Bachelor Thesis

Automatic Recognition, Processing and Attacking of Single Sign-On Protocols with Burp Suite

Tim Guenther

Date:	12/10/2015
Supervisor:	Prof. Dr. Jörg Schwenk
Advisor:	Dipl.-Ing. Vladislav Mladenov, M. Sc. Christian Mainka

Ruhr-University Bochum, Germany

Chair for Network and Data Security
Prof. Dr. Jörg Schwenk
Homepage: `www.nds.rub.de`

Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnliche Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen der Entlehnung kenntlich gemacht. Dies gilt sinngemäß auch für gelieferte Zeichnungen, Skizzen und bildliche Darstellungen und dergleichen.

Ort, Datum	Unterschrift

Acknowledgements

I want to thank Context Information Security Ltd., especially Christian Becker, Sven Schlüter, Jacek Groth and Bernhard Gomig, who supported me throughout the thesis. A special thanks goes to my advisors at the Chair for Network and Data Security, Vladislav Mladenov and Christian Mainka. They helped me with their enormous knowledge about Single Sign-On (SSO) and this work would not be possible without them. Their willingness to answer hundreds of mails was awesome. Last but not least, thanks to my lector Katherine Abercrombie.

Abstract

EsPReSSO does not only wake tired people all over the world but is now also a simple tool to use for recognition and attacking of Single Sign-On (SSO).
Single Sign-On is supported by a huge amount of web services and web applications. The major protocols are SAML, BrowserID, OpenID and the OAuth based protocols, namely OpenID Connect, Facebook Connect and Microsoft Account. In order to help researchers to distinguish between these protocols, EsPReSSO offers an automatic identification and presentation of the major protocols by analyzing the browser's HTTP traffic. Moreover, with the integration of the famous web service attacking tool, WS-Attacker [1], it is possible to attack SAML with over 200 vectors. This thesis will present the fundamentals required to understand EsPReSSO and its internal structure.

KEYWORDS: Burp Suite, Single Sign-On, Recognition, OAuth, OpenID Connect, Facebook Connect, Microsoft Account, OpenID, BrowserID, SAML, Signature Faking, Signature Wrapping

Contents

List of Figures

List of Tables

List of Acronyms

General

API Application Programming Interface

EsPReSSO Extension for Processing and Recognition of Single Sign-On

HTML Hyper Text Markup Languages

HTTP Hyper Text Transfer Protocol

IDE Integrated Development Environment

IT Information Technology

JSON JavaScript Object Notation

JWT JSON Web Token

SSO Single Sign-On

SAML Secure Assertion Markup Language

UI User Interface

URL Uniform Resource Locator

XML Extensible Markup Language

XRDS Extensible Resource Descriptor Sequence

Single Sign-On related

DTD Document Type Declaration

IdP Identity Provider

SP Service Provider

XXE XML External Entity

Organisations

IETF Internet Engineering Task Force

OASIS Organization for the Advancement of Structured Information Standards

W3C World Wide Web Consortium

1. Introduction

A common authentication mechanism is a combination of username and password which can be hard to remember as far as it chosen in a secure way[1]. With each reuse of a password, the difficulty of password management increases. A solution to reduce the password reuse is Single Sign-On (SSO), a technique which allows user to login with the same credentials over multiple websites. The authentication process is delegated to an Identity Provider (IdP), such as Facebook, Microsoft, Twitter or Google and makes it therefore easier for users to log in to web applications using their existing accounts. The technologies and protocols used by the IdPs are partly closed or open source or not documented and considering the fact that some of the protocols are based on the same structures and behaviors makes them therefore hard to distinguish and classify. The Burp Suite Extension for Processing and Recognition of Single Sign-On (EsPReSSO) provides support for penetration testers and researchers to identify and classify SSO protocols as well as attack protocols manual and/or automatically. To the best of our knowledge, there is no tool, which can add support to identification process of SSO protocols.

Contributions. The main contributions by EsPReSSO are:

- Recognition of the protocols SAML, BrowserID, OpenID, OAuth, OpenID Connect, Facebook Connect and Microsoft Account.
- A visualization of the detected protocols integrated in Burp Suite, as well as a user interface designed for SSO protocols.
- Editors to view and modify special encoded formats such as SAML, JSON and JWT.
- Integration of the WS-Attacker [1] to allow over 200 automatic, semi-automatic and manual attacks on SAML.

Structure. This thesis is structured as follows:

- **Chapter 2** provides the foundations for understanding the implementation of EsPReSSO and is a starting point for further research. The topics discussed are Single Sign-On in general, the basic technologies such as XML, JSON and JWT as well as the focused protocols and Burp Suite.
- **Chapter 3** illustrates the User Interface (UI) of EsPReSSO and the workflow with screenshots of the extension. Every aspect and feature is explained in detail.
- **Chapter 4** explains the implementation and internal structure of EsPReSSO. The chapter starts with a description of a guide to extend Burp Suite, followed by the internal structure, a tutorial on the extension of EsPReSSO, an evaluation of the Scanner, as well as the limitations.
- **Chapter 5** provides the conclusion and a look at the future development of EsPReSSO.

[1]The term secure is in this case referred as a random string containing upper- and lowercase characters, numbers and symbols, with a length greater then 8 characters [2].

2. Foundations

The following section presents a short overview of the standards and technologies used by the SSO protocols. This chapter provides the reader with the foundation for further research and should help to understand the user interface and implementation of EsPReSSO.

2.1. XML - Extensible Markup Language

Extensible Markup Language (XML) is a data description language, which is standardized by the World Wide Web Consortium (W3C). XML is widely accepted within the IT industry and used by applications, such as Android [3] and Microsoft Office [4]. XML consists roughly of tags or nodes, attributes, and data. Figure 2.1, describes an example XML structure.

```
<?xml version="1.0" encoding="UTF-8"?>
<Standards>
        <Standard id="1">
                <type>RFC</type>
                <number>5849</number>
        </Standard>
        <Standard id="2">
                <type>EMAC</type>
                <number>262</number>
        </Standard>
</Standards>
```

Figure 2.1.: XML example

The root node of the XML document is `<Standards>` and enclosed all elements. the children of the root node are identified by the tag `<Standard>`. The tag has additionally an attribute, which contains the `id` of the object. The tags `<type>` and `<number>` are containing the data.
For further information read the W3C recommendation [5].

2.2. JSON - JavaScript Object Notation

JavaScript Object Notation (JSON) is based on a subset of the EMACScript Standard 262 [6] which specifies a human-readable data-interchange format. The format is language independent and based on a collection of name/value pairs and ordered list of values. The structure has equivalents in every modern programming language which makes it a good choice for communicating in the world wide web. Figure 2.2 shows a representation of Figure 2.1 as JSON object.

```
{
  "Standards" : {
    "Standard": {"type" : "RFC", "number" : 5849},
    "Standrad": {"type" : "EMAC", "number" : 262}
  }
}
```

Figure 2.2.: JSON example

2.3. JWT - JSON Web Token

JSON Web Token (JWT) is, as described in RFC 7519 [7], a URL-safe, compact and JSON-based format for the exchange of claims between web applications. The payload can be encrypted by a JSON Web Encryption [8], or integrity protected with a JSON Web Signature [9].
The format consists of three Base64[1] encoded strings. The strings are than concatenated with a dot, as demonstrated in Figure 2.3.


```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZS
I6IlRpbSBHdWVudGhlciIsIndvcmsiOiJXcml0aW5nIFRoZXNpcyJ9.-axL6pTuJ0ZDz7xX
eMMkQ8l7k3Egtbet0d0tszY_g84
```


Figure 2.3.: Encoded JWT Element

The first part of the JWTis the header which describes the algorithm used for the signature and the token type. The next part describes the payload, a simple JSON object with standard fields. The signature is the last element and in this case a `HMACSHA256(base64UrlEncode(header)+'.'+base64UrlEncode(payload), secretValue)`[2]. Figure 2.4 presents the three decoded parts of a JSON Web Token.

```
{"alg":"HS256", "typ":"JWT"}
```

```
{"sub":"1234567890", "name":"Tim Guenther", "work":"Writing Thesis"}
```

```
6b 12 fa a5 3b 89 d1 90 f3 ef 15 de 30 c9 10 f2
5e e4 dc 48 2d 6d eb 74 77 4b 6c cd 88 3c
```

Figure 2.4.: Encoded JWT Header, Payload, Signature

[1]Base64 encoding maps the binary representation of a string to 64 ASCII characters [10].
[2]`HMAC` stands for *keyed-hash message authentication code* [11]. `SHA256` is the used hash algorithm.

2.4. The Basics of SSO - Single Sign-On

Registrations and login are important security mechanisms on the Internet today. A website where users have to identify themselves, for example for finacial transactions or accessing private content such as mails, profiles or photos, needs a kind of authentication or sign-on process. The SSO concept was developed in order to facilitate registration processes for both users and developers. Without SSO, the user has to register an individual account for each site, which increases the password management effort.

SSO describes how a sign-on process could be shared by numerous websites. The idea consists of to two different servers (Identity Provider (IdP) and Service Provider (SP)), and a user (Client). Figure 2.5 illustrates a specific example of a SSO scenario. The protocol sequence starts with the client trying to access a restricted resource from the Service Provider. The SP requests a login token. This token request is then forwarded to the Identity Provider, on which the actual login, and the authentication of the user, is conducted. In the next step the IdP generates a token for the client, which is afterwards used to grant access and validate the users identity.

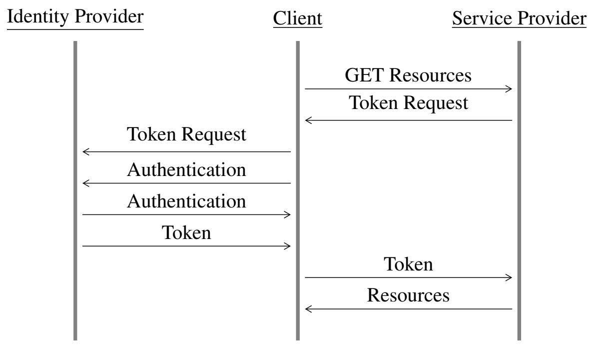

Figure 2.5.: Abstract Single Sign-On Protocol Flow

In the case of SSO the terms authentication, for instance verifying ones identity, and authorization, for instance verifying ones right to access a resource, often overlap but must be considered completely different.

2.5. Researched SSO Protocols

The investigated Single Sign-On protocols can be divided into two groups. The first group includes all protocols that are not based on OAuth, such as OpenID, SAML and BrowserID. The protocols in the second group are based on the authorization protocol OAuth. These protocols make use of OAuth typical parameters and behaviors. This intersection between the protocols increases the difficulty to distinguish and to analyze them. The following section provides a brief overview and guidance for further research. Each subsection is structured as follows: an introduction with a reference to the *official documentation*, a short overview over the used *technologies* and possible *dependencies* on other protocols. A common characteristic between all protocols is the data exchange with HTTP GET or POST parameters. A profound description of each protocol is considered to be out of scope for this thesis, but it is worth noting that the fundamentals used here for the recognition and distinction between the different protocols is based on Mainka et al. [12].

2.5.1. Non-OAuth Protocols

The following protocols are not based on the OAuth framework and are therefore easier to differentiate due to their uniqueness.

OpenID

OpenID, developed by the OpenID Foundation, is an authentication protocol used mainly for Single Sign-On. The focused versions are OpenID 1.1 [13] released in 2006 and the last version OpenID 2.0 [14] which was released in 2007. With OpenID it is possible to setup private IdPs, caused by its decentralized approach, and therefore the IdP must be negotiated in the discovery phase with the SP. A typical part of OpenID is the HTTP-GET parameters starting with `openid`. For IdP discovery OpenID 2.0 uses a XML derivation, called Extensible Resource Descriptor Sequence (XRDS), or HTML. As one of the first standardized SSO protocols, OpenID has no dependency on any of the other reviewed protocols.

BrowserID

BrowserID is developed and distributed under the name *Persona* by Mozilla. As a monolithic SSO service the communication to the Identity Provider takes place at `https://login.persona.org/`. As a feature, BrowserID supports an interface to integrate existing OpenID and OpenID Connect services, as well as a fallback IdP, as described in [15]. Concerning this interface, the detection of different SSO protocols during the protocol flow should be expected. The protocol data exchange uses JWT and JSON.

SAML - Secure Assertion Markup Language

The Secure Assertion Markup Language (SAML) has existed in its first version since November 2002 as an OASIS standard. The last version is 2.0 and is standardized in [16]. SAML describes a method for the exchange of cryptographic secure information, and is not designed for Single Sign-On but is capable of it. SAML is based on XML and it is possible to use it to sign and encrypt data. It has no dependencies on other protocols.

2.5.2. OAuth-Family Protocols

The following protocols are based on, or make use of, the authorization protocol OAuth. It should be noted that OAuth is an authorization framework and therefore not capable of doing Single Sign-On or authentication.

OAuth

The OAuth protocol is an open source standard for secure authorization for HTTP services. It enables and manages third party access to restricted resources. OAuth version 1.0 is standardized in RFC 5849 [17] by the Internet Engineering Task Force (IETF), and the current version, OAuth 2.0, was standardized in RFC 6749 [18] in late 2012, as described in [19]. OAuth is capable of using both, XML(in particular SAML) and JSON (in particular JWT) for data interchange, aside from the normal HTTP parameters. The OAuth Protocol has no dependencies on any of the other researched protocols.

OpenID Connect

OpenID Connect is an authentication and authorization framework build on top of the OAuth protocol, and is published by the OpenID Foundation [20]. Its communication is based on JSON and JWT, alongside the standard POST/GET parameters. Despite the similarities in the name, OpenID Connect is a complete different protocol than OpenID [21].

Facebook Connect

Facebook Connect [22] is developed by Facebook Inc. to authenticate users on third party websites and authorize web applications' to access user resources like email address or photos. The protocol is based on OAuth 2.0, and thus it uses JWT for communication [23].

Microsoft Account

Microsoft Account [24] is a proprietary protocol developed by Microsoft. It adopts the OpenID Connect protocol and OAuth framework, as well as the WS-Federation protocol[3] which is not discussed here. Hence, Microsoft Account utilizes the same technologies as OpenID Connect and OAuth.

2.6. Burp Suite

Burp Suite, developed by PortSwigger, is a Man-In-The-Middle HTTP proxy. It is possible to configure a redirection in applications such as a browser in order to forward the network traffic to Burp Suite. Burp Suite is able to intercept each request and response in order to modify the transmitted data. To enable analysis of HTTP messages Burp Suite offers an raw editor and a hexadecimal editor as well as parsed POST or GET parameters and HTTP-headers. As well as this history of processed messages Burp Suite also offers a number of other tools. For example, it is also possible to automatically crawl a website with the *Spider* tool, in order to discover the content automatically. The *Intruder* is capable of injecting user defined payloads multiple times to e.g. brute force a password. With the *Repeater*, a replay of already received messages is possible. The *Sequencer*, *Decoder* and *Comparer* are also build-in tools, but are not necessary for this thesis. The most important feature for this thesis is the possibility to extend Burp Suite via its own API.
For more information on Burp Suite and how to write an extension, see Chapter 4.

[3] `https://msdn.microsoft.com/en-us/library/bb498017.aspx`

2.7. Related Work

Except for the general security analysis of individual SSO protocols there are no papers on the recognition and distinguishing between different single sign-on protocols. SSO tools known to have a similar approach like EsPReSSO are:

SAMLRaider

SAMLRaider [25] is a Burp Suite Extension developed as part of a bachelor thesis by two students at the Hochschule für Technik Rapperswil (HSR), to test SAML setups with tempered SAML messages and manage the certificates. Compared to EsPReSSO, SAMLRaider has more features to modify SAML but no automatic recognition of the SAML or any other protocol. Apart from this, EsPReSSO is based on the WS-Attacker library and therefore able to access over 200 different SAML attack vectors.

SAMLyze

SAMLyze [26] is a penetration testing tool for SAML Service Provider (SP). The tests are focused on preconfigured payloads designed to test against XML External Entity (XXE) and Document Type Declaration (DTD) attacks, as well as a set of SAML validation methods. The user interface is based on a web interface which makes it, according to the author, easy to configure. Furthermore the workflow allows integration with both Burp Suite and Zed Attack Proxy.

3. EsPReSSO User Interface

The following chapter explains the User Interface (UI) of EsPReSSO. The workflow and usage of the extension is discussed in detail, and the following Chapter describes the internal structure and the implementation. Since EsPReSSO is a Burp Suite Extension, the integration was designed to be as close as possible to the look and feel of Burp Suite. Therefore the design is oriented towards the already existing components of Burp Suite. For a guide on how to build EsPReSSO from source and load it in Burp Suite, see Section 4.1.

3.1. Burp Suite Proxy

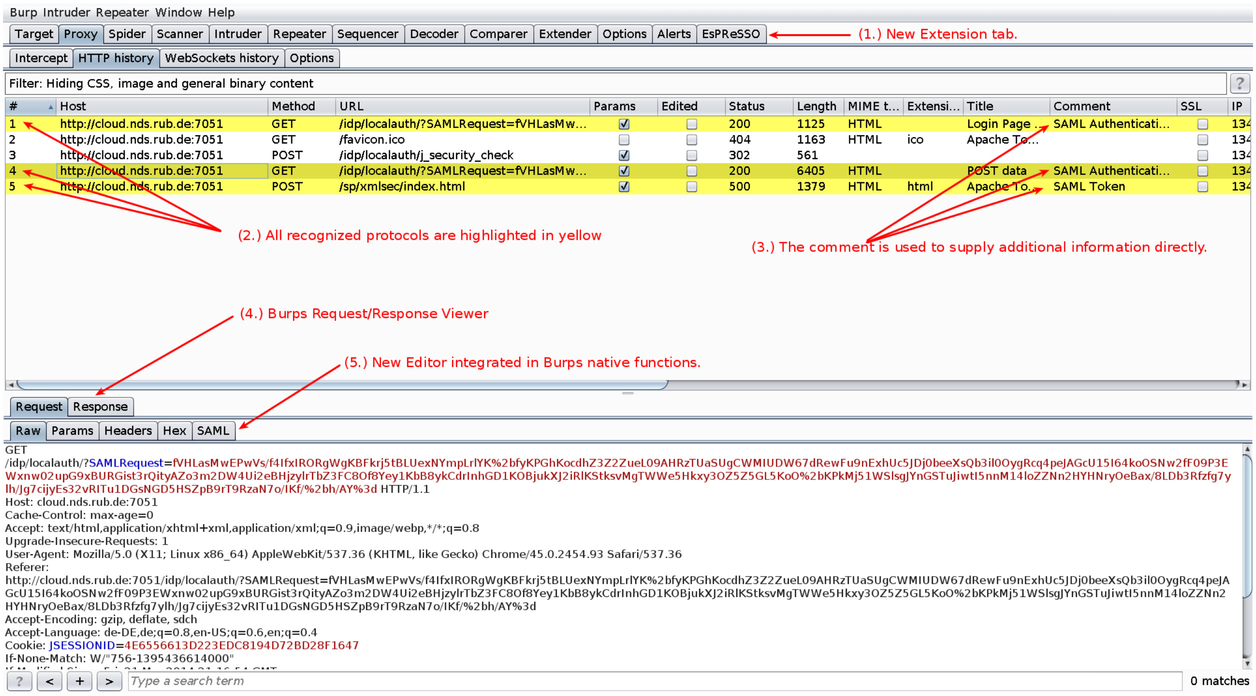

Figure 3.1.: The Burp Suite Proxy HTTP history.

Burp Suite's Proxy HTTP history is a tab which enables the user to review all processed HTTP messages which have been intercepted. Figure 3.1 shows Burp Suite's Proxy window. If EsPReSSO has already been loaded by Burp Suite, then a new tab, called *EsPReSSO* (1.), is attached to the top row. All recognized Single Sign-On protocols are highlighted in yellow (2.). The comment table column shows additional information about the recognized protocol (3.). Burp Suite's *Request/Response* viewer (4.)displays information such as raw HTTP message, parsed parameters and headers. New tabs of EsPReSSO are integrated into this view. For example (5.) shows, the *SAML* tab which decodes the SAML message. For more information on Editors see Section 3.5.

3.2. SSO History

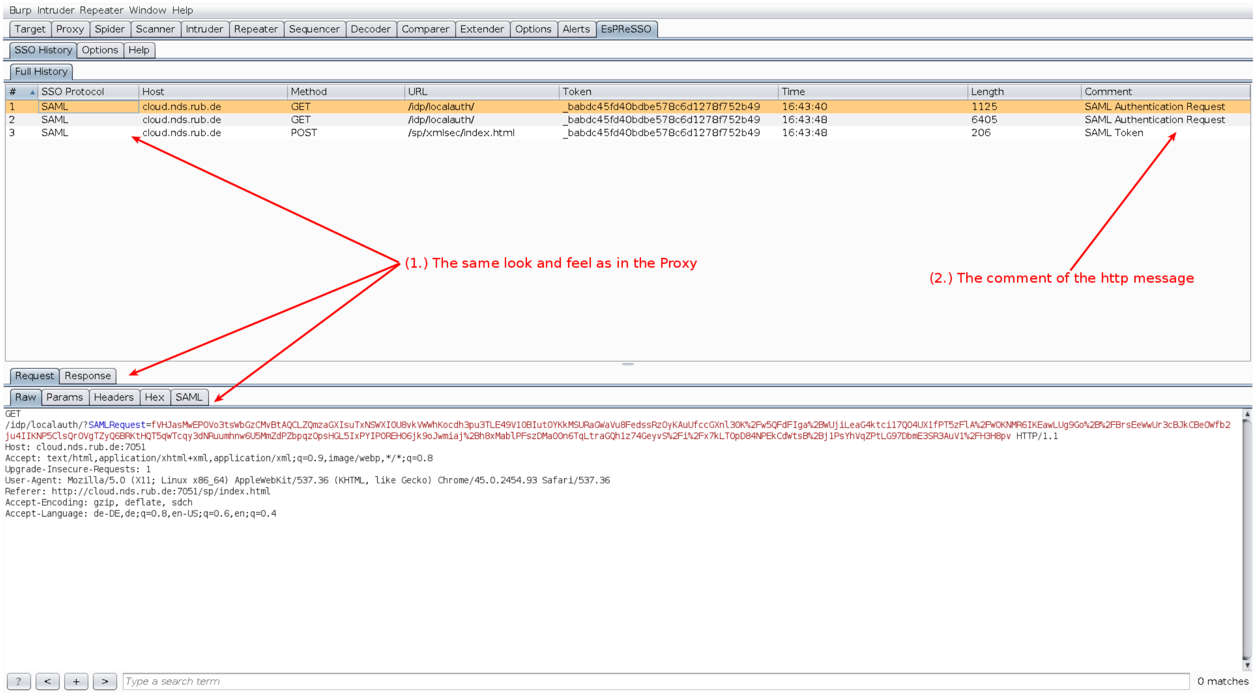

Figure 3.2.: EsPReSSOs Full History, displays all recognized protocols.

The SSO History (Figure 3.2) is based on the layout of Burp Suite's Proxy history (1.). In addition to the typical table entries, the columns titled *SSO Protocol* and *Token* are added. *SSO Protocol* describes the recognized protocol and *Token* displays an identifier of the protocol message.

Figure 3.3 shows the context menu added to each table entry. The extension provides a menu which can be opened, with a right click on a table entry. 'Anayse SSO Protocol' can be selected to start the analysis of the table for coherent SSO messages. Once the analysis is finished, all related entries are copied into a new table which is then attached next to the *Full History* tab. The new table is named after the protocol of the selected entry together with a consecutive number.

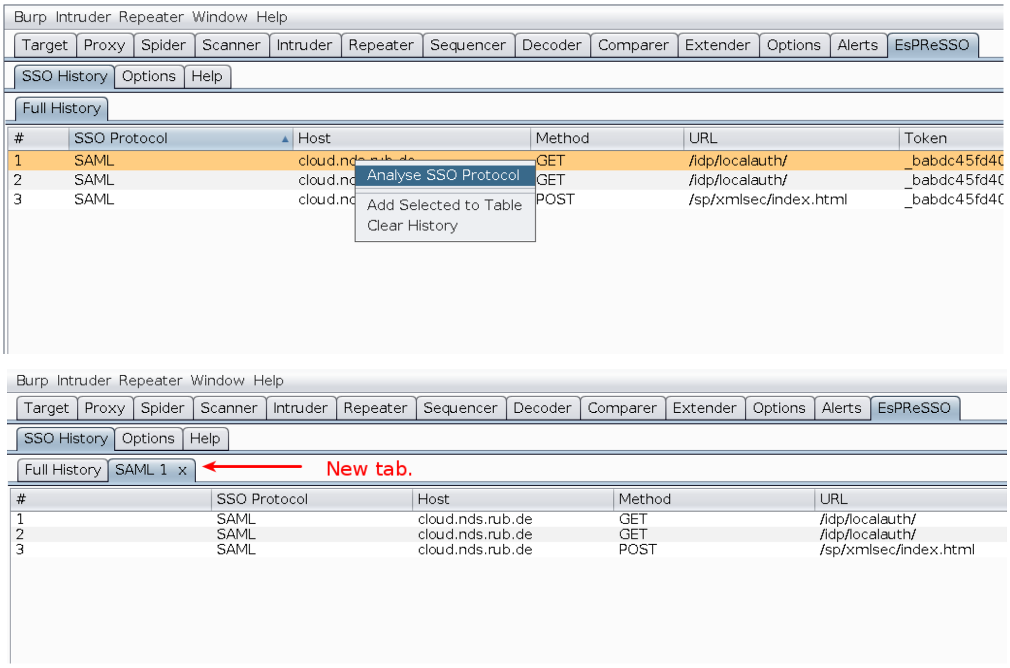

Figure 3.3.: Select 'Analyse SSO Protocol' to open a new tab with all inherent protocols.

3.3. Options

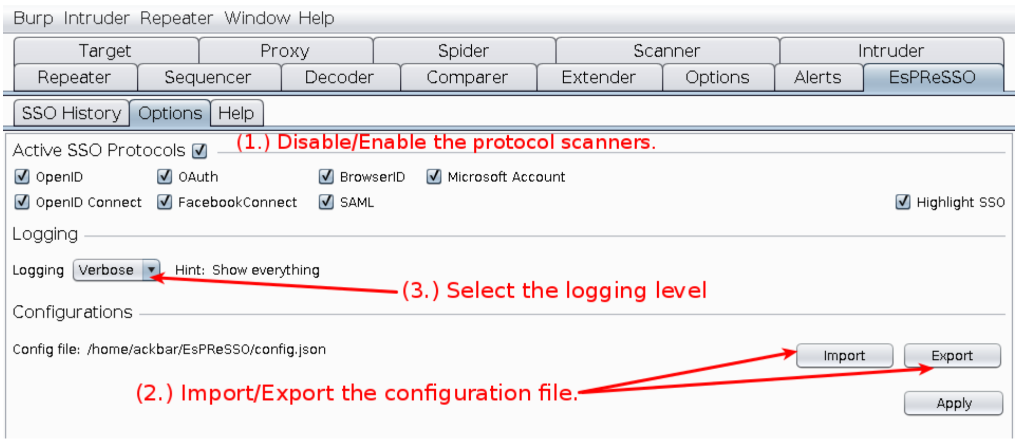

Figure 3.4.: The Options tab.

Via the *Options* tab, as shown in Figure 3.4, the configuration of the extension can be controlled. The checkboxes at the top are used to control the active protocols that are scanned for. If the box is checked, the the specific protocol is enabled during scanning. The checkbox next to the headline disables all protocols at once. To disable the highlighting within the *Proxy* history uncheck *Highlight SSO*. The configuration is stored in a JSON file in the home folder of the user. The user can load or save other configuration files with the buttons *Import* and *Export* (2.). The integrated logger is configurable via the drop-down menu (3.), the options are *Verbose*, which enables all logging levels at once, and *Debug*, which displays only the debugging and the error level messages. The option *Info* shows the info and error level messages. For more detail on where and how the logs are displayed, see Section 4.1.

3.4. Help

Figure 3.5.: The Help tab.

The *Help* tab, shown in Figure 3.5, displays the name, copyright info, license and dependencies of the extension in the about tab. The other tabs have not been implemented at the time of writing, but will later on include information about the recognized protocols.

3.5. Editors

Editors are a way to integrate features in Burp Suite's Request/Response viewer. In this thesis a JSON, a JWT and a SAML Editor were created. The editors attach themselves once the specific content is recognized in the request or response. In the case of the SAML Editor, Figure 3.6, the editor is integrated as soon as a message includes the parameters `SAMLRequest` or `SAMLResponse`.

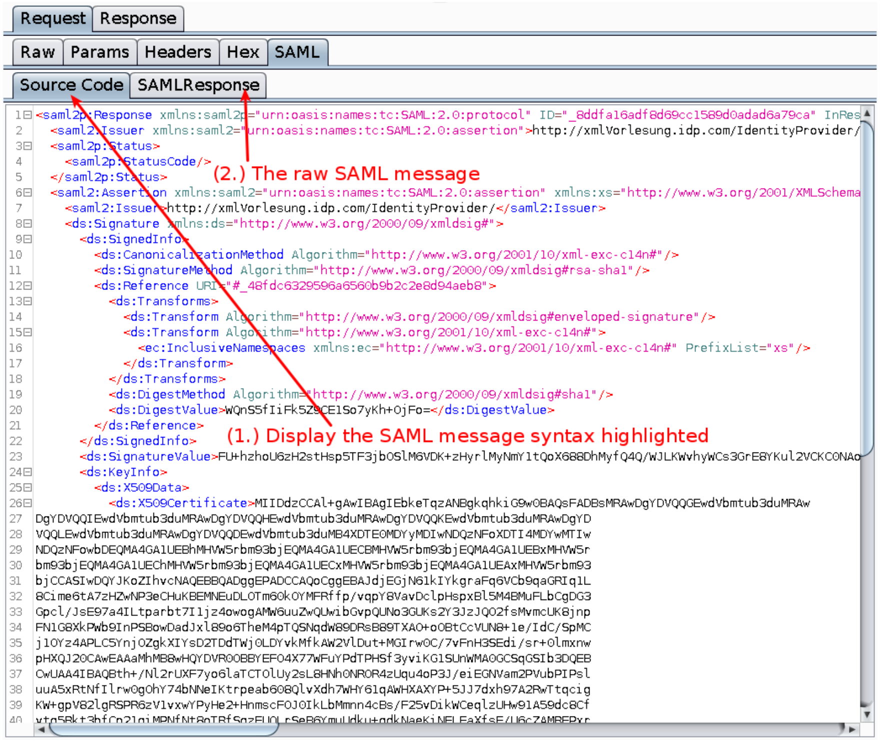

Figure 3.6.: The SAML Editor tab.

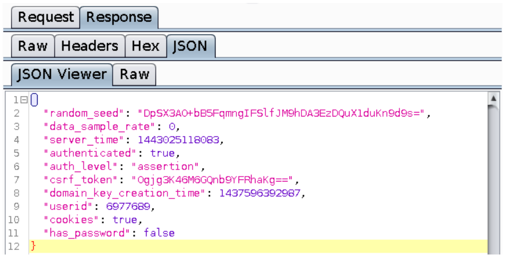

Figure 3.7.: The JSON Editor tab.

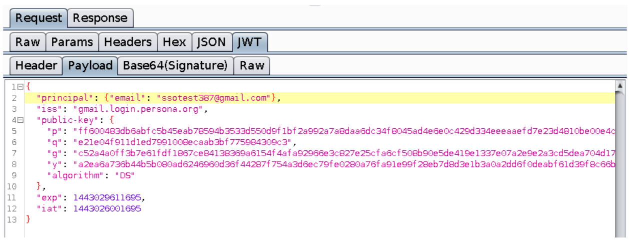

Figure 3.8.: The JWT Editor tab.

The JSON Editor, shown in Figure 3.7, displays beautified JSON code. The tab is attached if JSON was identified as the MIME-type in use. The JWT Editor, shown in Figure 3.8, is attached if a parameter known for JWT is present in the message.

3.6. Attacker

The Attacker tab is only enabled during the interception of a message. While a message is intercepted, it is possible to modify the message and run attacks against the server. The Attacker functionality is only available within the SAML Editor. To start an attack click on the tab (1.) and choose between the possible attacks (2.). The easiest attack to configure is the *Signature Faking* attack [27], where the signature elements in the message are replaced with a new generated signature, see Figure 3.9. Simply click on *Modify* (3.) and the message is altered into a message with a faked signature. To run the attack, press the *Forward* button in order to send the message to the server.

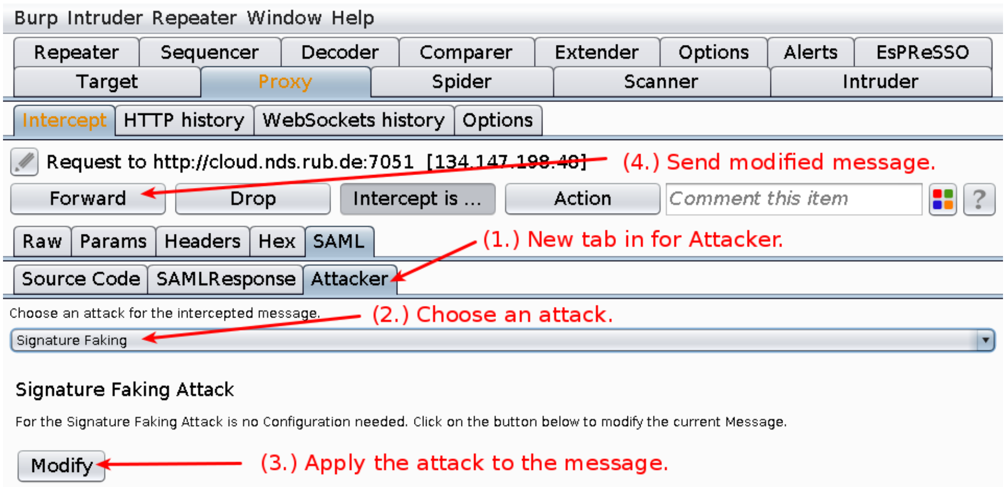

Figure 3.9.: The workflow in the Attacker tab for Signature Faking.

The *Signature Wrapping* [28] attack, shown in Figure 3.10, is more complex. If this type of attack was selected via the drop-down menu, then a possible payload is presented in the text area (5.). To generate possible attack vectors hit *Update Oracle* (6.). The vectors are selected using the slider (7.), a description on what is modified is presented below (8.). The final manual modifications of the attack are made in the last text area (9.), and all modifications can be applied to the message by selecting the *Modify* button (10.).

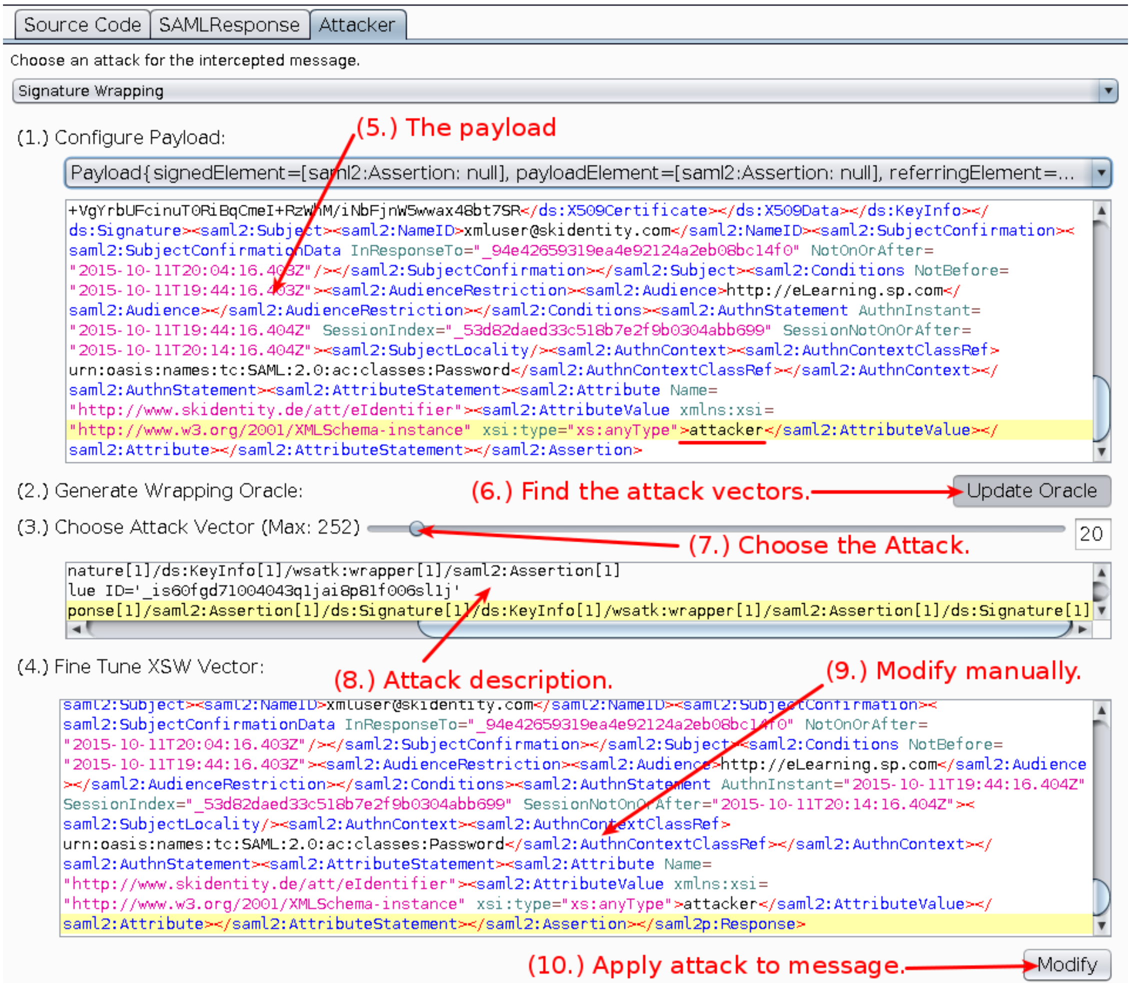

Figure 3.10.: The workflow in the Attacker tab for Signature Wrapping.

The *Attacker*s business logic is based on the WS-Attacker [1], see also Section 4.4.4.

4. Implementation

In this chapter the specific implementation for EsPReSSO is described by documenting all relevant classes, functions and attributes. All diagrams are reduced to the minimum to explain the context.

4.1. Compiling and Loading the Extension

The build management tool used for this is *Maven*, developed by the Apache Software Foundation, which is used to organize dependencies on Java projects. To compile the EsPReSSO tool form source run the commands in Figure 4.1 within the project directory.

```
$ mvn clean -Dskip package
```

Figure 4.1.: Compile the Package form Source, without JUnit tests.

The required Java version is Java 1.8. To successfully start the extension Burp Suite must be started with the same Java version. After Burp Suite is started, navigate to the *Extender* tab and use the *Add* button to add the newly compiled `.jar`-file from the folder `/target` in the project path.
If EsPReSSO has been successfully loaded a new tab called *EsPReSSO* is added to Burp Suite. Under Burp Suite's `Extender / Output` tab, lines, similar to the listing in Figure 4.1, are printed. Printing the output to a file is recommended for better debugging, since the output in Burp Suite is limited and fast exceeded with a stack trace.

```
+--------------------------------------------------------------------+
| EsPReSSO - Extension for Processing and Recognition of Single Sign-on |
|                      Started @ 02:31:32                              |
+--------------------------------------------------------------------+
[I] 02:31:32 - [de.rub.nds.burp.espresso.gui.UIOptions]:
The config from {$homedir}/EsPReSSO/config.json is now loaded.
[I] 02:31:32 - [burp.BurpExtender]:     Tab registered.
[I] 02:31:32 - [burp.BurpExtender]:     Scanner registered.
[I] 02:31:32 - [burp.BurpExtender]:     SAML editor registered.
[I] 02:31:32 - [burp.BurpExtender]:     JSON editor registered.
[I] 02:31:32 - [burp.BurpExtender]:     JWT editor registered.
[I] 02:31:32 - [burp.BurpExtender]:     ExtensionStateListener registered
[I] 02:31:32 - [burp.BurpExtender]:     Init. complete.
```

Figure 4.2.: Initialization Output of EsPReSSO

4.2. System Setup

To develop EsPReSSO the author used following versions of software:

Software	Version
Java	1.8.0_60 (OpenJDK)
Burp Suite	1.6.01
OS (Linux)	4.1.6-1-arch, amd64

Software	Version
NetBeans	8.0.2
Maven	3.3.3

Table 4.1.: Software versions.

4.3. Extending Burp Suite

It is possible to write etensions for Burp Suite in Python, Ruby or Java. EsPReSSO is a *Maven* based project. To integrate the Burp Suite API [29] into the project, add the listing of Figure 4.3 to the `pom.xml` between `<dependencies>` node.

```
<!-- Burp Suite Extension API -->
<dependency>
  <groupId>com.h3xstream.retirejs</groupId>
  <artifactId>burp-api</artifactId>
  <version>1.0.0</version>
</dependency>
```

Figure 4.3.: Burp Suite API

This adds the required interfaces to the project and can be used from now on. For Burp Suite specific questions please refer to the Burp Suite Support [30].

4.4. Internal Structure

The start of every Burp Suite Extension is the class `BurpExtender.java` which must be placed in the folder
`{project}/src/main/java/burp/` and implements the interface `IBurpExtender`. This interface is called every time Burp Suite loads the extension. Figure 4.4 describes the internal process after Burp Suite calls the interface using the method method `registerExtenderCallbacks(IBurpExtenderCallbacks callbacks)`. The `IBurpExtenderCallbacks` object is Burp Suite's main inter process communication interface. The main features and methods of Burp Suite can be triggered with the callbacks object. Within the aforementioned function all new components for the extension are registered such as a new tab, the Scanner and the Editors. The Attacker is a part of the SAML Editor.

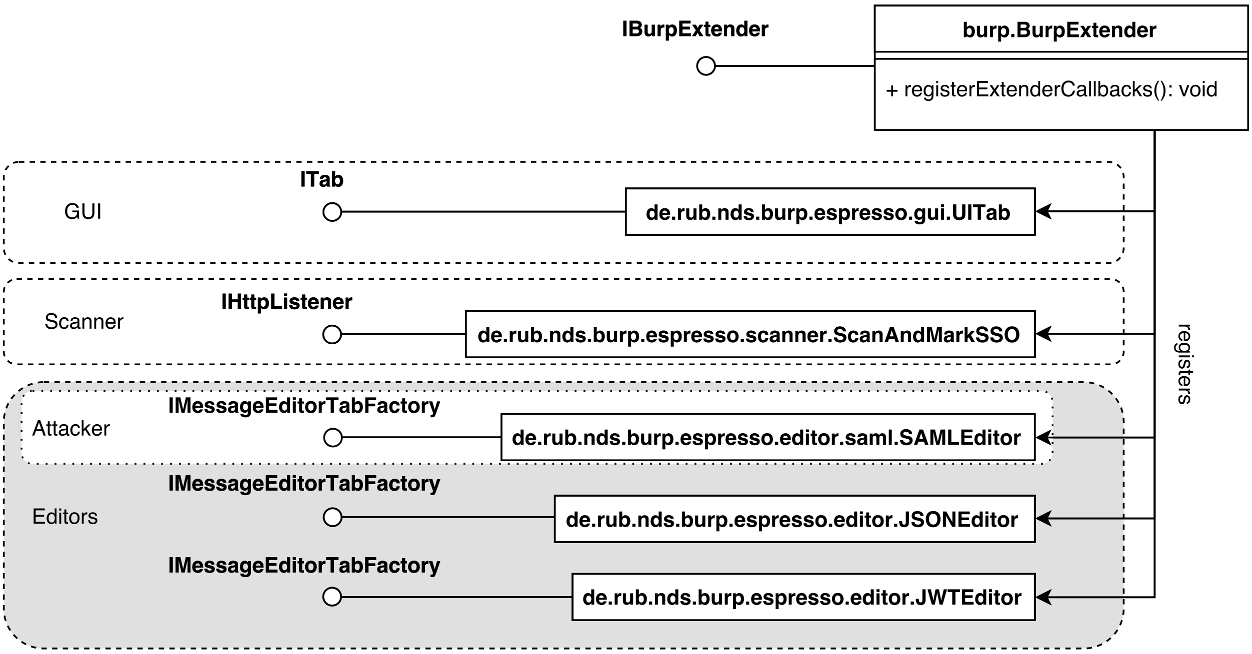

Figure 4.4.: The registered interfaces for the Burp Suite API

4.4.1. User Interface

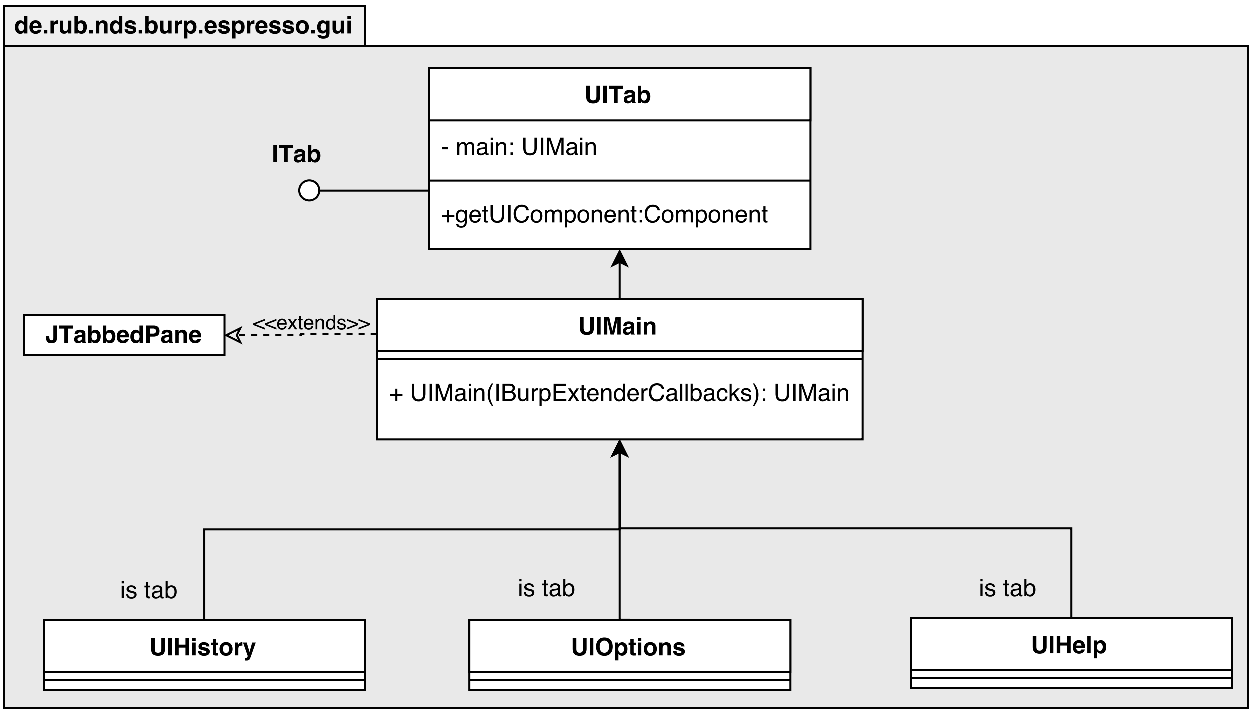

Figure 4.5.: The abstract User Interface setup.

The User Interface (UI) components are placed in the package `de.rub.nds.espresso.gui`, as shown in Figure 4.5. The Burp Suite interface for a new tab is implemented in `UITab`. The `getUIComponent()` function returns the `UIMain` object which is in the process attached next to Burp Suite's regular tabs. In `UIMain`, which extends the `java.swing` class `JTabbededPane`, the following classes `UIHistory`, `UIOptions` and `UIHelp` are added as new tabs.

SSO History

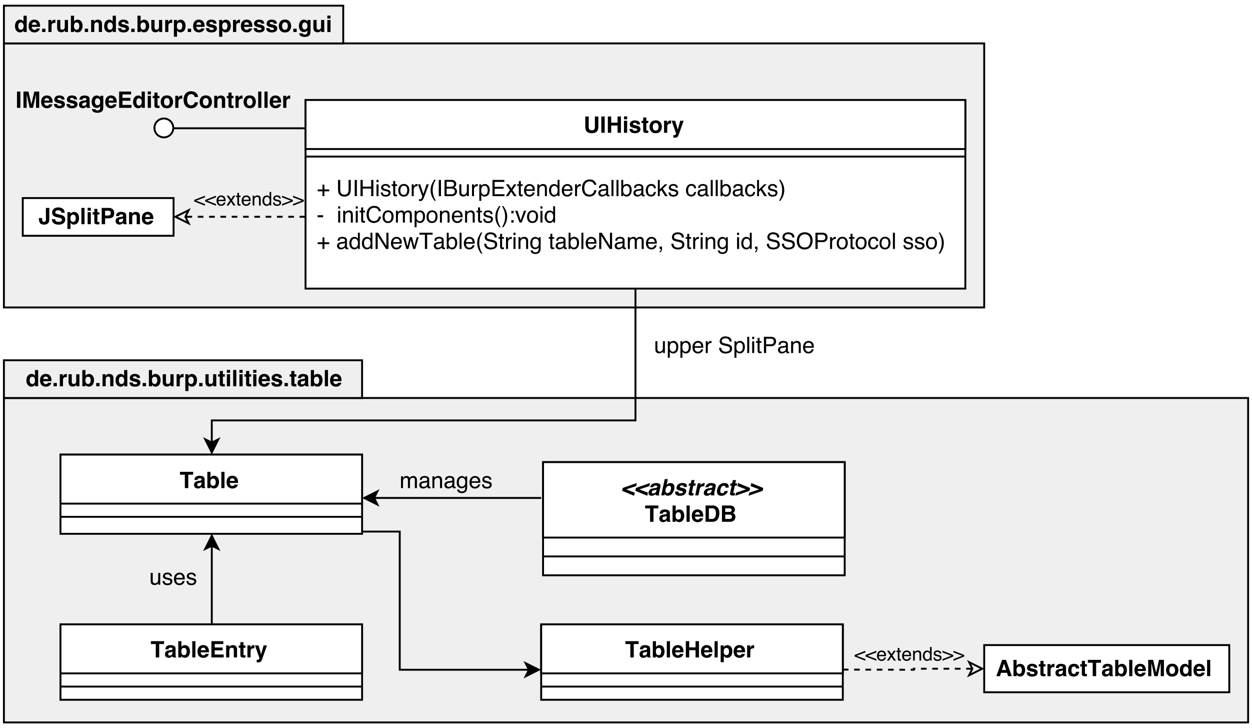

Figure 4.6.: The abstract UIHistory setup.

The `UIHistory` extends a JSplitPane, the top component of which is a table containing the SSO history. The bottom component is an `IMessageEditor` object, which is the same as the request/response viewer used in the rest of Burp Suite. The *Full History* is a customized `JTable` in the package `de.rub.nds.burp.utiliti` simply called `Table`. Each row is defined by the class `TableEntry`. The new table is created with a `TableHelper`, a caption for the new tab and an identifier. The `TableHelper` supplies operations for the table, as well as the table model information by extending the `AbstractTableModel`. The table layout can be seen in Figure 3.2. The class `TableDB` manages and stores all created tables. The basic operations are '*get a table*' by its index or id or '*remove a specific table*' from the storage.

A context menu can be opened by right clicking on a table row. This menu offers three operations on the `TableEntry`:

- **Analyse SSO Protocol:**[1] This operation starts the analysis of the protocols to find matching protocol

[1]This function is a feature but is unstable at the time of writing.

messages for the selected entry. At the time of writing the algorithm analyzes the protocol flow during the recognition phase.

- **Add Selected to Table:** This operation will move a selected entries to a table/protocol flow, but is not implemented at the time of writing.
- **Clear History:** This operation will remove all entries from the table to clear the view, but is not implemented at the time of writing.

The `TableMouseListener` is registered on the table it recognize these events.

Options

`UIOptions` extends `java.swing.JPanel`. The logic is straight forward: `public static` getters are used to control the functions outside the `UIOptions` class. The disabled components are not used in the version of the thesis but will be enabled in a later version.

Help

The `UIHelp` class contains no logic, therefore it is not explained here. Please refer the source code for more information in Appendix A.

4.4.2. Scanner

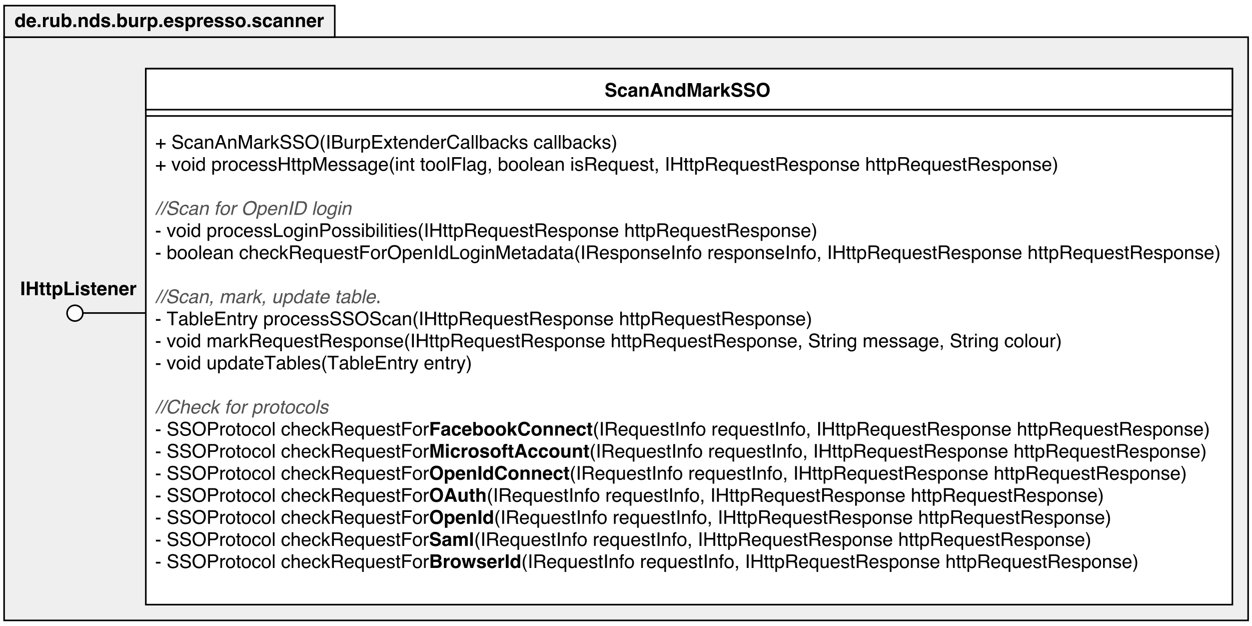

Figure 4.7.: The Scanner and important methods.

The Scanner is implemented in the class `de.rub.nds.burp.espresso.scanner.ScanAndMarkSSO`, demonstrated in Figure 4.7. It is the heart of the application and implements the `IHttpListener` from Burp Suite. This interface is called each time a request or response is processed by Burp Suite. Burp Suite

calls the method `processHttpMessage(int toolFlag, boolean isRequest, IHttpRequestResponse httpRequestResponse)` for each received message. Within this method the `toolFlag` represents a specific Burp Suite tool like the proxy, as an integer. The boolean `isRequest` states, if the message is a request (`true`) or a response (`false`). The object of the interface `IHttpRequestResponse` is Burp Suite's data structure to pass HTTP messages around. If a message is sent by the *Proxy Tool* and the corresponding response is received, the `IHttpRequestResponse` object is then passed to the function `processSSOScan()`. This is important as if the method does not wait for the response, then the `IHttpRequestResponse` object will be incomplete.
The method `proccessSSOScan()` includes the calls to all protocol check functions, which are discussed more precisely in the section 4.4.2 (*Check SSO Protocols*).
After a protocol is detected the HTTP message is highlighted in yellow and commented with the recognized result by the method `markRequestResponse()`. The `SSOProtocol` returned by the check method is then converted to an `TableEntry` and propagated to via `updateTables()` to the Tables.
Aside from the Single Sign-On recognition, the automatic recognition of *OpenID* logins and metadata is done by the method `checkRequestForOpenIdLoginMetadata()`, which is called by `processHttpMessage()` with `proccessLoginPossibilities()`.

SSO Protocols

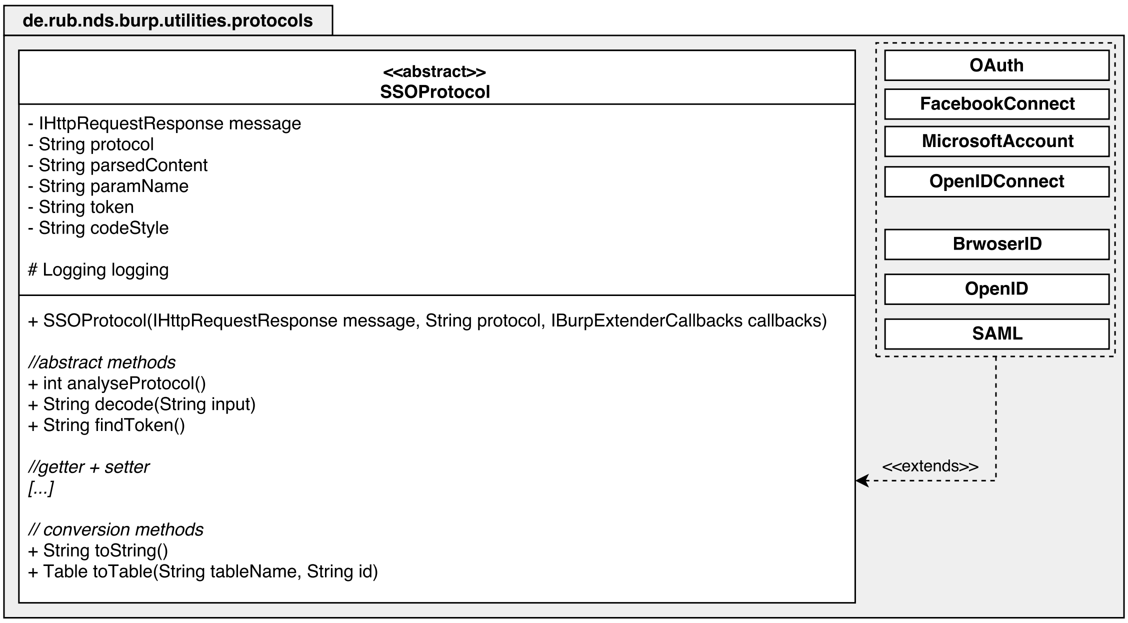

Figure 4.8.: The SSO Protocol classes.

`SSOProtocol` is a class to gather the common features needed to store the protocols. In order to implement protocol specific behaviors and provide the same interface three methods need to be implemented by the classes extending the `SSOProtocol`. The first, `analyseProtocol()`, should implement a method to analyze the protocol flow when the user hits '*Analyse SSO Protocol*', as demonstrated in Figure 3.3. The method `decode()` should provide a way to decode encoded input such as SAML parameters. The operation `findToken()` searches for a protocol specific identifier. For example, in SAML it is the `ID` or `InResponseTo` value within the XML message. Besides the get and set methods for the private variables, conversion operations like `toString()` and `toTable()` are available. Every supported protocol is represented through its extension and implementation of this class.

Check SSO Protocols

In the following section the `checkRequestFor`-methods are discussed, and the structure for the detection process for each protocol is presented. The implementation is based on the paper Mainka et al.[12]. The order of the protocols, presented below, is the same as in the implementation and is important due to the similarities between the OAuth-family protocols.

Facebook Connect
For Facebook Connect the message host must contain `facebook.com`. Afterwards the message URL is checked for `/ping?` and marked as *Facebook Connect Ping Request*, if this characteristic is present. Next the message is checked for the parameters `app_id`, `domain`, `origin` and/or `sdk`. If one or more of these parameters appear the message is classified as `Facebook Connect`, otherwise the `checkRequestForOauth()` method is called to check for OAuth specific behaviors for Facebook Connect. If the parameter `signed_request` is identified the message is marked as *Facebook Connect Authentication Response*. If the parameter `response_type` contains the value `signed_request` the message is marked as *Facebook Connect Authentication Request*.

Microsoft Account
For Microsoft Account the message host must contain `live.com`, `live.net` or `contoso.com`. The message is then identified as *Micosoft Account with OAuth* if the `scope` parameter is present and contains `wl.basic`, `wl.offline_access` or `wl.signin`. If `scope` contains `openid`, then the `checkRequestForOpenIDCOnnect()` method is called and as long as the returned value is not `null` the message is marked as *Microsoft Account*. If the message is still not classified, and the parameter `wa` is present with the value `wsignin1.0`, the message is marked as *Microsoft Account WS-Federation*. Otherwise the message is checked with `checkRequestForOAuth()` to ensure that no OAuth related message has been missed, and if the method returns a value other than `null`, the message is marked as *Microsoft Account*.
Note that the `checkForRequest`-methods will mark the message again.

OpenID Connect

The OpenID Connect check method is divided into two parts. The first part deals with the detection of parameters with an preceding redirect message (HTTP Status-Code 302), and the second part deals with the detection of the other parameters.

1. Preceding 302 messages and 200 messages are split as follows:

   a) *Hybrid Flow*: If the parameter `response_type` contains the value `code` and the value `id_token` or `token`, the message is marked as *OpenID Connect Hybrid Flow*. This also indirectly matches the cases of the permutation of all three values with the mandatory value `code`.

   b) *Authorization Code Flow*: If the message has the HTTP status code 302 and the response location matches the regular expressions `^Location:.*?response_type=code.*?$` and `^Location:.*?&?scope=[a-zA-Z+]*?openid[a-zA-Z+]*?&?.*?$`, the message is marked as *OpenID Connect ACF Request*. If the next message contains the parameter `response_type=code` and `scope=openid`, the message is also marked as *OpenID Connect ACF Request*.

   c) *Implicit Flow*: If the parameter `response_type=id_token` exists in a message with a 302 status-code, the message is marked as *OpenID Connect Implicit Flow Request*. If the next message has a parameter named `id_token` the message is marked as *OpenID Connect Implicit Flow Response*. If the following message contains the parameter `access_token`, the message is marked as *OpenID Connect Implicit Flow Access Token*.

2. The other parameters are split as follows:

   a) *Discovery Flow*: If the regular expression
   `"\\/\\.well-known\\/openid-configuration|\\/\\.well-known\\/webfinger"`
   matches, the message is classified as *OpenID Connect Discovery Flow*.

   b) *Generic Detection*: If the parameters `code` and/or `state` are present together with the parameter `scope`, the message is marked as *OpenID Connect / OAuth*. In addition, if the value of `scope` is `openid`, the message is only marked as *OpenID Connect*.

OAuth

The message is checked at the beginning of the OAuth detection for the parameters `redirect_uri`, `scope`, `client_id`, `client_secret` and `response_type`. If one or more of the parameters is present the algorithm proceeds. The checks are then divided into three parts:

1. *Authorization Code Flow*: If the parameters `grant_type` or `response_type` are present, the algorithm proceeds. If the previous message had status-code 302 and the current message contains the parameter `response_type` with the value `code`, the message is marked as *OAuth ACG Request*. If the parameter `code` exists, the message is classified as *OAuth ACG Code*. If the parameter `grant_type` exists with the value `auth_code`, the message is marked as *OAuth ACG Token Request*.

2. *Implicit Code Flow*: If the parameters `access_token` or `response_type` are present, the algorithm proceeds. If the previous message had the status-code 302 and the current message contains the parameter `response_type` with the value `token`, the message is marked as *OAuth Implicit Grant Request*. If the response of the message matches the regular expression `Location:.*?#.*?access_token=.*?&?`, the message is marked as *OAuth Implicit Token*, otherwise it is marked as *OAuth (IF)*.

3. *Other Flows*:
   The parameter `grant_type` is evaluated for the following values and marked as:
   - *OAuth Access Token Request* with the value `authorization_code`.
   - *OAuth Refresh Token Request* with the value `refresh_token`.
   - *OAuth Resource Owner Password Credentials Grant* with the value `password`.
   - *OAuth Client Credentials Grant* with the value `client_credentials`.
   - *OAuth Extension JWT Grant* with the value `urn:ietf:params:oauth:grant-type:jwt-bearer`.
   - *OAuth Extension SAML Grant* with the value `urn:oasis:names:tc:SAML:2.0:cm:bearer`.

If none of the three categories matches the message is marked as *OAuth*, through the generic detection.

OpenID

A message is classified as OpenID, if the parameter `openid.mode` contains the value `checkid_setup` (*OpenID Request*) or `id_res`. If the `openid.mode` contains the value `associate`, then the message is an *OpenID Association*. To identify the message as *OpenID 2.0 Token*, the parameters `openid.sig` and `openid.claimed_id` must be present. If the parameter `openid.claimed_id` is not present, the messages id marked as *OpenID 1.0 Token*.

SAML

A message is classified as SAML, if the parameter `SAMLRequest` (*SAML Authentication Request*) or `SAMLResponse` (*SAML Response Token*)is present.

BrowserID

The message host musst contain `persona.org`. The message is then searched for the parameters `assertion` and `browserid_state` and then classified as *BrowserID* if they are present.

4.4.3. Editors

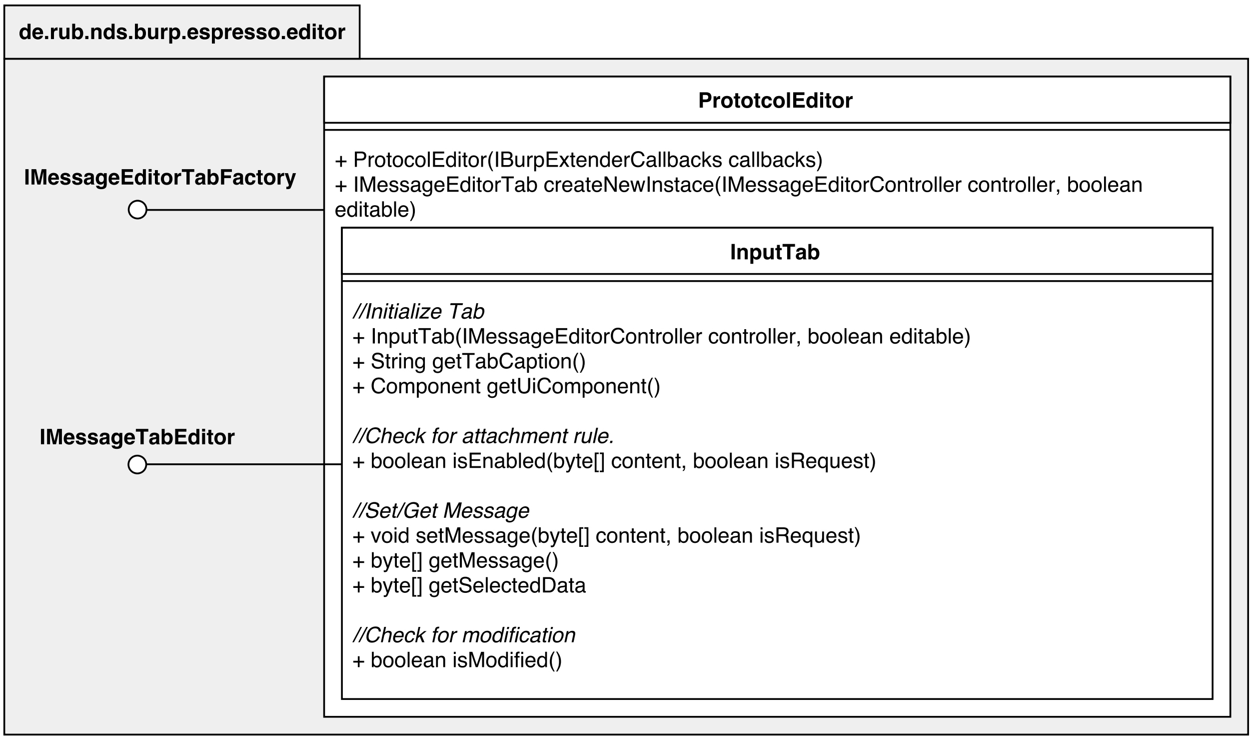

Figure 4.9.: A template message editor with all necessary methods.

Editors are new custom tabs attached to the request/response viewer. The tabs are registered with each of the six standard request/response viewers as well as with each of the new ones created by this extension. In order to supply each of the viewers with a new tab an `IMessageEditorTabFactory` must be implemented. The factory creates new objects of the interface `IMessageEditorTab` using the createNewInstance() method.

Figure 4.9 describes a template containing all of the necessary methods. The `IMessageEditorTab` implementation is an inner class of the factory, because in our case we do not reuse the tab elsewhere. The GUI components are initialized in the construction of the `InputTab`, the method `getTabCaption()` returns the headline of the tab, and the `getUiComponent()` allows Burp Suite to retrieve the UI to attach. An attachment rule can be defined by the content of the request/response to control the display of the tab. For example, the `ProtocolEditor` should only be attached if a specific protocol is in the request. Basic getter and setter are defined to display and obtain the whole message or only a selected part. If the method `isModified()` returns `true` Burp Suite will automatically display two tabs after the HTTP message is stored, one for the original message and one for the modified message.

JSON Editor

The setup of the `JSONEditor` is similar to the template, the SAML Editor. The attachment rule analyzes the given HTTP message for a JSON content type. If JSON is detected a tab called 'JSON' is added to the request/response viewer, as shown in Figure 3.7. The sub-tabs 'JSON Viewer' and 'Raw' display the decoded JSON. The viewer automatically indents and syntax highlights the object with the support of the `RSyntaxTextArea` integrated in the `UISourceViewer` class. The 'Raw'-tab simply displays the unmodified, decoded JSON in Burp Suite's own editor, implemented in `UIRawEditor`.

JWT Editor

The `JWTEditor` is almost identical to the `JSONEditor`. The only difference is the decoding and attachment rule. The attachment rule searches for specific parameters of SSO protocols known for JWT. At time of writing the editor searches for `assertion`, `id_token` and `access_token`. Due to this design limitation, the editor is, in contrast to the `JSONEditor`, not capable of decoding random JWTs. The decoder uses the format of three Base64 strings concatenated with dots. After the split and Base64 decoding the messages are displayed in the three tabs representing their value (see Figure 3.8).

SAML Editor

The `SAMLEditor` differs from the other editors as the *Attacker*'s attachment rules are also implemented. The details on the *Attacker* are discussed in the subsection 4.4.4. What makes this editor unique is that the de- and encoding function depends on the parameter identified. The searched for parameters are `SAMLRequest` and `SAMLResponse`. While `SAMLResponse` is encoded as a URL encoded Base64 string, `SAMLRequest` is redirect format encoded. This means that the XML data is deflated (compressed) and then Base64 and URL encoded. The same applies to the decoding process.

4.4.4. Attacker

This subsection explains the setup and structure of the Attacker, starting with the listeners and continuing with the internal logic. The Attacker is part of the SAML Editor and is attached only if the the `IHttpRequestRespones` is processed by the interceptor tool. This is the case whenever the variable `editable` is `true`, and the message is therefore editable.

Code Listener

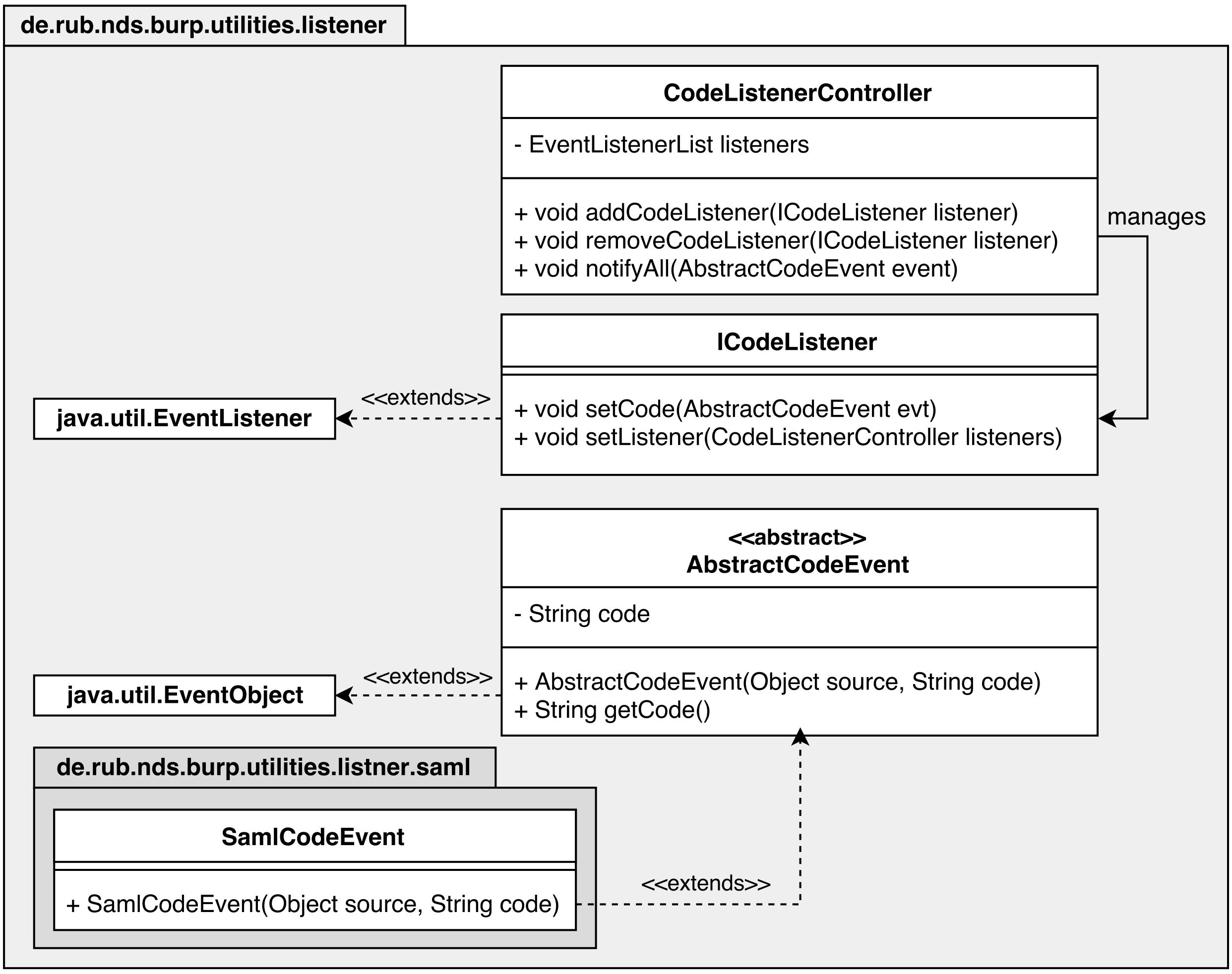

Figure 4.10.: The listeners for code change events.

The `listener` package, shown in Figure 4.10, provides an implementation of the observer design pattern. The classes are intended to manage the notification of changes in the code when a modification should be applied to the message. The `ICodeListener` is the interface implemented by the classes requests for notification. At the moment the classes `UISourceViewer` and `UIRawEditor` implement this interface. The listener retrieves the new modified code using `setCode(AbstractCodeEvent evt)`. The `SamlCodeEvent` is the extension of `AbstractCodeEvent` and stores the code internally. The

CodeListenerController manages the listeners and their notification, with registration, removal and notification to all listeners. Please not that the methods of CodeListenerController are explicitly not static because the events would trigger all interface implementations, not just the one in the interceptor.

Attacker Business Logic

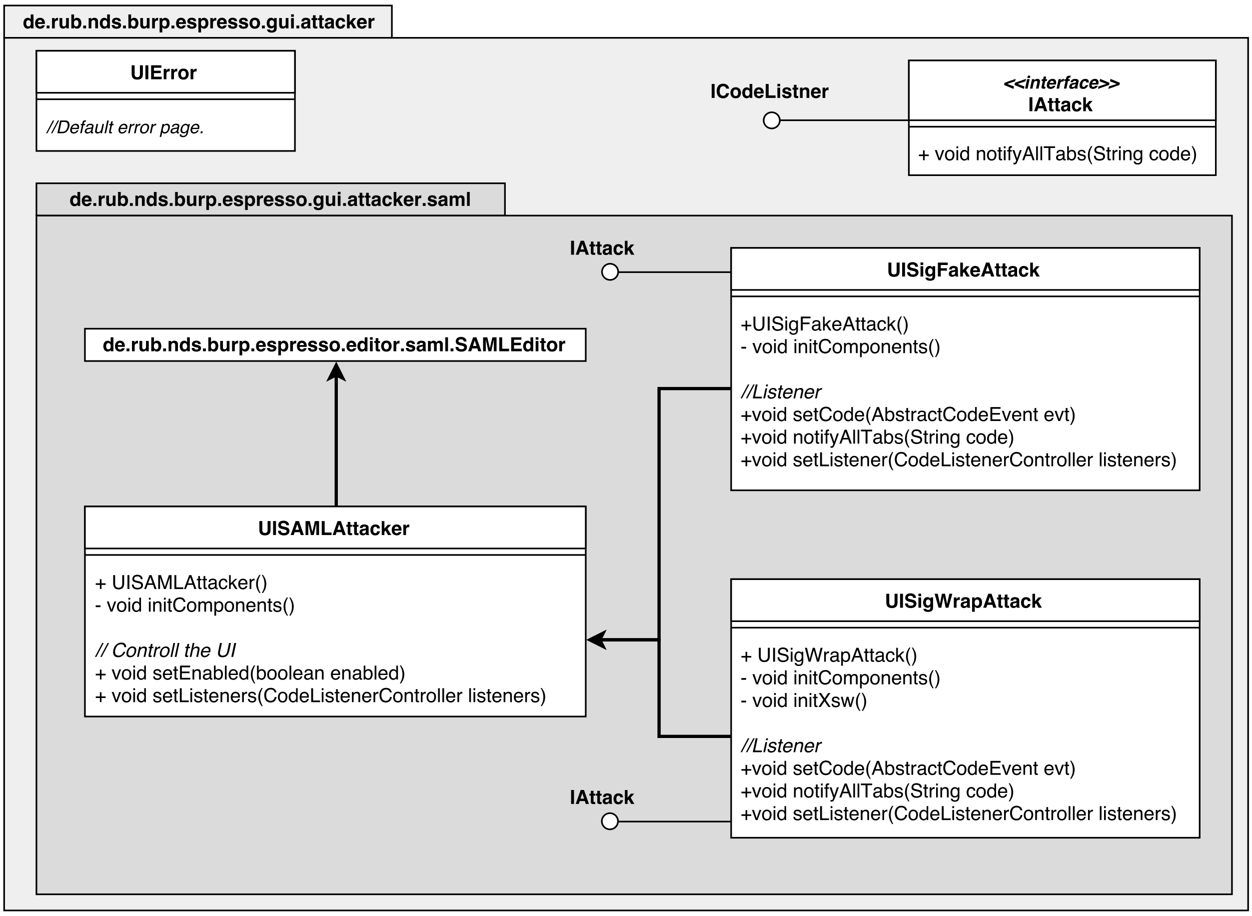

Figure 4.11.: The Attacker internal structure.

The Attacker is controlled by the class UISAMLAttacker, as shown in Figure 4.11. The components of the UI are controlled using UISAMLAttacker. The enabled status is controlled with the overridden method setEnabled(), which is important if the interceptor is disabled, otherwise UI methods can be used. The two java.swing.JPanels UISigFakeAttack and UISigWrapAttack are integrated using a java.awt.CardLayout, and both classes are implementing the interface IAttack.

UISigFakeAttack

The class UISigFakeAttack implements a *XML Signature Faking* attack based on the WS-Attacker [1] Signature Faking library. To fake a given XML signature no further configuration is needed. The following listing shows the usage of the library and the `SignatureFakingOracle`.

```
try {
        SignatureFakingOracle sof = new SignatureFakingOracle(code);
        sof.fakeSignatures();

        String fakedSignatureXML = sof.getDocument();
} catch (SignatureFakingException ex) {
        //Do some exception handling!
}
```

Initiate the `SignatureFakingOracle` with a XML string (here `code`) and generate a new String with a faked signature with `sof.fakeSignatures()`. The string with the attack vector is called with `sof.getDocument()`.

UISigWrapAttack

The class `UISigWrapAttack` implements a *XML Signature Wrapping* attack based on the WS-Attacker [1] Signature Wrapping library. More configuration is required compared to the *Signature Faking* attack. The following listing demonstrates the steps required to configure the library.

```
SchemaAnalyzer samlSchemaAnalyser =
                SchemaAnalyzerFactory.getInstance(SchemaAnalyzerFactory.SAML);
WrappingOracle wrappingOracle;
SignatureManager signatureManager;

Document doc;

//Init.
try {
        doc = DomUtilities.stringToDom(code);
        signatureManager = new SignatureManager();
        signatureManager.setDocument(doc);
} catch (SAXException ex) {
        //Do some exception handling!
}

//Generate the attack with the oracle
Document samlDoc = signatureManager.getDocument();
List<Payload> payloadList = signatureManager.getPayloads();
wrappingOracle = new WrappingOracle(samlDoc, payloadList, samlSchemaAnalyser);

//Choose attack and get modified XML
Document attackDoc = wrappingOracle.getPossibility(attack);
String attackString = DomUtilities.domToString(attackDoc);
```

First retrieve a SAML `SchemaAnalyzer` from `SchemaAnalyzerFactory` and declare all variables needed for `try-catch`-block, initialize the `SignatureManager` with the XML document (here `code`). Fetch the `Document` and the `Payload` list and create a new `WrappingOracle`. With this oracle and an integer representing an attack, the new generated `Document` can be used to attack the web service.

4.4.5. Logging

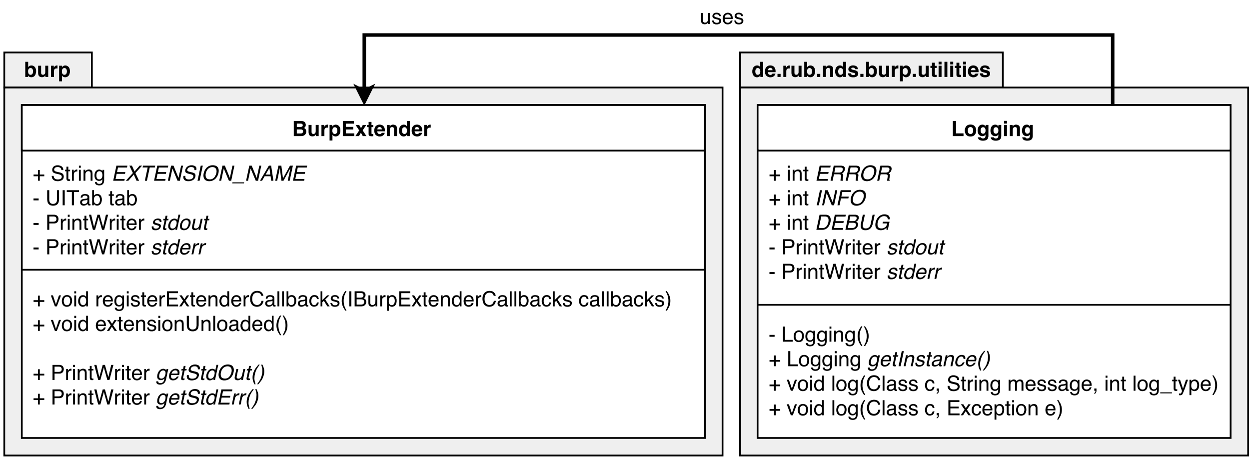

Figure 4.12.: The logging utility.

The `Logging` class integrates a better and more standardized way of logging into Burp Suite's own logging console. In order to write to Burp Suite's output and error console, two `PrintWriters` from Burp Suite's callbacks are needed. Therefore the static functions `getStdOut()` and `getStdErr()` are implemented in `BurpExtender`. These methods supply the `Logging` with the needed `PrintWriters`. With the singleton design pattern, there is only one instance of the class, which guarantees that the `Logging` will work without errors as long as the utility is not used before the `BurpExtender` has initialized the `PrintWriters`.

4.4.6. Utilities

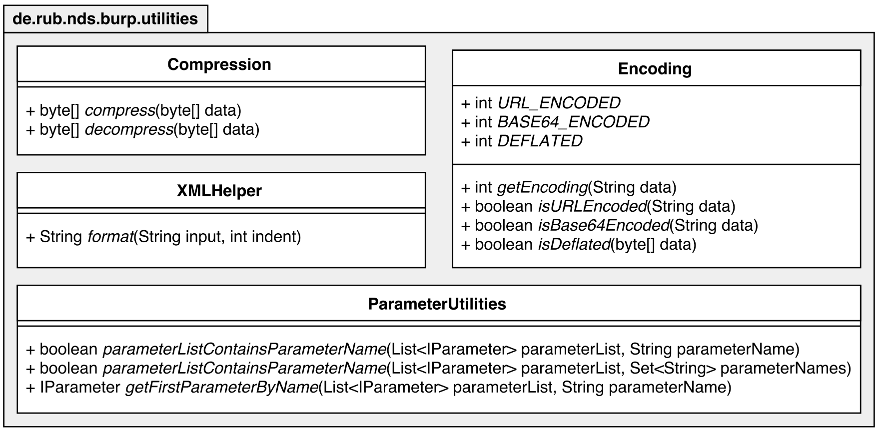

Figure 4.13.: The utilities package.

The following subsection gives a short description of the utilities in the package `de.rub.nds.burp.utilities`, which is shown in Figure 4.13.

Compression

The `Compression` utility is used for the de- and inflating of SAML messages. Therefore the class has a `compress()` and a `decompress()` method. The operations run on byte arrays and return modified byte arrays, and the algorithm used is zip (in)deflating.

Encoding

The `Encoding` class supplies operations to evaluate whether the input data is URL, Base64 encoded or deflated. The checks can be done on their own or with `getEncoding()` which returns an integer representing a specific encoding. The integer representing the encoding type can be compared with the static variables `URL_ENCODED`, `BASE64_ENCODED` and `DEFLATED`.

XMLHelper

The `XMLHelper` provides help to format XML messages with correct indenting. The method gets the XML String and the indention level and outputs the formatted content. If an error occurs, it is logged and the unmodified input is returned.

ParameterUtilities

The `ParameterUtilities` are a set of methods to retrieve HTTP parameters from a set of `burp.IParameters`. The operations evaluates if a parameter with a specific name or set of names are in the list of parameters or retrieves the parameter by its name.

4.5. Extensibility

During implementation the source code was designed to be extendable. The following chapter describes the ways to extend different components faster. To speed up the development the author recommends to import the documentation of `{projectpath}/doc/apidocs` into the Integrated Development Environment (IDE), this is not necessary if the project is directly modified.

4.5.1. Extend the GUI

To extend the Graphical User Interface (GUI) of EsPReSSO it is necessaryto add a new tab in `UIMain`, shown in Figure 4.5. This is done using the code listed in Figure 4.14.

Create a new `private` variable for your UI. Where `UINewPanel` is a new written class with the wanted User Interface.

```
private UINewPanel newPanel;
```

Add a new getter for your UI.

```
public UINewPanel getNewPanel(){
        return newPanel;
}
```

Add initiate and add the new panel as a tab in the `initComponents()` method.

```
newPanel = new UINewPanel();

//Add to the tab.
this.addTab("Tab Caption", newPanel);
```

Figure 4.14.: Add a new tab to the EsPReSSO GUI

If the extension should have a new tab, next to Burp Suite's regular tabs, create and register a new implementation of `burp.ITab` interface in the `burp.BurpExtender` class. The code to add a new tab is:

```
callbacks.addSuiteTab(new TheNewSuiteTabImplementation());
```

4.5.2. Extend a new Protocol in the Scanner

The following instructions guide through the process of adding a new Single Sign-On protocol to the scanner.

1. Extend the `SSOProtocol` with an new class in the package `de.rub.nds.burp.utilities.protocols`.
2. Implement all abstract methods.
3. Add a new method with the name `checkRequestForProtocolName()` to the `ScanAndMarkSSO` class.
4. Register the 'check'-method in the function `processSSOScan`.

```
if(UIOptions.isProtocolNameActive()){
        SSOProtocol protocol =
        checkRequestForProtocolName(requestInfo, httpRequestResponse);
        if(protocol != null){
                protocol.setCounter(counter++);
                return protocol.toTableEntry();
        }
}
```

Optional Add a checkbox in `UIOptions` to enable and disable the scanning.

4.5.3. Extend a new Attack in the Attacker

The following instructions provide a guide for the process of adding a new attack to the SAML Attacker.

1. Create a new class implementing the `IAttack` interface and extending a `java.swing.JPanel`.
2. In `UISAMLAttacker`:
   a) Create a `private` variable for your new class.
   b) Create a new `private final String ATTACK_NAME = 'Description Text'`.
   c) Extend the `String[] attackArray = {NO_ATTACK, SIGNATURE_FAKING, SIGNATURE_WRAPPING, ATTACK_NAME};` array with your new variable.
   d) Initiate your new class.
   e) Add the class to the settings container with `settingsContainer.add(uiNewAttackClass, ATTACK_NAME);`.
   f) Register a listener in `setListeners()` on the new class.

The following listings shows all steps at once.

UINewAttackClass.java

```
public class UINewAttackClass extends JPanel implements IAttack{
    private String code = null;
    private CodeListenerController listeners = null;

    @Override
    public void setCode(AbstractCodeEvent evt) {
                // Do your work!
    }

    @Override
    public void notifyAllTabs(String code) {
                // Do your work!
    }

    @Override
    public void setListener(CodeListenerController listeners) {
                // Do your work!
    }
}
```

UISAMLAttacker.java

```
public class UISAMLAttacker extends JPanel implements ItemListener{
        [...]
        private final String ATTACK\_NAME = "Description Text";
        [...]
        private UINewAttackClass uiNewAttackClass = null;
        [...]
        private void initComponents() {
                [...]
                String[] attackArray =
                {NO_ATTACK, SIGNATURE_FAKING, SIGNATURE_WRAPPING, ATTACK_NAME};
                [...]
                uiNewAttackClass = new UINewAttackClass();
                settingsContainer.add(uiNewAttackClass, ATTACK\_NAME);
                [...]
        }
        [...]
        public void setListeners(CodeListenerController listeners){
        [...]
        uiNewAttackClass.setListener(listeners);
    }
}
```

4.6. Evaluation

This section outlines the sites and protocols already tested using EsPReSSO to evaluate the implementation. Each subsection contains a list of sites tested and a brief description of faults. The colored circles as a traffic light system, ● the recognition is tested and worked, ● the implementation is tested but has still issues, ● a protocol characteristic appeared but the implementation failed to recognize it, and ● the implementation could not be tested due to the lack of the right test site or parameters.
All messages sent during the login process were manually evaluated and searched for the characteristic parameters.

Facebook Connect

- ● `http://cloud.nds.rub.de:8042/loginJS.html`
  - ● *Ping Request* is correct detected with `/ping?`
  - ● *Authentication Request* is correct detected with parameter `response_type=signed_request`.
  - ● *Authentication Response*
  - ● *Generic Detection* is successful, all HTTP messages with Facebook Connect parameters are detected.
- ● `http://forum.golem.de/login.php`
  - ● *Ping Request*
  - ● *Authentication Request*
  - ● *Authentication Response*
  - ● *Generic Detection* is successful, all HTTP messages with Facebook Connect parameters are detected.
- ● `https://stackoverflow.com/users/login`
  - ● *Ping Request*
  - ● *Authentication Request*
  - ● *Authentication Response*
  - ● *Generic Detection* The one, successfully detected, message was recognized by the host name `facebook.com` only.
    The other from `stackoverflow.com` send massages are detected by the OpenID Connect method, as generic *OpenID Connect / OAuth*.

The Facebook Connect sites were all tested using a valid Facebook account.

Microsoft Account

- http://forum.golem.de/login.php
  - *Microsoft Account with OAuth* is correct detected with scope=wl.baisc.
  - *Microsoft Account with WS-Federation* no matching message found.
- http://outlook.com
  - *Microsoft Account with OAuth* no matching message found.
  - *Microsoft Account with WS-Federation* is correct detected with wa=wsignin1.0.

The Microsoft Account sites were all tested using a valid live.de email address.

OpenID Connect

- https://demo.c2id.com/oidc-client/

  Select the *Response type* as code:
  - *OpenID Connect Authorization Code Flow Request* is detected correct.

  Select the *Response type* as id_token:
  - *OpenID Connect Implicit Flow Request* is detected correct.

  Select the *Response type* as code id_token:
  - *OpenID Connect Hybrid Flow* is detected with both parameters.
  - *OpenID Connect Generic* All messages that contain OpenID Connect parameters were correct detected.
  - *OpenID Connect Discovery Flow*
  - *OpenID Connect Implicit Flow Response*
  - *OpenID Connect Implicit Flow Access Token*

- The implementation is not capable of detecting parameters if they were transmitted as JSON.

OAuth

- https://developers.google.com/oauthplayground/
  - *OAuth Authorization Code Grant Request*
  - *OAuth Authorization Code Grant Code*
  - *OAuth Authorization Code Grant Token Request*
  - *OAuth Implicit Grant Request*
  - *OAuth Implicit Token*
  - *Other OAuth Flows*
  - *Generic OAuth Detection* has many false positives, but most OAuth messages where detected. Messages only containing code are not detected, as expected.

  The *OAuth Playground* site was tested with the *Blogger API v3* (Step 1) and the https://www.googleapis.com/auth/blogger option enabled. First click on *Exchange authorization code from tokens* (Step 2) and finally click *Send the request* (Step 3).

- The implementation is not capable of detecting parameters if they were transmitted as JSON.

OpenID

- `http://cloud.nds.rub.de:7051/consumer-servlet/index.jsp`
  - *OpenID Request* classification for parameter `openid.mode=checkid_setup` is correct.
  - *OpenID Token 1.0* classification for parameter `openid.sig` and `openid.return_to` is correct.
  - *OpenID Login Possibility* classification as false positive during the *OpenID Token 1.0*.
- `http://forum.golem.de/login.php`
  - *OpenID Request* classification for parameter `openid.mode=checkid_setup` is correct.
  - *OpenID Token 1.0* classification for parameter `openid.sig` and `openid.return_to` is correct.

Test also correct for the following sites:

- `https://stackoverflow.com/users/login`
- `http://csscreator.com`

The OpenID sites were all tested using a valid *Blogspot*[2] account. The option '*login only for this session*' is used, and the test concluded at the account configuration page.

- *OpenID 2.0 Token*
- *OpenID Associate*

SAML

- `http://cloud.nds.rub.de:7051/sp/index.html`
- `http://cloud.nds.rub.de:7023/idp/localauth/index.html` [3]

The SAML evaluation is straightforward, the parameters that are looked for are `SAMLRequest` and `SAMLReponse`. The decoding is tested with the output in the SAML tab and the token manually compared with the one in the parameter.

BrowserID

- `https://login.persona.org/`
- `https://www.voo.st/`
- `https://developer.mozilla.org/en/Persona/Quick_Setup`

All BrowserID sites were tested with a valid *Google Mail* account. After the press on the login with *Persona* the email address was entered and the option to log in for one session was chosen. The tests were concluded at the configuration page for the account on the SP. The traffic with other IdPs than `persona.org` is not analyzed in this test. All messages with the parameter `browserid_state` were detected. Therefore the token, which is the mentioned parameter, is identified correctly. The JSON and the JWT for the parameter `assertion` are correctly decoded and detected.

[2] A Google product also known as Blogger

[3] The request needs an additional parameter, it is included in the pdf version as a `href`.

4.7. Limitations

A limiting factor is Burp Suite itself because, with the exception of the public API, the source of Burp Suite remains private. This leads to the problem of testing the implementation with unit tests like *JUnit*. A test driven implementation would define the expected behavior at the beginning and test if the designed software matches the expectations. As seen in the previous section, the evaluation is incomplete. An enormous amount of effort is required to find all presented cases and analyze them manually. In the current implementation it is impossible to write a unit test without heavy mocking or implementation of Burp Suite's API, because the implementation of EsPReSSO bases heavily on the Burp Suite API.
The second limitation are the similarities between some protocol messages which made them hard to distinguish.

5. Conclusion

EsPReSSO is the first attempt to create a SSO Protocol analyzer with the capability to attack SAML. Using the concepts of automatic identification and classification, as well as a visualization of the recognized protocols, EsPReSSO will hopefully be a good contribution to the analysis of Single Sign-On technologies. The easy extensible source code will guarantee the potential for future development of EsPReSSO. As seen in the evaluation the detection of the individual protocols, despite the good results of some protocols, was not flawless. In particular it would require too much effort to test all special cases by hand, and be too inaccurate.
Therefore, future work on EsPReSSO would definitely benefit from a re-implementation, decoupled from the Burp Suite API, of the Scanner and its `checkRequestForXYZ()` methods. A *JUnit* test to ensure the correct behavior of the implementation would be a good starting point. Furthermore a side-by-side `diff` to compare the original with the modified message after the attack would be another great enhancement. Burp Suite's *Comparer* does not full-fill the need for further analyses. Moreover new features such as context menu entries, for example *'Add as SSO Protocol'* on right click in the Burp Suite Proxy, to enable manual add of protocols to SSO History of EsPReSSO or *'Change Protocol'* on right click in EsPReSSO's History to change a wrong detected protocol manually, would help to simplify EsPReSSO. A feature to store the results and recover already stored data would helpful for longer investigations.

A. Appendix

A.1. Source Code of EsPReSSO

The source code will available at `https://github.com/RUB-NDS/BurpSSOExtension` and is attached on the CD in the printed version.

A.2. License

The license is the GNU General Public License v2.0 as it published at `http://www.gnu.org/licenses/gpl-2.0.txt`.

GNU General Public License v2.0 - Short Form

A.3. Dependencies

All dependencies, licenses and copyrights used by this extension.

Dependencies		Access Date	Link	Copyright (c) Date, Name
RSyntaxTextArea	modified BSD license	20.09.2015	`https://github.com/bobbylight/RSyntaxTextArea/`	2012, Robert Futrell
json-simple	Apache License 2.0	20.09.2015	`https://code.google.com/p/json-simple/`	Unkown, Yidong Fang
WS-Attacker	GNU General Public License v2.0	20.09.2015	`https://github.com/RUB-NDS/WS-Attacker/`	2012, Mainka, Falkenberg, et al.

Table A.1.: Dependencies, licenses and copyrights

Bibliography

[1] Chair for Network and Data Security, Ruhr-University Bochum, "WS-Atacker," Accessed: 05.07.2015. [Online]. Available: https://github.com/RUB-NDS/WS-Attacker

[2] CERN Computer Security, "Password Recommendations," Accessed: 04.10.2015. [Online]. Available: https://security.web.cern.ch/security/recommendations/en/passwords.shtml

[3] Android Developers, "Accessing Resources," Accessed: 04.10.2015. [Online]. Available: http://developer.android.com/guide/topics/resources/accessing-resources.html

[4] ECMA International, "ECMA-376: Office Open XML File Formats," Accessed: 04.10.2015. [Online]. Available: http://www.ecma-international.org/publications/standards/Ecma-376.htm

[5] C. M. S.-M. e. a. Tim Bray, Jean Paoli, *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, W3C REC -20 081 126, Nov. 2008. [Online]. Available: http://www.w3.org/TR/2008/REC-xml-20081126/

[6] *The OAuth 2.0 Authorization Framework*, ECMA International ECMA -262, June 2015. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[7] N. S. M. Jones, J. Bradley, *JSON Web Token (JWT)*, IETF RFC 7519, May 2015. [Online]. Available: https://tools.ietf.org/html/rfc7519

[8] J. H. N. Sakimura, *JSON Web Encryption (JWE)*, IETF RFC 7516, May 2015. [Online]. Available: https://tools.ietf.org/html/rfc7516

[9] N. S. M. Jones, J. Bradley, *JSON Web Signature (JWS)*, IETF RFC 7515, May 2015. [Online]. Available: https://tools.ietf.org/html/rfc7515

[10] S. Josefsson, *The Base16, Base32, and Base64 Data Encodings*, IETF RFC 4648, Oct. 2006. [Online]. Available: https://tools.ietf.org/html/rfc4648

[11] NIST, *HMAC: Keyed-Hashing for Message Authentication*, FIPS PUB 198-1, July 2008. [Online]. Available: http://csrc.nist.gov/publications/fips/fips198-1/FIPS-198-1_final.pdf

[12] Christain Mainka, Vladislav Mladenov, Tim Guenther, Jörg Schwenk, "Automatic Recognition, Processing and Attacking of Single Sign-On Protocols with Burp Suite," 2015.

[13] B. F. D. Recordon, *OpenID Authentication 1.1*, OpenID Foundation OpenID 1.1, May 2006. [Online]. Available: http://openid.net/specs/openid-authentication-1_1.html

[14] *OpenID Authentication 2.0 - Final*, OpenID Foundation OpenID 2.0, Dec. 2007. [Online]. Available: http://openid.net/specs/openid-authentication-2_0.html

[15] H. Thiel, "Praktische Sicherheitsanalyse des Mozilla Single Sign-on Protokolls BrowserID," Bochum, Germany, 2014, Accessed: 22.09.2015. [Online]. Available: https://www.nds.rub.de/media/ei/arbeiten/2014/12/04/BrowerID.pdf

[16] R. P. e. a. Scott Cantor, John Kemp, "Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0," Mar. 2005. [Online]. Available: http://docs.oasis-open.org/security/saml/v2.0/

[17] E. Hammer-Lahav, *The OAuth 1.0 Protocol*, IETF RFC 5849, Apr. 2010. [Online]. Available: https://tools.ietf.org/html/rfc5849

[18] E. D. Hardt, *The OAuth 2.0 Authorization Framework*, IETF RFC 6749, Oct. 2012. [Online]. Available: https://tools.ietf.org/html/rfc6749

[19] C. Nickel, "Sicherheitsanalyse von OAuth 2.0 mittels Web Angriffen auf bestehende Implementierungen," Master's thesis, Lehrstuhl für Netz- und Datensicherheit, Ruhr-University Bochum, Bochum, Germany, 2013, Accessed: 22.09.2015. [Online]. Available: https://www.nds.rub.de/media/ei/arbeiten/2014/12/04/OAuth_Security.pdf

[20] e. a. N. Sakimura, J. Bradley, *OpenID Connect Core 1.0 incorporating errata set 1*, OpenID Foundation OpenID Connect 1.0, Nov. 2014. [Online]. Available: http://openid.net/specs/openid-connect-core-1_0.html

[21] J. Krautwald, "OpenID Connect," September 2014.

[22] Facebook Inc., "Facebook Login for Apps," Accessed: 22.09.2015. [Online]. Available: https://developers.facebook.com/docs/facebook-login/overview

[23] J. Rzeniewicz, "Log Me In with Facebook: Security Analysis of Facebook Connect," July 2015.

[24] Microsoft, "Microsoft Account," Accessed: 22.09.2015. [Online]. Available: https://account.microsoft.com/about

[25] Roland Bischofberger, Emanuel Duss, "SAML Raider - SAML2 Burp Extension," Accessed: 01.10.2015. [Online]. Available: https://github.com/SAMLRaider/SAMLRaider

[26] John Barber, "SAMLyze," Accessed: 01.10.2015. [Online]. Available: https://www.blackhat.com/us-15/arsenal.html#samlyze

[27] Mainka, Christian and Mladenov, Vladislav and Feldmann, Florian and Krautwald, Julian and Schwenk, Jörg, "Your Software at my Service," 2014.

[28] S. Gajek, M. Jensen, L. Liao, and J. Schwenk, "Analysis of signature wrapping attacks and countermeasures," in *Web Services, 2009. ICWS 2009. IEEE International Conference on*. IEEE, 2009, pp. 575–582.

[29] PortSwigger, "Burp Extension API," Accessed: 05.07.2015. [Online]. Available: http://portswigger.net/Burp/extender/api/index.html

[30] ——, "Burp Support," Accessed: 05.07.2015. [Online]. Available: https://support.portswigger.net/