a message modification).

7.1.3 Interface and user feedbacks

The interface of the tool is a bit raw, it is not very user-friendly, the user experience could be improved. During my work I did not focus my attention to these topics, but they are very important 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 surely has to be fixed, making more clear to the end user what is going wrong.

7.1.4 Message filtering could be improved

The message filtering part of the language could be extended by the use of an AI, making it possible to define a more abstract filter that does not solely rely on the search of parameters or exact strings.

7.2 Conclusions

The objectives fixed at the start of this thesis can be considered reached, the new tool and language is fully working as can be seen in [15], still, it is not perfect in terms of stability and affidability, as a

deep testing phase should be accomplished, but overall, the tool is functioning well.

7.3 Future works

First of all, the problems and the limitations should be solved. Then, the concept of a 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 so on. Making it possible to search for known vulnerabilities in real-time or just by analyzing saved packets files. For example, a plugin for Wireshark software could be built.

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.

[3] 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.

[4] 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.

[5] 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.

[6] Stefano facchini. Design and implementation of an automated tool for checking SAML SSO vulnerabilities and SPID compliance. http://www.ictbusiness.it/, 2019/2020.

[7] FBK. Micro-ID-Gym. https://st.fbk.eu/tools/Micro-Id-Gym.html, 2020. Last accessed 1/12/2021.

[8] Claudio Grisenti. A pentesting tool for OAuth and OIDC deployments, 2019/2020.

[9] 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.

[10] Okta. Protecting mobile apps with PKCE. https://www.oauth.com/oauth2-servers/pkce/, 2015.

[11] Giulio Pellizzari.

[12] 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.

[13] Nat Sakimura, John Bradley, and Naveen Agarwal. Proof key for code exchange by OAuth public clients. *Internet Engineering Task Force (IETF), Published Sep*, 2015. Page 28, last accessed 11/12/21.

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

[15] Sofia Zanrosso. Development of saml testsuite to facilitate automatic pentesting, 2020/2021.

Attachment A 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 construct
Remove string	only by regex	supported with regex and check construct
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

Attachment B PKCE test Example complete

```
{
    "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"
            }
        }
    ]
}
```