Penetration Testing Tool for Web Services Security

Christian Mainka, Juraj Somorovsky, Jörg Schwenk,
Horst Görtz Institute for IT Security
Ruhr University Bochum, Germany
{Christian.Mainka, Juraj.Somorovsky, Joerg.Schwenk}@rub.de

***Abstract*—XML-based SOAP Web Services are a widely used technology, which allows the users to execute remote operations and transport arbitrary data. It is currently adapted in Service Oriented Architectures, cloud interfaces, management of federated identities, eGovernment, or millitary services. The wide adoption of this technology has resulted in an emergence of numerous – mostly complex – extension specifications. Naturally, this has been followed by a rise in large number of Web Services attacks. They range from specific Denial of Service attacks to attacks breaking interfaces of cloud providers [1], [2] or confidentiality of encrypted messages [3].**

By implementing common web applications, the developers evaluate the security of their systems by applying different penetration testing tools. However, in comparison to the well-known attacks as SQL injection or Cross Site Scripting, there exist no penetration testing tools for Web Services specific attacks. This was the motivation for developing the first automated penetration testing tool for Web Services called WS-Attacker. In this paper we give an overview of our design decisions and provide evaluation of four Web Services frameworks and their resistance against WS-Addressing spoofing and SOAPAction spoofing attacks.

***Index Terms*—SOAP-based Web services, WS-Security, WS-Addressing spoofing, SOAPAction spoofing, Penetration Testing Tool**

I. Introduction

Service-oriented architectures (SOAs) have been developed as a new software paradigm, which enforces software modularization and reuse. The key technology to implement SOA has become the eXtensible Markup Language (XML), surrounded with the related W3C-standards such as SOAP [4], WSDL [5], or XML Schema [6]. It has been relatively quickly realized that these architectures need to support flexible security mechanisms. Thus, the OASIS consortium has developed additional standards describing application of security mechanisms in SOAP messages (WS-Security [7]), building security policies (WS-Security Policy [8]), or exchanging authentication (SAML [9]) and authorization tokens (XACML [10]).

Unfortunately, the complexity and a large number of these standards have led to emergence of Web Services specific attacks. A short overview of some well-known Web Service specific attacks mainly taken from [11] can be seen in Table I. The most important attacks are those breaking cryptographic primitives defined in the XML messages. The so called XML Signature Wrapping attacks described by McIntosh and Austel in 2005 [12] allow an attacker to arbitrarily modify signed messages. The practical impact of these attacks has been shown by its application on Amazon EC2 SOAP and Eucalyptus cloud Web Service interfaces [1], [2] or on different SAML-based

XML Signature Wrapping	Attack on XML Encryption
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	

TABLE I
Overview of existing Web Service specific attacks.

Single Sign-On frameworks [13]. Another relevant attack in this area has been presented at CCS'11 [3]. It has been shown that it is possible to decrypt arbitrary XML ciphertexts if a server working as a plaintext validity oracle is given. The development of countermeasures against this attack is not trivial as many side-channels can be revealed by the server-side implementation [14]. In addition to the attacks on cryptographic data primitives, there exists a whole series of highly efficient Denial of Service (DoS) attacks. The adversaries could e.g. apply the HashDoS attack on the XML-structure[1] or force the server to execute expensive cryptographic algorithms.

The developers are usually familiar only with a small number of the Web Services standards and therefore, they are not able to identify vulnerabilities in the implemented Web Services interfaces. Compared to attacks like SQL injection and Cross Site Scripting (XSS) – which can be checked with a number of penetration testing tools – there are no solutions offering automated testing of XML-specific vulnerabilities. For these reasons it was decided to develop the first automated penetration testing tool for XML-based Web Services called *WS-Attacker*. In this paper, the basic design decisions for creating this modularized tool are presented and a description of the implementation and inclusion of new attacks considering the interfaces of WS-Attacker is given. Afterwards, details of two attack implementations already included in the framework are deepened: WS-Addressing spoofing and SOAPAction spoofing. The evaluation of their implementation is presented using four widely used Web Services frameworks: Apache Axis2, JBossWS native, JBossWS CXF and .NET Web Services. WS-Attacker is offered as an open-source implementation[2] to a wide range of Web Services developers.

The rest of the paper is organized according to the structure

[1]https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2012-0841
[2]http://sourceforge.net/projects/ws-attacker

delineated below: Section II gives an overview of paper relevant technologies. Section III briefly describes the WS-Addressing spoofing and SOAPAction spoofing attacks. Basic design and implementation decisions are summarized in Sections IV and V. The evaluation results are included in Section VI and the conclusion in Section VII.

II. Foundations

This section introduces the fundamentals of XML and Web Services relevant aspects for further description of the penetration testing framework.

A. XML Security

The eXtensible Markup Language (XML) [15] is a specification offering a possibility for flexible storage of tree-based data. Due to their flexibility, the XML documents gained in recent years much popularity. They are currently used for data transmission, data storage, or in case of a Web Service for function invocation calls.

Since XML documents often contain confidential and reliable data, the W3C consortium has developed standards that describe the XML syntax for applying cryptographic primitives to arbitrary XML data. The resulting standards have become XML Encryption [16] and XML Signature [17]. Using XML Encryption to XML data ensures its confidentiality. In parallel, XML Signature guarantees data integrity and authenticity. Both can be applied to arbitrary data in the document.

B. SOAP-based Web Services

SOAP is a standard which describes message exchange with a Web Service [4]. SOAP messages basically consist of an *Envelope* element with two child elements named *Header* and *Body*: The SOAP header can contain meta information such as nonces, timestamps, or XML Signatures. The SOAP body is used for storing a Web Service operation and its parameters. The concrete structure of the SOAP message used to communicate with a Web Service and the binding information is described in the *Web Service Description Language* (WSDL) [5].

SOAP does not specify any concrete transport protocol. In most cases HTTP is used. This allows to define additional data in the HTTP headers and filter the messages using standard HTTP firewalls. One example of an additional HTTP header is the *SOAPAction* field, which redundantly corresponds the SOAP body operation. According to the SOAPAction field, the firewall can decide e.g. if the message sender can execute the given operation.

An example of a SOAP message including HTTP parameters is given in Listing 1.

```
POST /webservice HTTP/1.1
SOAPAction: addUser

<soapenv:Envelope>
  <soapenv:Header/>
  <soapenv:Body>
    <addUser>
      <name>Bob</name>
    </addUser>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing 1. SOAP message transfered over HTTP with an additional SOAPAction parameter.

C. Web Services Addressing

Web Services Addressing (WS-Addressing) [18] is a standard supporting the message routing definition directly inside of the exchanged SOAP messages. This makes the routing data independent of the underlying protocol and of any transport characteristics.

An example of WS-Addressing defined in the SOAP message header gives Listing 2. Using this message the client accesses the shops' services defined in the *wsa:To* element. He executes the function *buy_article* in the *wsa:Action* element. If the article is in the store, the message can be forwarded to the billing service in the *wsa:ReplyTo* property. Otherwise, the message is forwarded to a service defined in the *wsa:FaultTo* element, which handles reordering and informs the client about the next processing steps.

```
<soapenv:Header xmlns:wsa='.../addressing'>
  <wsa:To>services/shop</wsa:To>
  <wsa:Action>buy_article</wsa:Action>
  <wsa:ReplyTo>
    <wsa:Address>services/billing</wsa:Address>
  </wsa:ReplyTo>
  <wsa:FaultTo>
    <wsa:Address>services/out_of_stock</ ↷
        wsa:Address>
  </wsa:FaultTo>
</soapenv:Header>
```

Listing 2. WS-Addressing applied in the SOAP header

D. Web Services Security

SOAP-based Web Services are commonly exchanged over HTTP. This allows to use SSL/TLS [19] as a secure transport mechanism. This mechanism however does not bring advantages if the SOAP messages are transfered over more than one endpoints. Therefore, the OASIS group has maintained the *Web Service Security* (WS-Security) [7], which specifies how to:

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

This allows to secure the messages on the message-level and protect them during the whole transport even over a large number of endpoints.

III. Web Services Attacks

The following section gives an overview of existing attack classes used against Web Services relevant for this paper.

A. Web Service Specific vs. Non-Specific Attacks

Web Services use XML-based messages sent over different protocols. They can execute operations on a remote system or access a database. The XML-based SOAP messages can contain different data giving the client access rights to a service operation or addressing a next Web Service, which should be invoked. Considering these facts, a Web service interface can be 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 or SQL Injection.
- **Specific** Web Service attacks exploit vulnerabilities on SOAP and XML. They attack the XML-parser with Denial of Service attacks, build unexpected SOAP messages, or attack the confidential data transmitted in the SOAP message.

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[3]. A good all-in-one tool for testing web applications is the *Web Application Attack and Audit Framework* (w3af)[4]. Currently, there is nevertheless 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[5].

B. SOAPAction Spoofing

SOAPAction spoofing is a Web Service specific attack [11], which misuses the *SOAPAction* parameter in the HTTP header. The basic idea of the attack can be explained using the following 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* contains the corresponding names in both fields: the SOAPAction parameter is set to *ActionA* and the name of the first SOAP body element is *OperationA*.

A SOAPAction spoofing attack changes the SOAPAction header to a different action as shown in Listing 3.

```
POST /webservice HTTP/1.1
Host: soapActionSpoofingHost
SOAPAction: "ActionB"

<soapenv:Envelope>
  <soapenv:Header/>
  <soapenv:Body>
    <OperationA/>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing 3. SOAPAction spoofing attack message

[3]http://owasp.org
[4]http://w3af.sourceforge.net/
[5]http://www.soapui.org/

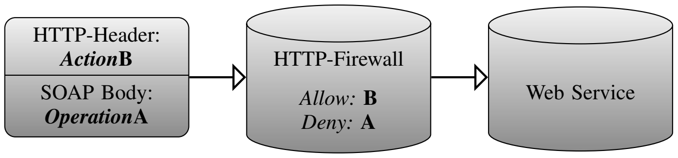


Fig. 1. Attacking a Web Service with SOAPAction spoofing.

In some cases, this message can provoke an unwanted reaction. 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 in Listing 3 is illegally allowed and will be forwarded to the Web Service, see Figure 1. 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. A real life example for this attack was published in January 2010[6]: *SourceSec Security Research* has found a vulnerability in D-Link routers, which allowed administrative access using SOAPAction spoofing.

C. WS-Addressing Spoofing

WS-Addressing spoofing is a further Web Service specific attack [11]. The idea of this attack is depicted in Figure 2: The attacker sends a SOAP request to the server containing a WS-Addressing header, which provokes the server to send 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. This will only work if the request was valid and no error occurs.
- **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. Thus, the impact by this method is more powerful, as provoking SOAP Faults is much easier as a valid request.
- **To:** The server uses a different endpoint for everything, including valid responses and SOAP Faults.

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. Thereby, 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.

[6]Hacking D-Link Routers With HNAP: http://www.sourcesec.com/Lab/dlink_hnap_captcha.pdf

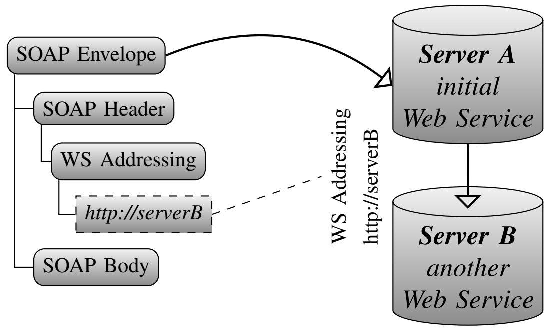


Fig. 2. Idea of WS-Addressing spoofing.

IV. Concept for a Web Services Penetration Testing Tool

This section describes the basic idea of the Web Services penetration testing tool WS-Attacker and deals with its requirements from different points of view.

A. 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**: there must be support even for prospective, not yet invented, attacks.

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.

B. 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 and especially, they do not need any knowledge about XML Security.

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. By using WS-Attacker, the company can easily check for vulnerabilities.

C. Processing Steps

For setting up the configuration to attack a Web Service, the framework will work as follows:

1) The user has to **load a WSDL**. This can be a local file or a URL.
2) The framework has to analyze it and extract all possible operations. Then, the user **selects the operation** which will be attacked.
3) The framework must be able to generate a valid request stub for the selected operation and provide input fields for message parameters.
4) The user **submits a test request**. The response to this request represents the normal state of the Web Service. Each attack plugin will get this request-response pair as a reference to build the attack vector.
5) The plugins have to be **configured** and **enabled**.
6) The framework **runs** 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. Thus, the main framework responsibilities are parsing of a WSDL and generating the SOAP request content out of it. After the attack plugins finish, the framework will present the results.

D. Framework Results

WS-Attacker distinguishes between three types of results:

1) It displays whether the attack was successful or not.
2) It gives an integer rating to view the impact of the attack.
3) It produces a kind of log entries that can be filtered by their importance level and may contain additional infos, e.g. concrete SOAP messages.

It is very important to distinguish between those kinds of results: The first one is just an indicator for the detection of a vulnerability in general, so the result will be **True** or **False**. The second one rates the potential risk of an attack. As an example, a Denial of Service attack might stop a server for several minutes or even completely so that a reboot is necessary. Furthermore, the rating can be used to describe the level of difficulty to apply the attack. The last results can be seen as an advanced log: Each plugin can produce a Log Entry which belongs to a display level. The user can then change this level to filter in and out those entries to view only the ones he is interested in.

V. WS-Attacker – Implementation Details

This section gives an overview of implementation details.

A. Overview

The general components of WS-Attacker are shown in Figure 3. 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 and manage the processing steps described in Section IV-C.
- **Plugin Architecture:** WS-Attacker can hold any number of plugins, where each plugin represents an attack.

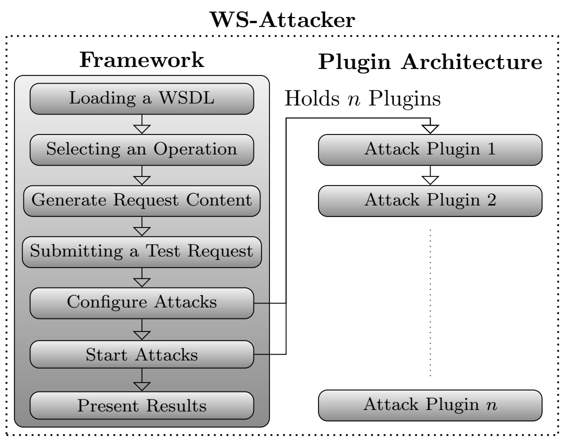


Fig. 3. General overview of WS-Attacker components and processing steps.

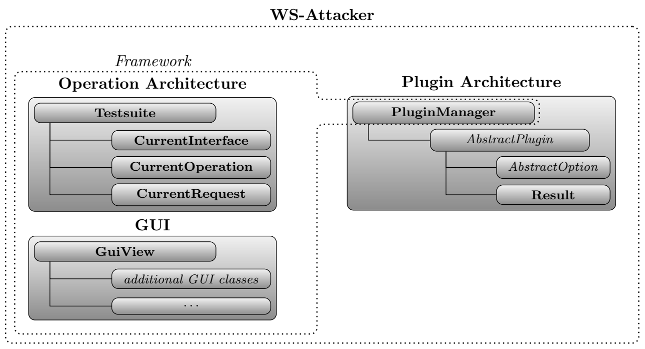


Fig. 4. The internal structure of WS-Attacker.

B. SoapUI as Back-end

As mentioned in Section IV-C, a Web Service testing framework like WS-Attacker needs to create requests from a WSDL, edit the request parameters, and send it to the server. Using Java, there are a few possibilities for doing this:

1) Implement own classes handling these steps. 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* [20]: 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 promising, 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 a single object. However, especially for Web Service attacks, one must be able to create malformed messages, e.g. create only open tags and no end tags. There are also problems when adding special characters as they are escaped automatically.
2) SAAJ does not provide a WSDL parser, so there is no possibility to create the basic SOAP 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 SOAP 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 SOAP 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. The LGPL license allows 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 SOAP request content, which allows us to manipulate them and create malformed requests.

C. Program Structure

A more detailed overview of WS-Attacker's internal structure is shown in Figure 4. Mainly, there are two parts: The *Plugin Architecture* and the *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.
- The **Plugin Architecture** represents the plugin system. This is the place for the attack plugins.

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 the *CurrentRequest*. The *CurrentRequest* will also be sent to the Web Service server to learn the normal state and the behavior on correctly formated messages. Therefore, each attack plugin can use that 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* were 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 SOAP request and response contents.

D. Attack Plugin Interface

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.

Thus, while processing the attack, information about what is happening must be saved as logging results. The user will see those results in real-time 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 the SOAP request/response contents, he chooses the *tracing* level.

Furthermore, each plugin needs to rate the attack success. Therefore, a plugin author is responsible for the following two steps:

1) Giving an integer rating for the 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.

E. Extending AbstractPlugin

In order to build attack plugins, each plugin must extend the *AbstractPlugin* interface. Listing 4 gives an overview of its methods, which can be divided into the following parts:

```
public abstract class AbstractPlugin {

    // attack identity
    public String getName(),      getAuthor()
                , getVersion(),   getDescription()
                , getCategory();

    // success interface
    public boolean wasSuccessful();
    public int getCurrentPoints();
    public int getMaxPoints();

    // result log
    final void result(level, content);

    // get the plugin options
    public OptionContainer getPluginOptions();

    // main part
    public void startAttack(testRequest
                          , testResponse);
}
```

Listing 4. The *AbstractPlugin* interface (shorted).

- **Attack identity** contains methods to describe a plugin. This includes an attack name, an author, a version and a description. In addition, a plugin category must be set, so that the GUI can sort conjugated attacks, e.g *Spoofing Attacks* or *DoS Attacks*.
- **Success interface** implements the idea mentioned in Section IV-D. The interface implements a *wasSucessful()* function as well as a success rating, which can be seen as a fraction: $\frac{\text{getCurrentPoints()}}{\text{getMaxPoints()}}$.
- Logging **Results** can be generated inside the implementation using the *result()* method. This is a helper method, so it is final.
- **Plugin Options** can be accessed by the *getPluginOptions()* method. It returns a container containing Objects which extend an *AbstractOption* interface. This let the GUI choose the correct input form, e.g. an input field or a dropdown box.
- **Main part:** A plugin is started by the *startAttack()* method. This is the place where the plugin author has to implement his concrete attack.

Most of these methods will be used as subroutines in the main part. The *startAttack()* method has two arguments which hold the request/response pair of the test request. Those can be used by the plugin for comparing the attack responses to the normal-state. Inside this method, the author should generate the results and depend if an attack was successful using the success interface.

F. Minimal Implementation

This section gives a minimal implementation example for a SOAPAction spoofing attack plugin. 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 SOAP-Action 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 implement the *startAttack()* method, which will the test request/response 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:
   a) Submit the attack request with the specified SOAP-Action.
   b) Search for the first body child in the response.
   c) Compare this child to the one from the comparison response, determine the kind of success and set the points for this.

VI. Evaluation

We tested feasibility of WS-Attacker using four widely deployed Web Services frameworks: Apache Axis2 v1.6.1[7], JBossWS Native 6.0, JBossWS CXF 7.0[8], and .NET Web Services 3.0[9]. We analyzed their resistance to the described SOAPAction and WS-Addressing spoofing attacks. Thereby, each framework was equipped with two Web Services endpoints with default configurations.

	SOAPAction spoofing		WS-Addressing spoofing	
	Success?	Rating	Success?	Rating
Apache Axis2	**True**	3/3	**True**	2/3
JBossWS native	**True**	2/3	false	0/3
JBoss CXF	**True**	2/3	false	0/3
.NET WS	**True**	3/3	false	0/3

TABLE II
All the tested frameworks revealed vulnerabilities. Apache Axis2 was vulnerable to both presented attacks.

The results of our analysis are provided in Table II. As can be seen, all tested frameworks are vulnerable to SOAPAction spoofing. Apache Axis2 and .NET WS got the maximum rating, as the Web Service always executes the operation defined in the SOAPAction header (3/3). JBossWS native and JBossWS CXF do it vice versa: They execute the operation defined in the SOAP body element, ignoring the SOAPAction header (2/3). We decided to identify this behavior also as a vulnerability as the Web Service could be compromised if it would be put behind an HTTP firewall validating SOAPAction headers.

For WS-Addressing, the only framework vulnerable for this attack in this setup is Apache Axis2. The rating for this attack is analoge to the WS-Addressing methods described in Section III-C: One point if the Web Service accepts any host for the **replyTo** method. Two points if the **faultTo** method works, because generating SOAP Faults, e.g. by sending an empty SOAP body, is much easier. This is what Axis2 does. Three points if the **to** method is accepted, which means that any response, valid or invalid, is sent to the specified host.

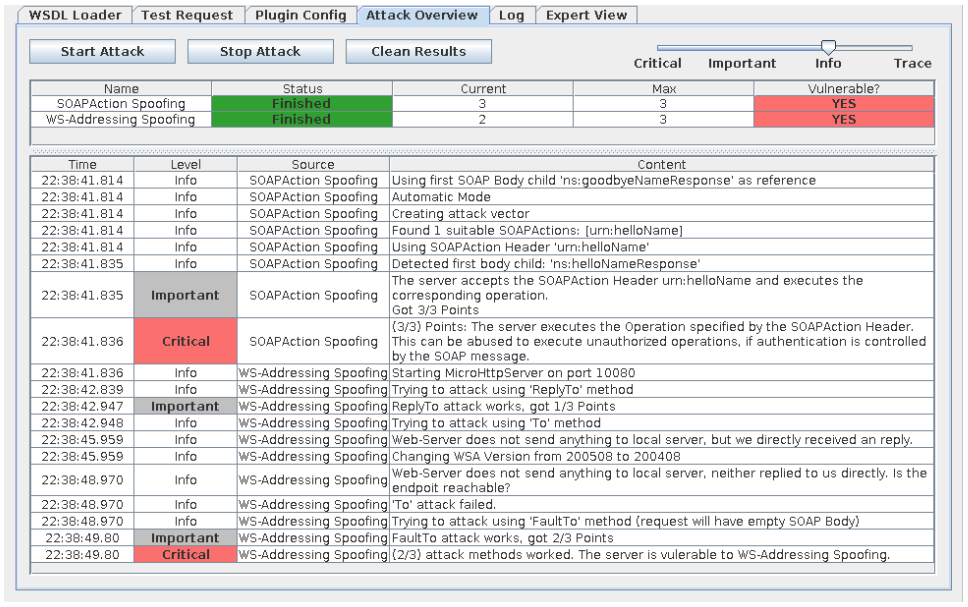
An exemplary attack result presenting the vulnerable Apache Axis2 framework is depicted in Figure 5. Later evaluation showed that this framework is vulnerable to SOAP-Action spoofing even if XML Signature is applied. Thus, the adversary could execute arbitrary function on a secured server being in possession of a single validly signed document. The only prerequisite is the equal number of parameters in the original and the malicious operation.

VII. Conclusion

In this paper we presented the penetration testing tool for Web Services called WS-Attacker. We showed our design decisions, which enabled to construct a general framework extensible with Web Service specific attack plugins. Evaluation of the implemented plugins – for SOAPAction and WS-Addressing spoofing – was executed using four widely deployed Web Services frameworks. The results proved feasibility of our approach.

The future work will cover a large number of other Web Service specific attacks, which bring additional design and implementation challenges. We mention e.g. Denial of Service and XML Signature Wrapping attacks, or attacks on XML

[7]http://axis.apache.org/axis2/java/core
[8]http://www.jboss.org/jbossws
[9]https://www.microsoft.com/download/en/details.aspx?id=14089



Name	Status	Current	Max	Vulnerable?
SOAPAction Spoofing	Finished	3	3	YES
WS-Addressing Spoofing	Finished	2	3	YES

Time	Level	Source	Content
22:38:41.814	Info	SOAPAction Spoofing	Using first SOAP Body child 'ns:goodbyeNameResponse' as reference
22:38:41.814	Info	SOAPAction Spoofing	Automatic Mode
22:38:41.814	Info	SOAPAction Spoofing	Creating attack vector
22:38:41.814	Info	SOAPAction Spoofing	Found 1 suitable SOAPActions: [urn:helloName]
22:38:41.814	Info	SOAPAction Spoofing	Using SOAPAction Header 'urn:helloName'
22:38:41.835	Info	SOAPAction Spoofing	Detected first body child: 'ns:helloNameResponse'
22:38:41.835	Important	SOAPAction Spoofing	The server accepts the SOAPAction Header urn:helloName and executes the corresponding operation. Got 3/3 Points
22:38:41.836	Critical	SOAPAction Spoofing	(3/3) Points: The server executes the Operation specified by the SOAPAction Header. This can be abused to execute unauthorized operations, if authentication is controlled by the SOAP message.
22:38:41.836	Info	WS-Addressing Spoofing	Starting MicroHttpServer on port 10080
22:38:42.839	Info	WS-Addressing Spoofing	Trying to attack using 'ReplyTo' method
22:38:42.947	Important	WS-Addressing Spoofing	ReplyTo attack works, got 1/3 Points
22:38:42.948	Info	WS-Addressing Spoofing	Trying to attack using 'To' method
22:38:45.959	Info	WS-Addressing Spoofing	Web-Server does not send anything to local server, but we directly received an reply.
22:38:45.959	Info	WS-Addressing Spoofing	Changing WSA Version from 200508 to 200408
22:38:48.970	Info	WS-Addressing Spoofing	Web-Server does not send anything to local server, neither replied to us directly. Is the endpoit reachable?
22:38:48.970	Info	WS-Addressing Spoofing	'To' attack failed.
22:38:48.970	Info	WS-Addressing Spoofing	Trying to attack using 'FaultTo' method (request will have empty SOAP Body)
22:38:49.80	Important	WS-Addressing Spoofing	FaultTo attack works, got 2/3 Points
22:38:49.80	Critical	WS-Addressing Spoofing	(2/3) attack methods worked. The server is vulerable to WS-Addressing Spoofing.

Fig. 5. Exemplary result window after penetration test execution on the Apache Axis2 framework.

Encryption. Additionally, an extension of our framework can be seen in the direction of Single Sign-On, which is typically realized using the XML-based SAML specification.

We hope that usage of WS-Attacker will force the developers to harden their services and make the area of secure Web Services more attractive and accessible also for smaller companies.

ACKNOWLEDGEMENTS

This work was sponsored by the XSpRES project funded by the Federal Office for Information Security in Germany (BSI) under the contract number 882/2010.

The authors would like to thank Meiko Jensen for many fruitful discussions and his valuable input.

REFERENCES

[1] N. Gruschka and L. Lo Iacono, “Vulnerable Cloud: SOAP Message Security Validation Revisited,” in *ICWS ’09: Proceedings of the IEEE International Conference on Web Services*. Los Angeles, USA: IEEE, 2009.
[2] J. Somorovsky, M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. L. Iacono, “All Your Clouds are Belong to us – Security Analysis of Cloud Management Interfaces,” in *The ACM Cloud Computing Security Workshop (CCSW)*, Oct. 2011.
[3] T. Jager and J. Somorovsky, “How To Break XML Encryption,” in *The 18th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2011.
[4] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, “Soap version 1.2 part 1: Messaging framework (second edition),” Tech. Rep., April 2007. [Online]. Available: http://www.w3.org/TR/soap12-part1/
[5] J. J. Moreau, R. Chinnici, A. Ryman, and S. Weerawarana, “Web services description language (WSDL) version 2.0 part 1: Core language,” W3C, Candidate Recommendation, March 2006.
[6] C. M. Sperberg-McQueen, H. S. Thompson, M. Maloney, H. S. Thompson, D. Beech, N. Mendelsohn, and S. S. Gao, “W3C xml schema definition language (XSD) 1.1 part 1: Structures,” W3C, Last Call WD, Dec. 2009. [Online]. Available: http://www.w3.org/TR/2009/WD-xmlschema11-1-20091203/
[7] A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker, “Web Services Security: SOAP Message Security 1.1 (WS-Security 2004),” *OASIS Standard*, 2006.
[8] C. Kaler and A. Nadalin, “Web Services Security Policy Language (WS-SecurityPolicy) 1.1,” 2005.
[9] S. Cantor, J. Kemp, R. Philpott, and E. Maler, “Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0,” OASIS Standard, 15.03.2005, 2005, http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf.
[10] T. Moses, “eXtensible Access Control Markup Language (XACML) Version 2.0,” *OASIS Standard*, 2005.
[11] M. Jensen, N. Gruschka, and R. Herkenhöner, “A survey of attacks on web services,” *Computer Science - R&D*, vol. 24, no. 4, pp. 185–197, 2009.
[12] M. McIntosh and P. Austel, “XML signature element wrapping attacks and countermeasures,” in *SWS ’05: Proceedings of the 2005 Workshop on Secure Web Services*. New York, NY, USA: ACM Press, 2005, pp. 20–27.
[13] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen, “On breaking saml: Be whoever you want to be,” in *Submission*.
[14] J. Somorovsky and J. Schwenk, “Technical Analysis of Countermeasures against Attack on XML Encryption – or – Just Another Motivation for Authenticated Encryption,” in *Submission, http://www.w3.org/2008/xmlsec/papers/xmlEncCountermeasuresW3C.pdf*.
[15] T. Bray, J. Paoli, E. Maler, F. Yergeau, and C. M. Sperberg-McQueen, “Extensible markup language (XML) 1.0 (fifth edition),” W3C, W3C Recommendation, Nov. 2008. [Online]. Available: http://www.w3.org/TR/2008/REC-xml-20081126/
[16] D. Eastlake, J. Reagle, T. Imamura, B. Dillaway, and E. Simon, “XML Encryption Syntax and Processing,” *W3C Recommendation*, 2002.
[17] D. Eastlake, J. Reagle, D. Solo, F. Hirsch, and T. Roessler, “XML Signature Syntax and Processing (Second Edition),” *W3C Recommendation*, 2008.
[18] D. Box, E. Christensen, F. Curbera, D. Ferguson, J. Frey, M. Hadley, C. Kaler, D. Langworthy, F. Leymann, B. Lovering, S. Lucco, S. Millet, N. Mukhi, M. Nottingham, D. Orchard, J. Shewchuk, E. Sindambiwe, T. Storey, S. Weerawarana, and S. Winkler, “Web Services Addressing (WS-Addressing),” W3C, Tech. Rep., August 2004. [Online]. Available: http://www.w3.org/Submission/ws-addressing/
[19] T. Dierks and C. Allen, “The TLS Protocol Version 1.0,” RFC 2246 (Proposed Standard), Internet Engineering Task Force, Jan. 1999, obsoleted by RFC 4346, updated by RFCs 3546, 5746. [Online]. Available: http://www.ietf.org/rfc/rfc2246.txt
[20] E. Hewitt, *Java SOA Cookbook*. O’Reilly Media, Inc., 2009.