Bachelor Thesis

Implementing a Modular Framework for Web Services Penetration Testing

Ruhr-Universität Bochum

Christian Mainka

Matr.-Nr:108007212667

24. November 2010

Lehrstuhl für Netz- und Datensicherheit
Ruhr-Universität Bochum
Universitätsstr. 150
D-44789 Bochum

Supervision: Prof. Dr.-Ing Jörg Schwenk
Lehrstuhl für Netz- und Datensicherheit, Ruhr-Universität Bochum

Prof. Dr.-Ing. Dipl.-Wirtsch.-Ing. York Tüchelmann
Arbeitsgruppe Integrierte Informationssysteme

Abstract

Web services are everywhere – and everyone needs them.

However, they have hardly been analyzed with reference to security so far. This thesis will introduce WS-Attacker as a modular framework for web service penetration testing. It is a a free and easy to use software solution, which provides an all-in-one security checking interface with only a few clicks.

Contents

1 Introduction

Web services become more and more popular in the modern world. Companies like Google and Amazon provide an API for their services so that everyone can use them within their own applications. A new market for these services are smart-phone applications, e.g. a mobile navigator can use Google maps to find the nearest restaurant or train station. For doing this, it is not necessary to navigate to the Google site, nor to parse HTML. The application simply has to create a SOAP request, send it to the server and evaluate the response. This response uses the power of XML to structure the information so that data processing is smart. There are also a lot of web sites which use the benefits of web services – e.g. an online forum can use Google search to find replies in threads. Altogether, web services have a high potential, as they are easy to use and can be easily integrated into new applications. With the increasing popularity of web services, it is enormously important to test for security.

A web service is vulnerable to two different classes of attack. One of them is well-known, as these attacks also work on web applications, e.g. SQL Injection [1]. Nevertheless, the other class of attack is completely new: *web service specific* attacks abuse weaknesses in SOAP specifications, XML parsers and many more. Web services must be secure against all of those attacks. At the moment, there are lots of tools for penetration tests on web applications, but none of them can be used for penetration tests on web services.

This thesis will introduce the modular web service attacking framework *WS-Attacker*, which provides an easy to use GUI for automated penetration testing and a simple extendable interface for building attack plugins. It allows fast and easy testing for security issues on web services by using attack plugins. First, web services and their needed fundamentals are introduced in Section 2. A general overview of existing threats to web services is given in Section 3. As examples for web service specific attacks, SOAPAction Spoofing and WS-Addressing Spoofing are described.

Section 4 explains the WS-Attacker concept. Both, the framework itself and its plugin architecture are presented and as a proof of concept, an attack plugin is built for SOAP-Action Spoofing. The plugin is evaluated in Section 5. Finally, a concluding outlook and discussion about WS-Attacker are given in Section 6.

2 Basics

This section introduces the fundamentals to understand web services. Section 2.1 gives a short overview about XML, the markup language used for communication with web services. Section 2.2 introduces the idea of services-oriented architectures, which are the basis for web services that are explained in Section 2.3.

2.1 XML – Extensible Markup Language

The Extensible Markup Language (XML) is a set of rules to encode documents. It is a textual data format with Unicode support and is widely used for message exchange in the Internet.

2.1.1 XML in General

The *World Wide Web Consortium* (W3C) describes XML as the following [2]:

> *"Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere."*

In general XML is a data description and structuring language, which is designed for message exchange over the world wide web and has a large scope of application.

Example:

Consider someone wants to send a contact information to his friend, both using different application for managing this data. He could export the data to CSV[1] or Microsoft Excel format. Nevertheless, when his friend tries to import this data, there might be problems, because his application expects a different format, e.g. first name after last name or similar. The advantage of XML is, that it does not only structure data, but it also describes data.

[1] Comma Separated Values

Listing 1: XML Example for a person

```
1 <person>
2     <firstname>John</firstname>
3     <lastname>Doe</lastname>
4     <phone type="mobile">01701234567</phone>
5 </person>
```

Listing 1 gives an example of an XML formatted person named *John Doe*. Each value is enclosed by an opening (e.g. <firstname>) and a closing (e.g. </firstname>) tag, which forms the element. The element name describes the data type and let one distinguish between first name and last name[2], even if the document builder changes the order of the elements. Elements may also have attributes, which can give more information about an element value. In Listing 1, the attribute type="mobile" specifies the type of phone number: it is a mobile number.

2.1.2 Namespaces

Namespaces are used to uniquely identify XML elements and attributes [3]. This is important in cases, where XML documents embed other XML documents.

Example:

Let document $\mathcal{A}$ define an element $\mathcal{Z}$. Document $\mathcal{B}$ also defines a different element $\mathcal{Z}$. If $\mathcal{A}$ embeds $\mathcal{B}$, the parser can not distinguish between the two $\mathcal{Z}$ elements. This can be solved by using namespaces:

Listing 2: XML namespaces

```
1   <A xmlns="http://ns1.com">
2     <Z/>
3   </A>
4   <B xmlns="http://ns2.com">
5     <Z/>
6   </B>
```

In Listing 2, element $\mathcal{A}$ uses another namespace than element $\mathcal{B}$. Each child from $\mathcal{A}$ respecting $\mathcal{B}$ inherits the namespace from its parent element. This way, a parser can distinguish between both $\mathcal{Z}$ children, because they identified by their namespaces.

[2]Of course, the application needs to know these element names, see Section 2.1.3

Note: It is also possible to define a namespace prefix like xmlns:ns1="http://n1.com". If an element belongs to this namespace, it must start with the prefix. Listing 3 gives an example for using prefixes.

Listing 3: XML namespaces with prefixes

```
1 <ns1:A xmlns:ns1="http://ns1.com">
2   <ns1:Z/>
3 </ns1:A>
```

This can be used to declare a namespace only once and use the prefixes for the corresponding elements.

2.1.3 XML Schema

Remember the example of Listing 1: If an application wants to use such a document defining a *person* element, it needs to know its syntax. If it does not know the exact element name, it will have to guess it and search for element names like *firstname*, *first-name*, *prename*, etc. to get the first name. This leads back to the import problem mentioned in the previous example. In order to solve this problem, the application needs information about how elements are named and which children they can have.

XML Schema (XSD) is a well-founded commendation from W3C to define the structure of an XML document [4]. In contrast to its predecessor *Document Type Definition* (DTD), XSD uses XML itself to describe an XML Document.

XSD can be used to describe very complex data types. It uses control structures to define elements and their allowed sub elements. The leafs of all elements contain data which corresponds to one of 19 primitive data types, e.g. *xsd:string* or *xsd:integer*. One special child is *xsd:any*, which means, that any child elements are accepted.

2.1.4 DOM and SAX

There are two possibilities for parsing XML documents:

DOM: The *Document Object Model* reads the whole XML Data into memory and builds an object for each element. They are saved in a tree-like order, which allows easy access to each data node and relationship information like child, parent and sibling nodes.

SAX: The *Simple API for XML* does not save any object into memory. It parses an XML stream and sends an event if a new element starts or ends. This is extremely fast but more difficult to use, since the programmer can not go backwards, e.g. to get the parent node. He has to save all relevant data, like ancestor or sibling nodes, himself while processing and he can not directly access to any other node.

Both models are widely used and at first view DOM seems to be the better parser, because it is easier to use, but it is notable, that parsing only a small document will result in much higher memory consumption. For a server, which has to process several documents per second, this can lead to a bottleneck in memory, being abused by denial of service attacks. However, the SAX model also has its disadvantages: although it is fast and consumes minimal memory, some XML specific operations, like XPath [5], are not fully supported.

A third solution for parsing XML is the *Streaming API for XML* (StAX). It is a mixture compared to the features of SAX and DOM. SAX is a push parser: it reads a stream and sends out events. StAX works like a cursor: the programmer can ask for the next event, so StAX is a *pull* parser, which does not interact with the program actively.

2.2 SOA – Service-Oriented Architectures

Service-oriented architectures (SOA) define an abstract model of software architecture [6]. These services will be used by platform independent applications over a network.

The roots of SOA are located in business processes. They are high level descriptions of provided services, e.g. a bank provides a service “give a credit to”, which is the high level description for a lot of low level processes like “check credit rating”, “create new account” etc. SOA hides these low level processes as they are not interesting for clients.

From software point of view, it does not provide any implementation parts, it only defines an interface which can be used to talk to the service.

2.3 Web Services

Web services represent a concrete implementation for SOA. A web service client will load a WSDL (Section 2.3.1), which defines the request content of the SOAP message (Section 2.3.2) and submits it to the server hosting the requested service.

2.3.1 WSDL – Web Service Description Language

The *Web Service Description Language* (WSDL) is a specification for creating a client's web service message. Currently, there are two versions: Version 1.1 was published in 2001 by the W3C [7], but since 2007, there also exists a recommendation for version 2.0 [8]. Figure 1 shows the differences between them. Both versions are not meant for human reading, but if you try to do so, you should start reading from bottom to top.

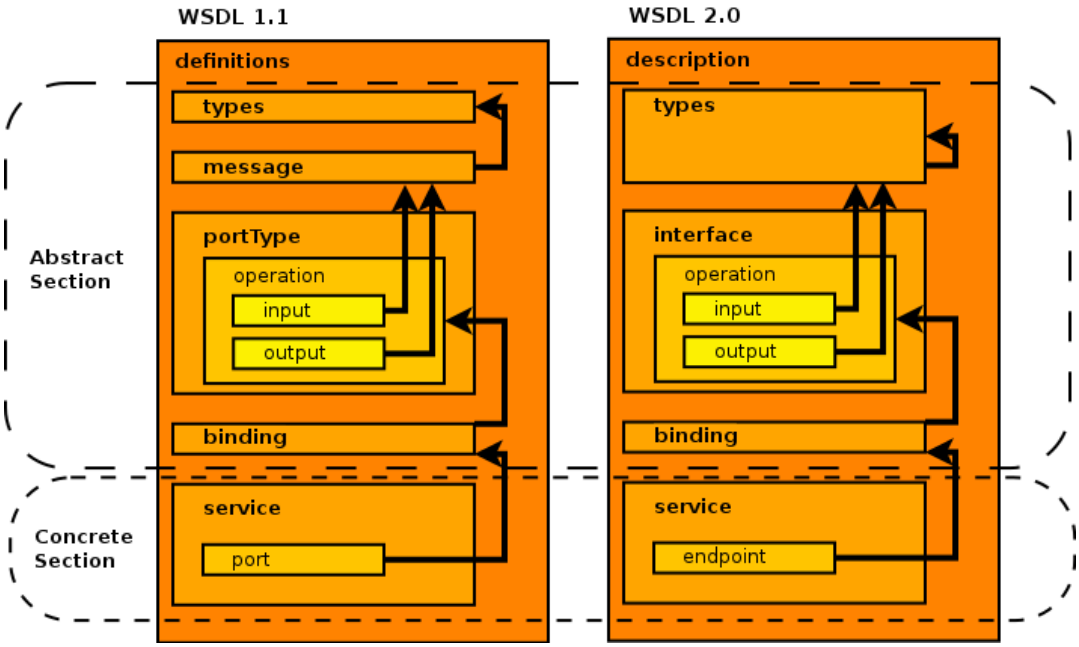

Figure 1: Comparison of WSDL 1.1 and WSDL 2.0

A WSDL can be divided into two parts:

1. The **Abstract Section** defines platform independent parts for creating messages. This means, it can be reused if the network address changes.
2. The **Concrete Section** defines the port/endpoint where the messages should be sent to.

The structure of a WSDL consists of the following elements from bottom to top:

Service: A container for similar endpoints. Some frameworks like Axis2 [9] generate one Service for SOAP 1.1 and one for SOAP 1.2 messages, see Section 2.3.2.

Port/Endpoint: Defines a network endpoint address or connection to the web service, in most cases represented by an HTTP URL string.

Binding: Concrete protocol definition and data format given by the **PortType/Interface**.

PortType/Interface: Defines the possible operations for an endpoint.

Operation: Like a function call in a programming language. The SOAPAction is defined and the structure of the **Message** is given.

Message/N.A.: Typically, a Message corresponds to an **Operation** and contains the information for executing it. Commonly, the operation name is used as a child of the SOAP Body. The Message can refer to **Types** for a more complex message, e.g. if the operation needs parameters.

Types: Defines an XSD (Section 2.1.3) to structure a message. In WSDL 2.0, this replaces the **Message** part.

For more information about WSDL see [7] and [8].

2.3.2 SOAP – Simple Object Access Protocol

The *Simple Object Access Protocol* (SOAP) is a standard which describes message exchange with a web service.

The W3C specified two versions of SOAP. Version 1.1 was published in May 2000 and Version 1.2 in April 2007. Both versions are still widely used. There are a few differences, like different SOAP Faults, different namespaces, and in general, one can say that SOAP 1.2 is more precise. For further details, see the specification [10] and [11].

SOAP does not specify any concrete transport protocol, but in most cases HTTP is used. Consequently, SOAP Messages can not be distinguished from normal HTTP traffic without any content analysis. This allows bypassing of most standard firewalls. Therefore, web services can be used whenever HTTP protocol is used.

The SOAP message basically consists of an *Envelope* element with two child elements named *Header* and *Body* (Figure 2):

Envelope: Container element for the SOAP **Header** and **Body**.

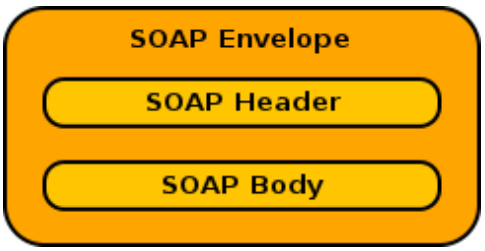

Figure 2: Structure of a SOAP Message (Namespaces are omitted).

Header: The optional SOAP Header element can contain any child elements (*xsd:any*). The idea is to have a place for additionally needed information like credentials or signatures.

Body: The SOAP Body also has a child of type *xsd:any*, which is used for storing the message defined by the WSDL for a web service operation. For SOAP responses, the Body element can contain a SOAP Fault element, which gives details if a SOAP request results in an error.

2.3.3 Security in Web Services

Web services are normally used via SOAP messages over HTTP. This allows two possibilities for secure communication:

SSL/TLS can be used, since it relies on a well-known and tested protocol [12]. As HTTP is used in most cases for message exchange, it is already implemented in most servers and clients.

XML specified security can be used to encrypt/sign XML Elements (also known as Message-Level Security).

On a quick peek, the second solution seems to be unnecessary. However, consider a web service $\mathcal{W}$ behind an application firewall $\mathcal{A}$: If a client $\mathcal{C}$ wants to use a $\mathcal{W}$, there is the following message exchange:

$$\mathcal{C} \rightleftarrows \mathcal{A} \rightleftarrows \mathcal{W}$$

$\mathcal{C}$ must establish an SSL connection to the firewall $\mathcal{A}$, but then, $\mathcal{A}$ has to forward the message to $\mathcal{W}$. For doing this, $\mathcal{A}$ has two possibilities:

1. Establishing a new SSL connection with $\mathcal{W}$, which is computationally intensive for $\mathcal{A}$.
2. Sending the message unencrypted to $\mathcal{W}$, allowing internal attackers to eavesdrop the message.

Since both approaches have its downside, this leads to the second idea for web service security: XML specific security. Therefore, the OASIS group maintained the *Web Service Security* (WS-Security) [13], which standardized how to:

1. Sign and verify (parts of) SOAP messages.
2. Encrypt and decrypt (parts of) SOAP messages.
3. Add security tokens (like timestamps, credentials) to SOAP messages.

If a web service wants to use these security features, it needs a *Web Service Security Policy* (WS-Security-Policy) [14], which can be embedded in a WSDL. The WS-Security-Policy provides details to a client, to which parts of the messages which security features shall be applied and which concrete algorithms must be used.

3 Threats to Web Services

The following section will give an overview of existing attack classes for web services. This thesis will not go into detail. For more information about attacks, see [15].

As two examples, SOAPAction Spoofing in Section 3.2 and WS-Addressing Spoofing in Section 3.3 are described. Both attacks are implemented as attack plugins for *WS-Attacker* and SOAPAction Spoofing will be used as an example implementation in Section 4.5.4.

3.1 Web Service Specific vs. Non-Specific Attacks

Web services use XML based messages for communication over HTTP. However, they are also programmed applications, which use a specific programming language and/or data base queries. Considering these two points of view, a web service is vulnerable to the following groups of attacks:

Non-specific web service attacks are abusing weaknesses in the back-end of an application, e.g Buffer Overflows [16] or SQL Injection [1].

Specific web service attacks exploit vulnerabilities on SOAP and XML. They attack the parser (DOM, SAX see Section 2.1.4) with Denial of Service attacks or build unexpected SOAP messages, e.g. change the SOAPAction header (Section 3.2).

The preferred way to build secure web services is checking for security by means of an attack framework. If the service resists these attacks, it is a good indicator for security.

Non-specific web service attacks are well-known from web applications. Information about them can be found on the OWASP web site [17]. A good all-in-one tool for testing web applications is the *Web Application Attack and Audit Framework* (w3af) [18].

A short overview of some well-known web service specific attacks taken from [19] can be seen in Table 1. Although there is a large number of attacks, there are only a few tools which can be used for security testing on web services.

SOAP Sonar by Crosscheck Networks [20] provides an automated one-click security scan, but it can only do non-specific web services attacks, and the annual licence costs about $15.000 per year.

Table 1: Overview of existing web service specific attacks taken from [19]

Oversize Payload
Coercive Parsing
SOAPAction Spoofing
XML Injection
WSDL Scanning
Metadata Spoofing
Attack Obfuscation
Oversized Cryptography
BPEL State Deviation
Instantiation Flooding
Indirect Flooding
WS-Addressing Spoofing
Middleware Hijacking

Currently, there is no automated vulnerability scanner which uses web service specific attacks. The only possibility to attack web services is to do manual tests, e.g. using soapUI [21]. As this is very time consuming, *WS-Attacker* is presented as an automated all-in-one web services attack and penetration testing framework in Section 4.

3.2 SOAPAction Spoofing

SOAPAction Spoofing is a web service specific attack [19]. Each SOAP request has a special HTTP header named *SOAPAction*, which indicates the operation that shall be performed. The goal of this header is to reduce computational time, as the operation can be detected without parsing the XML content to find the first child of the SOAP Body.

The idea of SOAPAction Spoofing is to have a SOAP message, which contains a SOAP-Action header that does not match the SOAP Body element.

Example:

Consider a web service with two operations: *OperationA* and *OperationB*. The WSDL for this service defines the SOAPAction for each operation in the *operation* element.

Let *ActionA* and *ActionB* be the corresponding actions. A valid SOAP message for *OperationA* is shown in Listing 4 (namespaces are left out).

Listing 4: A valid SOAP message for *OperationA*

```
1 POST /webservice HTTP/1.1
2 Host: soapActionSpoofingHost
3 SOAPAction: "ActionA"
4 <Envelope>
5   <Header/>
6   <Body>
7     <OperationA/>
8   </Body>
9 </Envelope>
```

A SOAPAction Spoofing attack will change the SOAPAction header to a different action as Listing 5 shows.

Listing 5: SOAPAction Spoofing Attack message

```
1 POST /webservice HTTP/1.1
2 Host: soapActionSpoofingHost
3 SOAPAction: "ActionB"
4 <Envelope>
5   <Header/>
6   <Body>
7     <OperationA/>
8   </Body>
9 </Envelope>
```

In some cases, this message can provoke an unwanted reaction.

Example:

Consider an HTTP Firewall, which handles incoming requests and a web service with two operations. If the firewall only checks the SOAPAction header, the message of Listing 5 is illegally allowed and will be forwarded to the web service, see Figure 3. The web service logic executes the SOAP Body operation, because it does not check authentication – it believes, that the firewall performs this task.

If *OperationB* is a public operation like *getServerTime* and *OperationA* one, that needs authentication, e.g. *deleteAllUsers*, the SOAPAction Spoofing attack can be used to execute *deleteAllUsers* without any authentication.

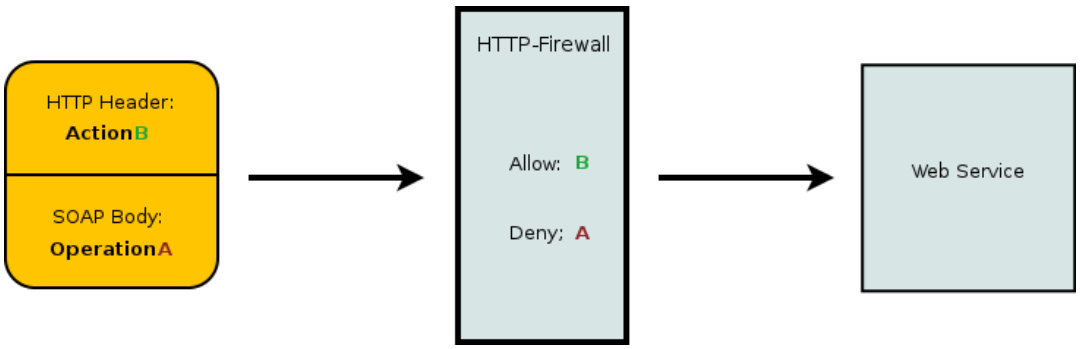

Figure 3: Attacking a web service with SOAPAction Spoofing

This attack may also apply in scenarios without a firewall. Some web services only check the SOAPAction header and execute the corresponding operation in order to save computational time. The SOAP message itself is only parsed if the operation needs additional parameters. If the server does not find them, because the SOAP Body child is different, it sets all parameters to their default value, e.g. empty strings. This behavior can be found in .NET services, see Section 5.

A real life example for this attack was published in January 2010 [22]. *SourceSec Security Research* found a vulnerability in D-Link routers, which allows administrative access using SOAPAction Spoofing.

3.3 WS-Addressing Spoofing

WS-Addressing Spoofing is a further web service specific attack [19]. The idea can be seen in Figure 4: The attacker sends a request to the server, which has a WS-Addressing Header [23] to provoke the server sending the SOAP response to a different endpoint.

The specification has three different methods for doing this:

ReplyTo: The server sends the response to any different endpoint.

FaultTo: The server sends any SOAP Fault to a different endpoint. For attacking a web service, a SOAP Body without any children can be used, as this will always return a SOAP Fault.

To: The server uses a different endpoint for everything.

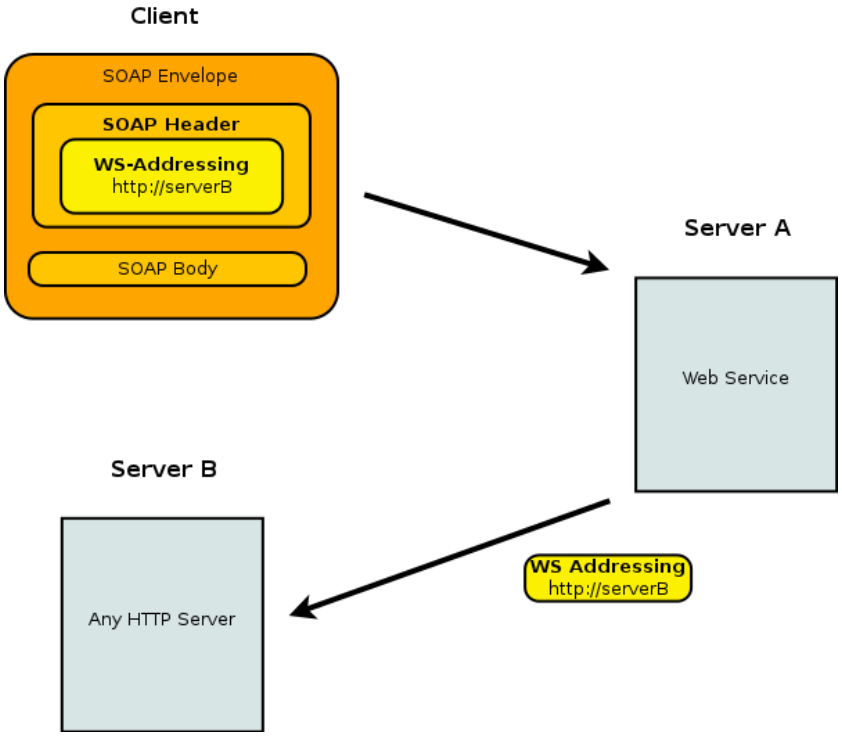

Figure 4: Idea of WS-Addressing Spoofing

Using WS-Addressing for asynchronous message exchange raises different attack possibilities, e.g. flooding another web service, or even Distributed Denial of Service is possible. Therefore, only one of the three methods mentioned above is sufficient.

A countermeasure against WS-Addressing Spoofing is the verification of the endpoint reference (Whitelist), ideally before any computation.

4 WS-Attacker – A Modular Testing Framework

Based on [15], a new project has started to categorize attacks for web services. The web services attacks open community project [24] provides information about existing web service attacks. The goal of this thesis is to build a web service attack framework, which easily allows an automated vulnerability scan. This section introduces the WS-Attacker framework, which can be extended via attack plugins. These plugins can contain *any* kind of web service attacks, respecting web service specific *and* non-specific attacks.

Section 4.1 specifies WS-Attacker's requirements by distinguishing two groups: plugin developers and framework users. Section 4.3 introduces soapUI as a main part of the framework, whereas Section 4.4 gives a general overview of WS-Attacker's internal structure. In Section 4.5, the attack plugin interface is described in detail.

4.1 Framework Requirements

WS-Attacker shall do a vulnerability scan on every web service and only needs a WSDL as input. This thesis is only aimed of creating an architecture, which can be used for implementing attacks. These attacks can be started via an easy understandable GUI[3]. Therefore, one has to distinguish the requirement for two groups: *plugin developers* and WS-Attacker *users*.

4.1.1 Requirements for Plugin Developers

The requirements for plugin developers can be summarized to the following aspects:

1. It must be **easy to implement** new attacks, where each attack is represented by a WS-Attacker plugin.
2. **Any attack category** must be supported (spoofing attacks, denial of service, etc.)
3. **Open for extension**, that means, there must be support even for prospective, not yet invented, attacks.

[3]Graphical User Interface

In general, a plugin author has the task to create the attack-plugin without knowing WS-Attacker's internals. It must be as easy as possible to create new attacks and any kind of attacks must be supported. This is why WS-Attacker will only provide a plugin interface and some helper classes. Each attack request must be sent by the plugin itself. This way, plugins can have multiple or single requests. An attack will get a request-response pair as a reference to build the attack vector and send it to the server. Receiving the new responses, each plugin will get some results, which will be published in a special way, so that the framework can visualize them.

4.1.2 Requirements for Framework Users

The requirements for framework users must be seen from a different point of view. They can be summarized as followed:

1. The framework must be **easy to use**.
2. Only a **few clicks** should be necessary to test a web service.
3. The users **do not need any knowledge about XML** or web services.

A typical WS-Attacker user might be a company, which provides a web service either for their clients, or for its internal processes. This service should be secure against all known attacks to web services. By using WS-Attacker, the company can easily check for vulnerabilities. In most cases, companies do not have employees who know a lot about security in web services. Moreover, they even do not know anything about XML, because they use a framework like Axis2 [9] to provide the service. In such cases, WS-Attacker is the perfect software solution. WS-Attacker uses a WSDL as input, the user chooses the operation to be tested and selects the attacks. A further click starts these attacks and the results will appear on the screen.

4.1.3 Goal of this Thesis

The goal of this thesis is to build the WS-Attacker *framework* and realize the requirements mentioned above. A vulnerability scanner is only as good as the attacks it can perform. It is not in the scope of this thesis to build a complete penetration testing tool, as this would imply to build attack plugins for all known attacks.

4.2 Overview

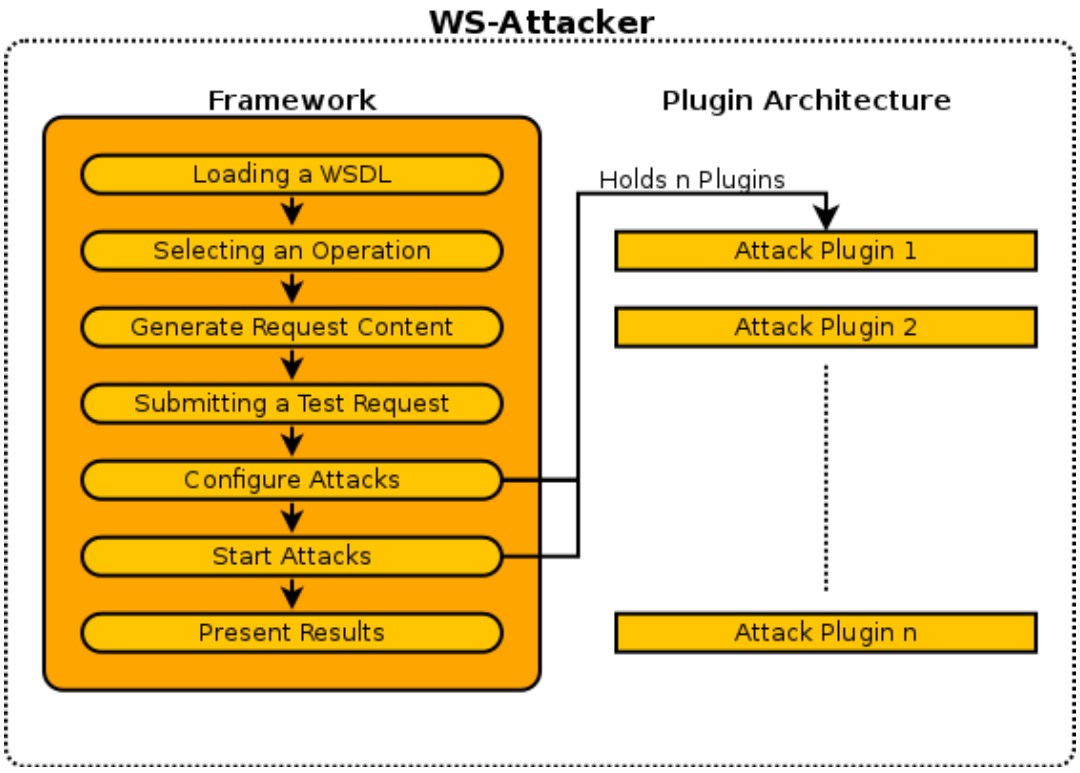

Figure 5: WS-Attacker: General overview

The general idea of WS-Attacker is shown in Figure 5. The program is divided into two parts:

Framework: This is the main part of WS-Attacker. Its task is to set up the environment for attacking web services.

Plugin Architecture: WS-Attacker can hold any number of plugins, where each plugin represents an attack.

The main task of the framework is to set up the configuration for attacking the web service. This will work as follows:

1. The user has to **load a WSDL**. This can be a local file or an URL.
2. After that, he has to **select the operation** which will be attacked.

3. The framework generates the request content and provides input fields for message parameters.
4. The user **submits a test request**, which will be used by the plugins for comparing attack responses to regular responses.
5. The plugins have to be **configured** and **enabled**.
6. The framework **starts** the enabled plugins.
7. **Results** generated by the plugins are presented to the user.

The plugin architecture allows to extend the framework with new attacks. Each plugin represents exactly one attack and the framework uses a plugin manager to hold and activate these plugins. The plugin structure is described in Section 4.5.

4.3 SoapUI as Back-end

SoapUI by Eviware Software is a tool for using and testing web services [21]. It is published under LGPL [25], can create requests out of an WSDL and send them to a server hosting the web service. Since 2005, soapUI is hosted on *sourceforge.net* and is one of the most frequently used testing tools in the world.

4.3.1 Advantages of soapUI

A web service testing framework like WS-Attacker needs to

1. create requests from a WSDL
2. edit the request parameters
3. send it to the server

Using Java, there are a few possibilities for doing this:

1. Do it yourself.
   This includes building an XML and an XSD parser. To send a request, some helper methods must be provided unless the plugin authors want to use raw HTTP sockets.

2. Use the Java SAAJ tools from *javax.xml.soap* [26]:
   This package can manipulate SOAP messages and provides some helper classes for sending requests.

3. Use a third party solution.

The first approach is very complex and time-consuming. A lot of tests needs to be created to find bugs. It is just simpler, faster and safer to rely on standards.

The second approach seems to be very good, since it uses standard Java packages and SAAJ is very flexible for creating and manipulating SOAP messages. Nevertheless, there are some problems:

1. Each XML element is saved as one object, but especially for web service specific attacks, one must be able to create malformed messages, e.g. create only open tags and no end tags. There are also problems if you try to add *greater than* or *lesser than* signs, because they are automatically escaped.

2. SAAJ does not provide a WSDL parser, so there is no possibility to create the basic request content for a defined operation.

Where (1) could be wrapped by serializing SAAJ objects and sending a manual request via custom HTTP sockets, problem (2) can not be solved as easily as (1).

There are two possibilities for creating a request from a WSDL. The first one uses the WSDL parser *wsdl4java*, which can parse a WSDL file and extract the operation name as well as the endpoint URI, but which is not able to generate the request content. It can only be generated by means of an XSD parser, which can extract the information from the *Types* section of a WSDL. The second makes use of the Axis2 tool *wsdl2java*. This one can create a request but generates Java code, giving no direct access to the request content.

All these problems lead to the third approach: Use a third party tool: soapUI is the perfect solution for doing this. It is written in Java and LGPL allows us to use it for custom programs. SoapUI is able to parse WSDL files, generate requests out of it and also to support helper methods for Basic Authentication, WS-Security etc. Sending requests to the server is just as easy. SoapUI uses strings to save the request content, which allows us to manipulate them and create malformed requests.

4.3.2 The Structure of the soapUI API

This section will give a brief overview of the soapUI API, which is used in the back-end of WS-Attacker for creating and sending requests. Although plugins can be built without using soapUI, it can simplify the process. For more details on this API, visit the online documentation [4].

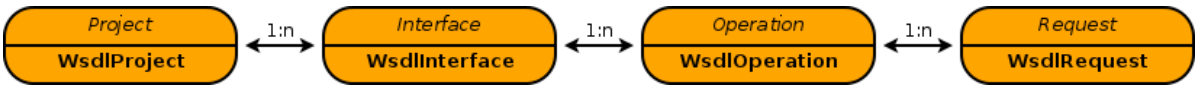

Figure 6: Overview: The soapUI API

Figure 6 shows the most important model items. The italic names represents the general Java interface, the bold ones the concrete Java class. The model can be seen as an object view of a WSDL. There is always a one to many relationship, i.e. one interface can have many operations. All relations are bidirectional: A request always knows its operation (like a parent in a tree) and each operation knows all of its requests.

Each model item has its own properties, e.g. the property *SOAPAction* is a property defined in *WsdlOperation*, like it is defined in the WSDL file. Each property can be changed and overwritten: If one defines another property *SOAPAction* in a *WsdlRequest*, this property will be used when the request is submitted and the equivalent property defined in *WsdlOperation* will be ignored.

Listing 6: Submitting a *WsdlRequest*

```
1 WsdlSubmit<WsdlRequest> submit = request.submit(new
    WsdlSubmitContext(request), false);
```

Listing 6 gives an example for submitting a request. The *WsdlSubmitContext* object resolves the properties of the given request, that means, it expands properties which are not directly set by the request and adds them from its parents as in the *SOAPAction* example mentioned above. The second parameter defines whether the request is submitted asynchronously.

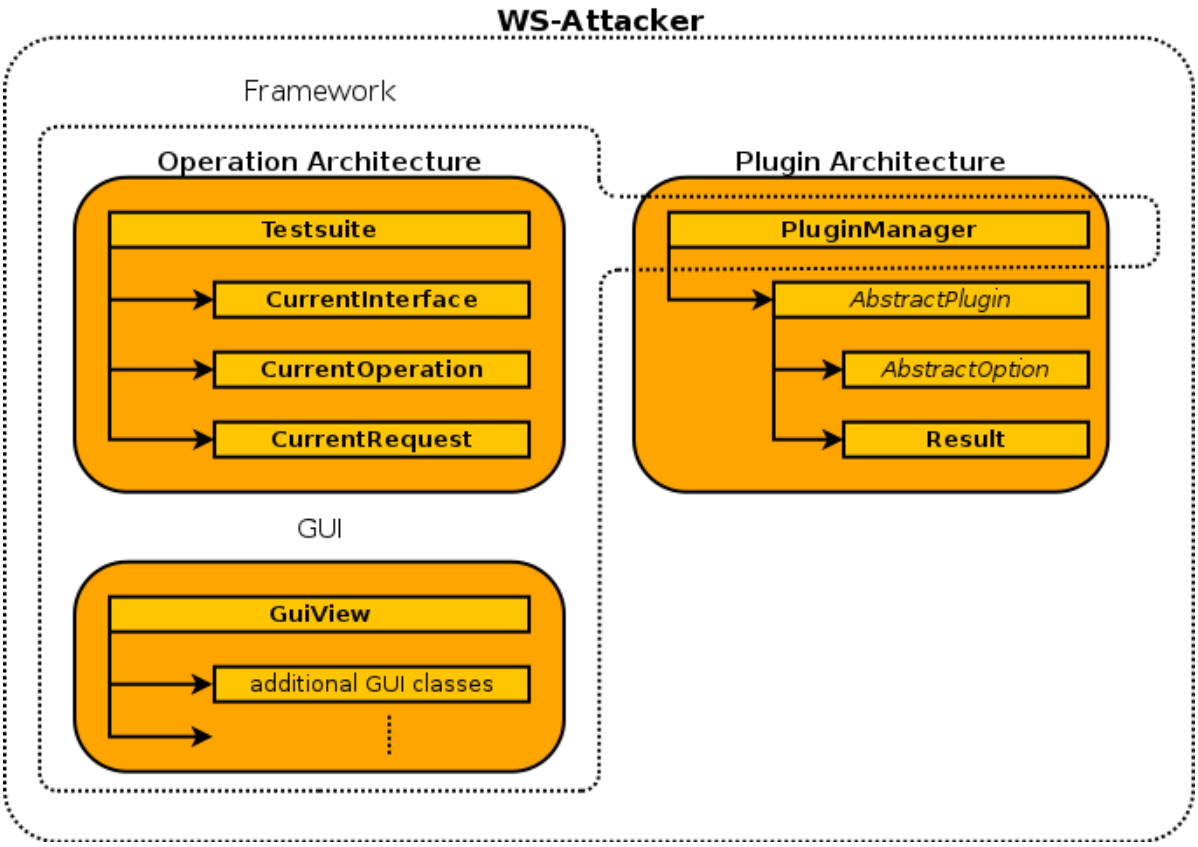

Figure 7: The internal structure of WS-Attacker

4.4 Program Structure

This section will give a more detailed overview of WS-Attacker's internal structure. Figure 7 shows a class overview of the must important parts: The *Plugin Architecture* and the *Operation Architecture*. Concrete classes have bold names in rectangles, abstract classes are italic.

Operation Architecture: The operation architecture represents everything that has to do with creating web service requests and can be seen as the main part of the framework.

Plugin Architecture: The plugin architecture represents the plugin system.

The Operation Architecture has a *Testsuite*, which acts like a wrapper for soapUI. It can load a WSDL and select the *CurrentInterface* as well as a *CurrentOperation* to generate

[4] `http://www.soapui.org/apidocs/overview-summary.html`

the *CurrentRequest*. WS-Attacker only needs *one* operation and *one* request that will be used for attacking. The *CurrentRequest* can also be sent to the web service server so that each attack plugin can use the response for comparing it to the attack response.

The *PluginManager* holds all available attack plugins. Each plugin extends the *AbstractPlugin* class and can have one or more *AbstractOption*s, for example a signature file or some other configuration parameters. The WS-Attacker GUI will read these options and present a graphical input method to the framework user. To distinguish between different data types, sub-interfaces of *AbstractOptions* like *AbstractOptionInteger* and *AbstractOptionBoolean* have been built.

The results of the attack are collected in the *Result* object. It can be compared to an advanced log file, which the GUI will use to present the results to the user. Results can be filtered by the plugin source and a level, which indicates how important a result is. The user can choose whether he wants to see only the most important results, e.g. which parts of the attack was successful, or even request and response contents.

4.5 Attack Plugin Interface

4.5.1 Basic Idea

In general, an attack plugin has the following tasks:

1. Running the attack using the given operation, request and plugin options.
2. Generating some results which shall be displayed to the user.
3. Giving a rating about the outcome of the attack.

The first point is obvious – an attack implementation is needed.

While processing the attack, information about what is happening must be saved as results. The user will see those results and can filter them according to their level. If he only wants to know the most important pieces of information, he can filter only *critical* results. Nevertheless, if he wants to see request/response contents, he chooses the *tracing* level.

Furthermore, each plugin needs to rate the attack success. Therefore, a plugin author has to take the following two steps:

1. Giving an integer rating for his attack, which means, he has to set a maximum number of points (integer), that can be reached during the attack and increase the reached points depending on the attack success.
2. Implementing a *wasSucessful()* method to give a Boolean result.

It is important to distinguish between these two ways: Although a specific attack might be successful, it could have a variable attack potential. E.g. a denial of service attack might stop a server for several minutes or even completely so that a reboot is necessary. This can be described using such a rating. The *wasSucessful()* method should just indicate whether a web service is vulnerable in general or not.

4.5.2 Extending *AbstractPlugin*

To build attack plugins, each plugin must extend the *AbstractPlugin* interface. Listing 7 gives an overview of its methods.

Listing 7: The *AbstractPlugin* interface (shorted).

```
1  public abstract class AbstractPlugin implements ↷
       SuccessInterface {
2
3      public abstract void initializePlugin();
4
5      // attack identity
6      public abstract String getName();
7      public abstract String getDescription();
8      public abstract String[] getCategory();
9      public abstract String getAuthor();
10     public abstract String getVersion();
11
12     // success interface
13     public abstract int getMaxPoints();
14     public abstract boolean wasSuccessful();
15     final public int getCurrentPoints();
16     // for internal use
17     final protected void setCurrentPoints(int points);
18     final protected void addOnePoint();
19
20     // logger for outputs like errors, which are no plugin ↷
           results
21     final protected Logger log();
22     // generating results
```

```
23      final protected void result(ResultLevel level, String content);
24      final protected void critical(String content);
25      final protected void important(String content);
26      final protected void info(String content);
27      final protected void trace(String content);
28
29      // the plugin state
30      final public PluginState getState();
31      final protected void setState(PluginState state);
32      final public boolean isRunning().
33      final public boolean isReady().
34      final public boolean isFinished().
35      final public boolean isFailed().
36
37      // get the plugin options
38      final public PluginOptionContainer getPluginOptions();
39      // the plugin will be observer if an option value changed
40      // this can be used to set the plugin state to ready/not configured
41      public void optionValueChanged(AbstractOption option);
42
43      // restoring an old configuration
44      public void restoreConfiguration(AbstractPlugin plugin);
45
46      // main part
47      final public boolean startAttack();
48      // implement the attack here
49      protected abstract void attackImplementationHook(final RequestResponsePair original);
50      // clean: prepare plugin for next run
51      public abstract void clean();
52      // stop: if user interrupted the attack
53      public abstract void stopAttack();
54  }
```

The *AbstractPlugin* Class contains three types of methods:

Abstract methods must be implemented by the plugin author.

Final methods can not be overridden. They shall be used in the implementation as helper methods..

Public methods that are neither *abstract* nor *final* **can** optionally be overridden to change their default behavior.

The methods can be divided into different parts:

Attack identity contains methods to describe a plugin. This includes an attack name, an author, a version and a description. The plugin category must be an array of strings, where the first element is the main category and any further element is a sub category of its predecessor.

Success interface implements the idea mentioned in Section 4.5.1. The interface implements the *wasSucessful()* function as well as a success rating, which can be seen as a fraction: $\frac{\text{getCurrentPoints()}}{\text{getMaxPoints()}}$.

A logger is provided via the *log()* method. It shall be used to log internal errors, e.g. if a plugin can not start.

Results can be generated. Therefore, some helper methods are given for smarter creation.

Plugin states are used to describe the status of a plugin. These methods can set weather the plugin is ready, running, finished etc.

Plugin Options must be accessed by the *getPluginOptions()* method. This will return a container where plugin authors can add their own options. The added options will be automatically observed. If an option value changed, the plugin is notified with the *optionValueChanged()* method.

Restoring configuration of plugins is possible via the *restoreConfiguration()* method. The default implementation takes each option from *getPluginOptions()*, gets its value and resets it on the corresponding option of the current plugin.

Main part: A plugin will be started by the *startAttack()* method. This method is *final* but internally calls the *attackHook()* method, which the plugin author has to implement. Furthermore, a *clean()* method must be implemented to reset the plugin (e.g. to reset the success rating) and prepare it for the next run. The *stopAttack()* method will be called, after the user requested to abort the current attack. The attack thread will be stopped and the stop method can be used to clean things, e.g. free sockets.

Most of these methods will be used as subroutines in the main part. The attack will use the log and result methods depending on its processing; set the current points to a specific value and finally change the state to finished or failed.

4.5.3 Using Plugin Options

WS-Attacker includes a very expandable plugin option system:

Each plugin option has to extend *AbstractOption*. Its methods are shown in Listing 8.

Listing 8: The *AbstractOption* interface (shorted)

```
1  public abstract class AbstractOption {
2      // constructors
3      public AbstractOption(String name, String description);
4      public AbstractOption(String name);
5
6      // an option at least has a name and a description
7      public String getName();
8      public String getDescription();
9
10     // each option belongs to one option container
11     public final PluginOptionContainer getCollection();
12     public final void setCollection(PluginOptionContainer ↷
           collection);
13
14     // notify if value has changed
15     protected final void notifyValueChanged();
16
17     // check if a value is valid for this option
18     public abstract boolean isValid(String value); // only for ↷
           generic proposals
19
20     // each option can at least be set and read by strings
21     public abstract boolean parseValue(String value); // only ↷
           for generic proposals
22     public abstract String getValueAsString(); // only for ↷
           generic proposals
23 }
```

An Option is characterized by its name and a description. The *parseValue()* and *getValueAsString()* methods ensure that each option is at least accessible by strings. There

are further sub interfaces of *AbstractOption* like *AbstractOptionBoolean* and *AbstractOptionInteger*, which provides methods to access the value with the specific type. The GUI will check for these interfaces and provide a fitting input method.

There are also some concrete implementations like *OptionSimpleBoolean*. This option is just what the name says: A check-box which can be on or off. However, the clue of *AbstractOption* is the *isValid()* method, which can be used to generate specific options. Consider two Boolean options, which are disjunct: If A is on, B must be off and vice versa – or consider just any string option which starts with *foo* and ends with *bar*. To implement such options, you could easily extend the corresponding interfaces and implement the *isValid()* method.

WS-Attacker also supports complex options via the *AbstractOptionComplex* interface. It can be used if the common options do not fit. Therefore, one has to implement only one method named *getComplexGUI* which returns an *AbstractOptionGUI* component. This way, any possible option can be implemented, but this is a lot more work, since the author has to create a GUI by himself.

4.5.4 Minimal Implementation

This Section will give a minimal implementation example for an attack plugin, using SOAPAction Spoofing as an example. It will not give source code examples, but rather describe the idea of building a plugin.

In general, there are four steps to take:

1. Implementing the attack identity methods like *getName()* and *getDescription()*.
2. Implementing the success interface.
3. Implementing the plugin options (configuration parameters), if any are needed.
4. Implementing the attack itself.

The first step is obvious. After this, a success interface has to be implemented. Therefore, it will be distinguished between the following aspects depending on the SOAP response:

0. The response has a SOAP Fault. This is the only correct handling for SOAPAction Spoofing requests.

1. The response is not a SOAP message. Maybe it does not even contain any XML. This leads to a server internal error or misconfiguration. Eventually, some internals can be revealed, as this failure must be unwanted from server side – otherwise, the server would have sent a SOAP Fault.
2. The server ignores the SOAPAction Header and executes the first child of the SOAP Body. This could be used to bypass authentication, e.g. if a web service firewall only checks the SOAPAction header and the web service logic always executes the operation defined in the SOAP Body.
3. The server just executes the operation defined in the SOAPAction Header. This can be abused to invoke operations, which do not have any parameters (consider an operation like *deleteAllUsers*), because the server will search for them in the first SOAP Body child, which is different to the one the server expects.

As mentioned before, an attack can have different levels of success: Although (2) and (3) can be abused to executed operations without any authorization, (3) is easier to use, as it only needs to change the SOAPAction. The list above will be used for the integer success interface, so that a framework user can see what was successful and how dangerous it was. Additionally, a Boolean result will be implemented: *wasSucessful()* will return **true** if the attack reached two or three points. Again: The only correct handling for such requests is to send a SOAP Fault.

Next step is to implement the plugin options, if necessary. In case of SOAPAction Spoofing, the attack can have two different modes:

1. An automatic mode, which will generate a list of all possible SOAPAction headers and send an attack request for each of it.
2. A manual mode, which will let the user set the SOAPAction header manually. Therefore, a drop-down list with all operations different to the current operation are shown. The user can select the operation and the corresponding SOAPAction will be displayed in an input field. This field can also be edited.

In most cases, a user will start the attack in automatic mode, but the manual mode provides a way to set up a SOAPAction to anything the user chooses – e.g. if the user only wants to check a specific action, because some action may cause damage to the system.

As a last implementation step, the concrete attack must be implemented. Therefore, one has to override the *attackHook()* method, which will get a *RequestResponsePair* as a parameter for comparison. The attack will work as follows:

1. Get a list of SOAPActions to be checked, depending on automatic or manual mode.
2. Generate a new attack request as a copy of the comparison request.
3. Repeat for each SOAPAction while the maximum points are not reached:
   3.1. Submit the attack request with the specified SOAPAction.
   3.2. Search for the first body child in the response.
   3.3. Compare this child to the one from the comparison response, determine the kind of success and set the points for this.

For a more detailed documentation, a view into the Java source code is recommended.

4.5.5 Adding a plugin to WS-Attacker

The last Section dealt with how to build a plugin. This section now explains how to add the plugin to the WS-Attacker suite. Therefore, a Jar file must be built.

WS-Attacker's *PluginManager* uses Suns Java service loader from *java.util.ServiceLoader* to instantiate a plugin. This works as follows:

1. Add all Jar files from folder *plugin* to the classpath.
2. Search each Jar file for a file named META−INF/services/wsattacker.main. ↷ composition.plugin.AbstractPlugin.
3. Instantiate the class where the content of this file points to.

For adding Jars to the classpath, the soapUI utility *ClasspathHacker.addFile()* is used. Step two and three are realized by using the Java service loader:

Listing 9: Loading plugins from classpath using Java service load

```
1 ServiceLoader<AbstractPlugin> loader = ServiceLoader.load( ↷
    AbstractPlugin.class);
2 for ( AbstractPlugin p : loader) {
3         p.initializePlugin();
4         addPlugin(p);
5 }
```

Thus, for completing the development of the SOAPAction Spoofing plugin, a Jar file must be generated, which includes all compiled classes. Additionally, it has to include the *META-INF/services/*-file which points on the name of the absolute Java class that shall be initialized.

After this, the Jar file can be placed in WS-Attacker's plugin folder.

5 Evaluation

In this section, WS-Attacker is used for penetration testing on two different web services. Apart from the SOAPAction Spoofing Plugin, a second attack plugin for WS-Addressing Spoofing based on Section 3.3 is build analog to Section 4.5.4. These two plugins will be used for attacking a .NET based web service in Section 5.1 and an Axis2 based web service in Section 5.2.

5.1 Microsoft .NET

In the first penetration test, a custom web service is created with *Microsoft Visual Studio 2005*. It has two operations: *HelloName* and *GoodbyeName*.

After starting WS-Attacker, the GUI appears and offers an input field to enter the location of the WSDL, see Figure 8.

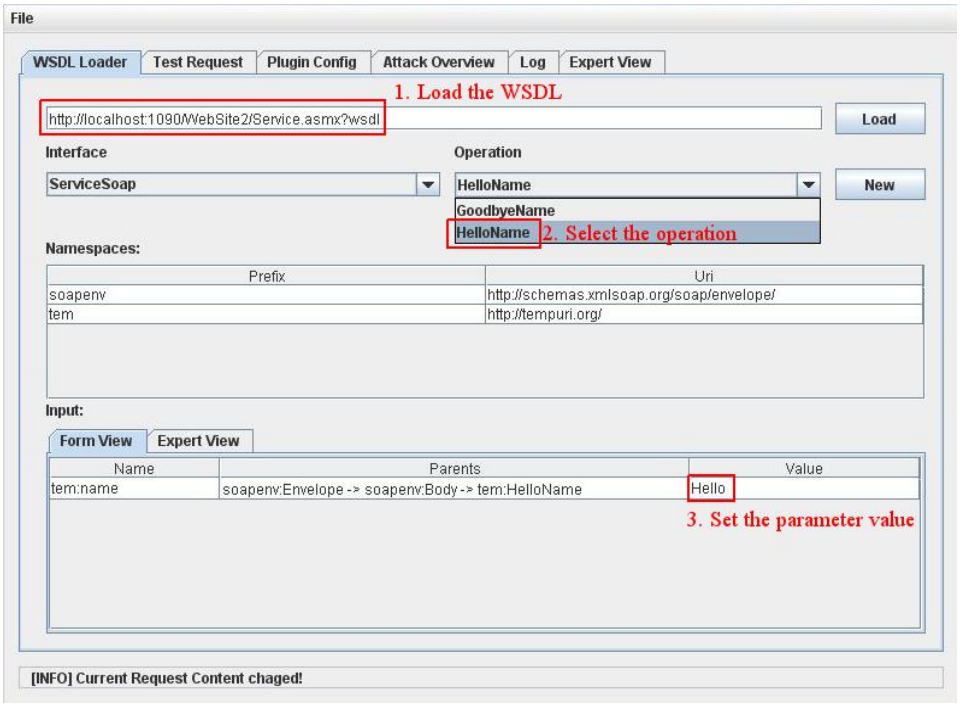

Figure 8: Loading the WSDL

HelloName is chosen as the operation to be tested. The table at the bottom gives a form based input possibility for all request parameters and in this case, *name* is set to *John*.

Next step is to do a test request: Figure 9 shows the test request and the corresponding response. The request contains a "HelloName" element as first body child and the response holds the corresponding element "HelloNameResult". This request is very important as attack plugins will use its response for comparing it with the responses of the attack request. This allows to check, what has really changed due to attack modifications.

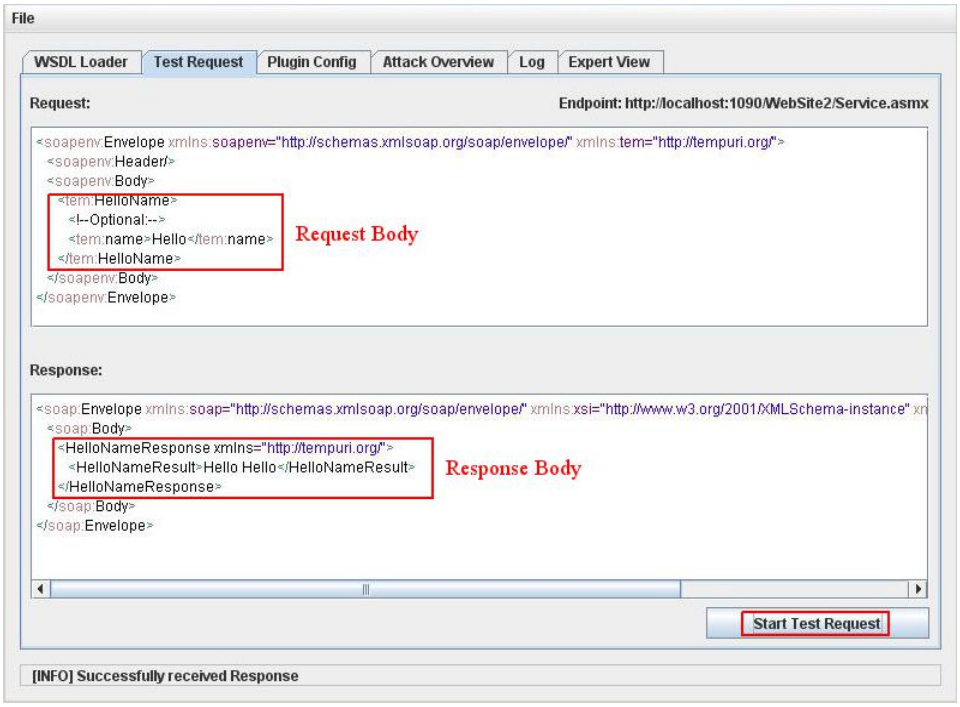

Figure 9: Submitting a test request

The next step is to configure the plugins. In this case, the automatic mode is used for SOAPAction Spoofing (Figure 10) and the WS-Addressing Spoofing plugin detects the endpoint URL automatically (Figure 11), too – there is nothing to configure manually. The tree on the left shares different views on the plugins. *Active Plugins* contains all plugins which will be used for attacking the server, *All Plugins* contains all plugins ordered by their category and *Alphabetical Sorted* shows all plugins in an alphabetical

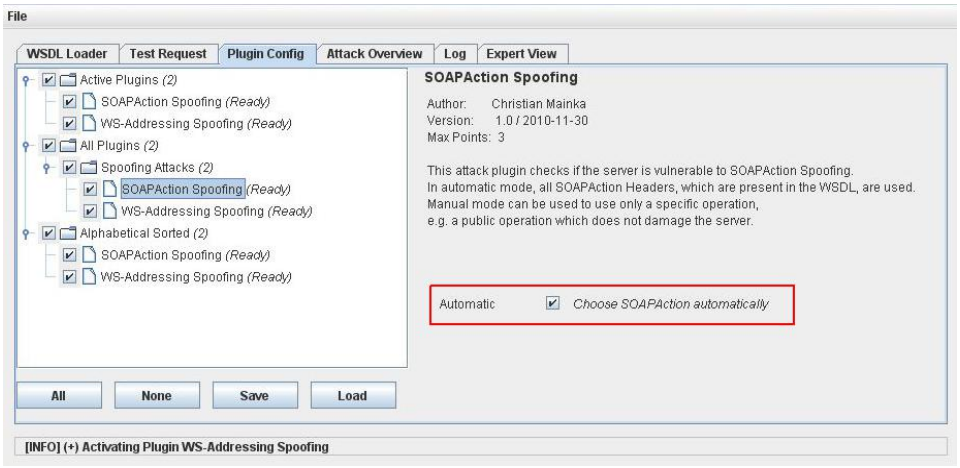

Figure 10: Plugin configuration for SOAPAction Spoofing

order.

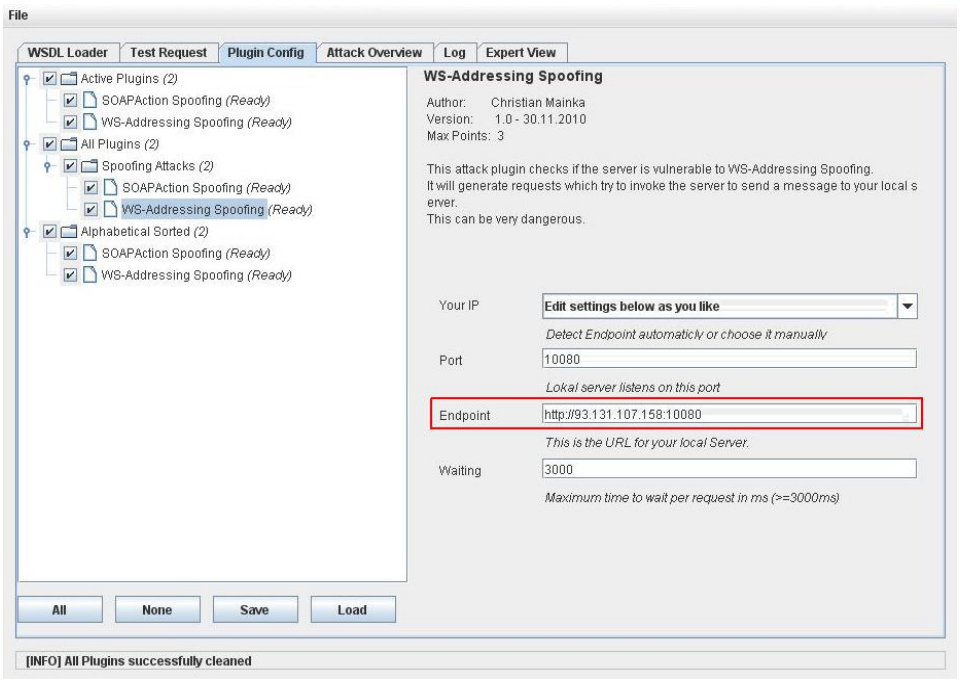

Figure 11: Plugin configuration for WS-Addressing Spoofing

The last step is to start the attack. Figure 12 shows the overview of a finished attack run. Active plugins are displayed on the top, their results at the bottom. The slider in the top part allows to filter the results by their level. The user can choose to see only

the most *important* results, or see even the request content at the *tracing* level.

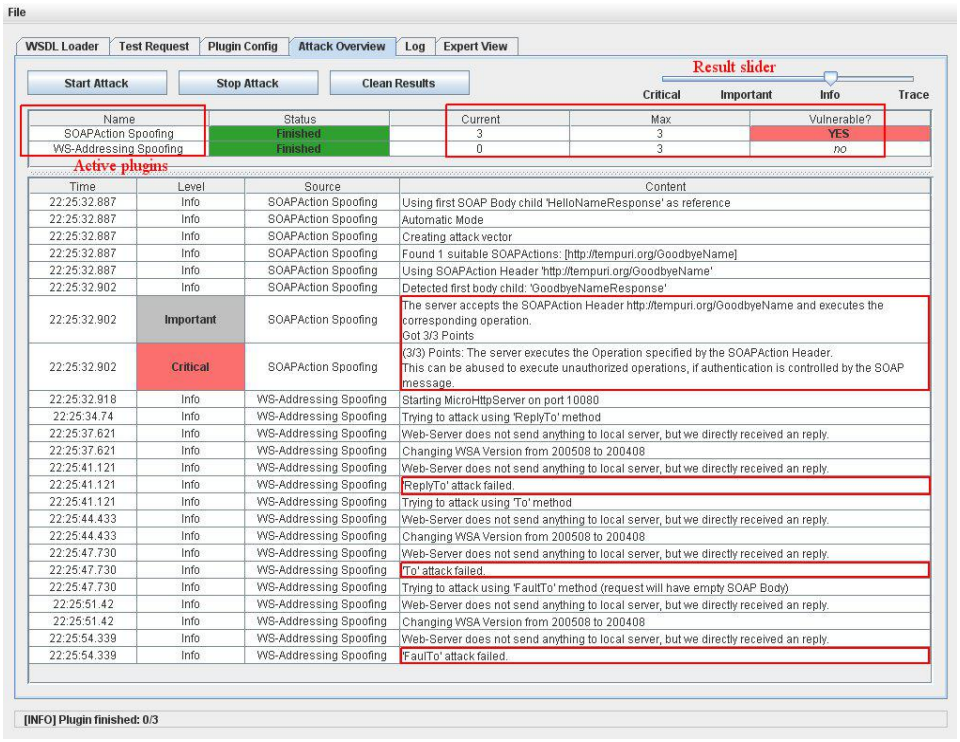

Figure 12: Penetration test on .NET finished.

The .NET web service is vulnerable to SOAPAction Spoofing but resistant to WS-Addressing Spoofing. This is indicated different aspects:

1. The *vulnerable* column values show **YES** for SOAPAction Spoofing and *no* for WS-Addressing Spoofing.
2. The SOAPAction Spoofing plugin got the maximum rating – three of three points in this case – and WS-Addressing Spoofing got zero points.
3. The results show, that the server has executed the operation defined in the SOAP-Action Header, which is the most critical security issue.

5.2 Apache Axis2

In the second penetration test, a new web service with the same operations and parameters as mentioned in Section 5.1 is created for Axis2 (Version 1.5.3) running on *Apache Tomcat/7.0.2* – this means, only recent program versions are used and everything runs under default configuration.

The plugin configuration is identically to the one from the previous test.

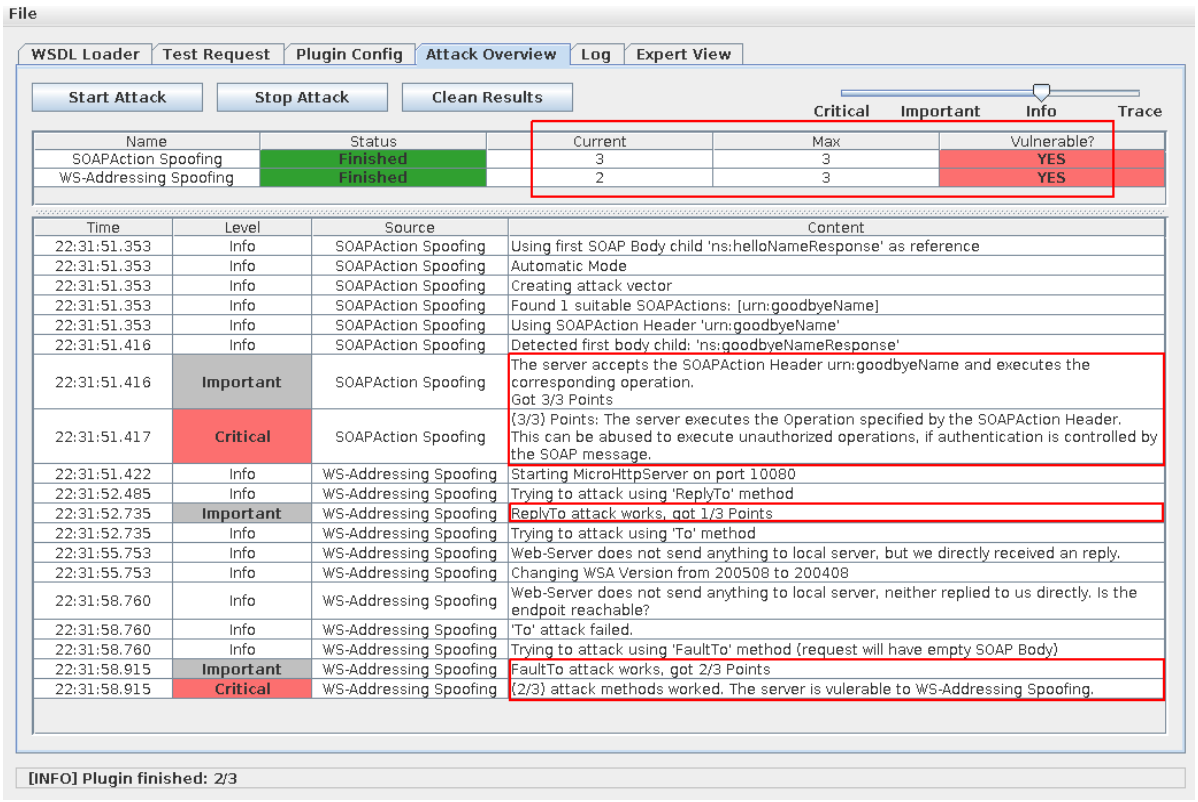

Figure 13: Penetration test on Axis2 finished.

The results can be seen in Figure 13. The Axis2 web service is vulnerable for both attacks:

- The server executes the operation defined by the SOAPAction Header, which is the worst case.
- The server understands WS-Addressing and accepts *any* endpoint for the *ReplyTo* and *FaultTo* methods.

This test shows, that these vulnerabilities even exist on recent program versions (Axis2 Version 1.5.3 was released on 12th November 2010) and accentuate the need of a web services attack framework like WS-Attacker.

6 Conclusion and Outlook

Web services are widely used. However, until now, mainly manual security testing is possible, which is very time and money consuming. This requires a lot of special knowledge about how a web service works, its internals and, of course what kinds of attacks exist. WS-Attacker is a modular framework, which allows a user to do web services penetration testing, without requiring any of this knowledge for using it. It is an all-in-one security checking tool, which can be used with only a few clicks. The user has to load a WSDL, activate the attacks and start them. Afterwards, an overview of the results is shown. The user is able to filter the displayed results by a level, e.g. critical or tracing, and sees which attack was successful including the level of success.

For plugin developers, WS-Attacker provides some helper utilities for attack implementation, like plugin options and conversation tools[5] from strings to SAAJ objects. It is possible to distinguish between different levels of success to rate the impact of an attack.

The development of WS-Attacker has just started, the framework itself is usable and already two attacks are implemented as a proof of concept. More attack implementations are needed, as WS-Attacker is only as good as its attacks. If this comes into truth, WS-Attacker will be the web services equivalent to web application testing frameworks like *w3af* [18]. Companies and web service framework developers can use WS-Attacker to check their applications for security issues. Even the Axis2 Framework, which is one of the most deployed web service framework in the world, is vulnerable to WS specific attacks like SOAPAction Spoofing and WS-Addressing Spoofing as show in Section 5. There are also attacks like signature wrapping [27] which are known for many years, but still not fixed in many web services implementations. These issues can be detected by simply using WS-Attacker and its plugins.

WS-Attacker also has its limits. Even if all known attacks are implemented, it can never say for sure that a web service is secure, as there might be unknown attacks in the future, unimaginable at the moment. It is only possible to say, that a web service is secure against existing implemented attacks. With the help of the web services attacks open community project [24], new attacks can instantly be reported and new plugins can be built to extend WS-Attacker's functionality.

[5] See class *wsattacker.util.SoapUtilities*

The fundamentals are set up, web service attacks can be categorized and attack plugins can be built. Now it is up to the community to extend the possibilities and securing web services by using WS-Attacker.

Appendix

References

[1] *SQL Injection.* http://www.owasp.org/index.php/SQL_Injection, visited on 18th October 2010.

[2] Bray, Tim, Jean Paoli, Eve Maler, François Yergeau, and C. M. Sperberg-McQueen: *Extensible Markup Language (XML) 1.0 (Fifth Edition).* W3C Recommendation, W3C, November 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

[3] Tobin, Richard, Andrew Layman, Tim Bray, and Dave Hollander: *Namespaces in XML 1.1 (Second Edition).* W3C Recommendation, W3C, August 2006. http://www.w3.org/TR/2006/REC-xml-names11-20060816.

[4] Sperberg-McQueen, C. M., Henry S. Thompson, Murray Maloney, Henry S. Thompson, David Beech, Noah Mendelsohn, and Shudi (Sandy) Gao: *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures.* Last Call WD, W3C, December 2009. http://www.w3.org/TR/2009/WD-xmlschema11-1-20091203/.

[5] Christensen, Erik, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana: *XML Path Language (XPath) Version 1.0.* Technical report, World Wide Web Consortium, 1999. http://www.w3.org/TR/2001/NOTE-wsdl-20010315, visited on 13th October 2010.

[6] Melzer, Ingo: *Service-orientierte Architekturen mit Web Services: Konzepte - Standards - Praxis (German Edition).* Spektrum Akademischer Verlag, 2010, ISBN 3827425492.

[7] Christensen, E., F. Curbera, G. Meredith, and S. Weerawarana: *Web Services Description Language (WSDL) 1.1.* W3C Note, World Wide Web Consortium, March 2001. http://www.w3.org/TR/wsdl.

[8] Moreau, Jean J., Roberto Chinnici, Arthur Ryman, and Sanjiva Weerawarana: *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language.* Candidate Recommendation, W3C, March 2006.

[9] *Apache Axix2 - Next Generation Web Services.* http://ws.apache.org/axis2/, visited on 16th October 2010.

[10] Box, D., D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer: *Simple Object Access Protocol (SOAP) 1.1*, 2000. `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`.

[11] Gudgin, Martin, Marc Hadley, Noah Mendelsohn, Jean Jacques Moreau, Henrik F. Nielsen, Anish Karmarkar, and Yves Lafon: *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. Technical report, April 2007. `http://www.w3.org/TR/soap12-part1/`.

[12] Dierks, T. and C. Allen: *RFC 2246: The TLS Protocol Version 1*, January 1999. `ftp://ftp.internic.net/rfc/rfc2246.txt,acknowledgement=ack-nhfb,format=`, Status: PROPOSED STANDARD.

[13] *WS-Security*. `http://www.oasis-open.org/specs/#wssv1.1`, visited on 16th October 2010.

[14] Vedamuthu, Asir S., David Orchard, Frederick Hirsch, Maryann Hondo, Prasad Yendluri, Toufic Boubez, and Ümit Yalçinalp: *Web Services Policy 1.5 - Framework*. Technical report, September 2007. `http://www.w3.org/TR/ws-policy/`.

[15] *Categorization and description of attacks on web services*, 2010.

[16] *Buffer overflow*. `http://www.owasp.org/index.php/Buffer_overflow`, visited on 18th October 2010.

[17] *The Free and Open Application Security Community*. `http://www.owasp.org/`, visited on 18th October 2010.

[18] *w3af - Web Application Attack and Audit Framework*. `http://w3af.sourceforge.net/`, visited on 18th October 2010.

[19] Jensen, Meiko, Nils Gruschka, and Ralph Herkenhöner: *A survey of attacks on web services*. Computer Science - R&D, 24(4):185–197, 2009.

[20] *SOAPSonar Enterprise*. `http://www.crosschecknet.com/products/soapsonardetails_platinum.php`, visited on 18th October 2010.

[21] *soapUI*. `http://www.eviware.com/`, visited on 18th October 2010.

[22] *Hacking D-Link Routers With HNAP*. `http://www.sourcesec.com/Lab/dlink_hnap_captcha.pdf`, visited on 4th October 2010.

[23] Box, Don, Erik Christensen, Francisco Curbera, Donald Ferguson, Jeffrey Frey, Marc Hadley, Chris Kaler, David Langworthy, Frank Leymann, Brad Lovering, Steve Lucco, Steve Millet, Nirmal Mukhi, Mark Nottingham, David Orchard, John Shewchuk, Eugène Sindambiwe, Tony Storey, Sanjiva Weerawarana, and Steve Winkler: *Web Services Addressing (WS-Addressing)*. Technical report, W3C, August 2004. `http://www.w3.org/Submission/ws-addressing/`.

[24] *The web services attacks open community project*. `http://ws-attacks.org/`.

[25] *GNU Lesser General Public License (LGPL)*. `http://www.gnu.org/licenses/lgpl.html`, visited on 22nd October 2010.

[26] Hewitt, Eben: *Java SOA Cookbook*. O'Reilly Media, Inc., 2009, ISBN 0596520727, 9780596520724.

[27] McIntosh, Michael and Paula Austel: *XML signature element wrapping attacks and countermeasures*. In *Proceedings of the 2005 workshop on Secure web services*, SWS '05, pages 20–27, New York, NY, USA, 2005. ACM, ISBN 1-59593-234-8. `http://doi.acm.org/10.1145/1103022.1103026`.

List of Figures

Listings

Used Software

The list below gives a detailed description of used software and libraries for creating WS-Attacker.

Name	Version	Company	Used for/as
OpenJDK	6.b20	Sun Microsystems	Java compiler
Eclipse	3.6.1	Eclipse Foundation	IDE, Released under the terms of the Eclipse Public License, Eclipse is free and open source software
soapUI	3.5.1	Eviware Software	Back-end library for parsing WSDL and creating requests, LGPL
Jigloo	4.6.2	CloudGarden	Assistant for building the GUI, free for non-commercial use
Checkboxtree	3.2	`zeus.pin.unifi.it`	Categorize plugins in a tree, GPL

Eigenständigkeitserklärung

Hiermit versichere ich, Christian Mainka (Matrikelnummer: 108007212667), dass ich die Arbeit selbständig angefertigt, außer den im Quellen- und Literaturverzeichnis sowie in den Anmerkungen genannten Hilfsmitteln keine weiteren benutzt und alle Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen sind, unter Angabe der Quellen als Entlehnung kenntlich gemacht habe.

Ort, Datum

Christian Mainka