

---

**Secologic**

# Security Tests for Web Services

Document Version 1.0 – March 2007

---

This material is provided by SAP AG for informational purposes, without representation or warranty of any kind. This material was created with support of the Fraunhofer Institut ‚Sichere Informations-Technologie‘ (SIT) in the context of the project “*secologic*”, a public funded project of the German Ministry of Economy and Technology (BMWi). We are grateful to the BMWi for supporting this project. For further information have a look at [www.secologic.de](http://www.secologic.de) or contact Rosemaria Giesecke ([rosemaria.giesecke@sap.com](mailto:rosemaria.giesecke@sap.com)).

## Index

1	Introduction .....	4
2	List of Abstract Test Cases for Web Service Security Features.....	5
2.1	Denial of Service Attack .....	5
2.2	Capture Replay Attack .....	7
2.3	Man-in-the-Middle Attack.....	9
2.4	WSDL and Access Scanning.....	11
2.5	External Entity Attacks .....	13
2.6	XML Bomb Attack .....	14
2.7	Test XML Parser on Resilience against large XML Payloads .....	16
2.8	XPath Injections.....	18
3	Supporting Tools .....	21
3.1	Parasoft SOAtest.....	21
3.2	SoapUI.....	21
3.3	Conclusion.....	22
4	References .....	23

# 1 Introduction

The importance of tests during the software development lifecycle to assure functionality, component integration and user acceptance is undisputed. Tests that aim to assert the compliance with security requirements and to check the vulnerability against certain attacks are not widely in use yet. This document takes up the section on attacks against web services in the *Web Service Security – Best Practice Guide* document ([Secologic07]) of the Secologic project and proposes abstract test cases for the threats described there. Most new threats and attacks that web services introduce stem from the improper handling of XML-based request messages or are connected to XML parser specific issues. They make up the the biggest part of this document. On the other hand, well-known threats and attacks like buffer-overflows and code injection attacks are not eliminated through the usage of web services. For more information on this kind of security threats refer to [Secologic05a] and [Secologic05b].

Section 2 offers a list of threats and attacks against web services and XML parsers accompanied by test cases descriptions that aim to verify or falsify the presence of vulnerabilities to these attacks. Section 3 introduces a commercial and an open source test toolkit and framework which not only support functional testing for web services but can also execute security tests.

## 2 List of Abstract Test Cases for Web Service Security Features

### 2.1 Denial of Service Attack

#### 2.1.1 Attack Description

##### 2.1.1.1 Initial Situation

Web services and web applications are either interfaces for other applications or perform the requested operations themselves. In both cases the services consume and use resources like CPU-time, memory, or database access.

##### 2.1.1.2 Vulnerability

The web services do not check the validity of incoming requests or they check the validity too late during processing of the request message.

##### 2.1.1.3 Attack

The attacker consumes resources in abundance by sending authorized or unauthorized requests. The requests are either very small and numerous or sporadic but with a very large payload.

##### 2.1.1.4 Threat

Authorized users may be inhibited using the web service.

##### 2.1.1.5 Threatened Protection Goals

- Availability

##### 2.1.1.6 Countermeasures

- Define and check boundaries for all incoming request parameters
- Assign quotas for all users and service threads
- Check all requests as soon as possible for authorization

##### 2.1.1.7 Comment

There are many different variants of denial of service attacks. For example capture replay attacks or buffer overflows can be used to disturb servers. In this document we focus on attacks which are realized on the SOAP level.

#### 2.1.2 Test Goal

The test intends to analyze the behavior of the web service under certain load situations. The goal is to examine how the service and the system the service is running on react, if the service has to process a large number of requests. These requests may differ in size, may have a correct or incorrect structure or may derive from authorized or unauthorized sources.

### 2.1.3 Components under Test

The test described in this document focuses on web services, so the test aims on the web service stack processing the incoming requests and the application behind the stack. Beside this, we recommend to test the full server system with all accessible network applications, as the web service stack is most often only one part of an application server system.

### 2.1.4 Testing phase

The liability for denial of service attacks depends both on the hardware and the software which processes incoming web service request. Running on a local developer workplace, the capacity to handle requests may be a lot smaller than on a full featured server system. Nevertheless, testing at an early time on a rather small sized system can be very useful to inspect the stability of the implementation under load situations and the efficiency of implemented counter measures. But it should never replace the tests to determine the liability for denial of service attacks of the productive system.

### 2.1.5 Test Personnel

The test Personnel should be qualified and authorized to test the web service with varying SOAP messages and to analyze the condition of the server system.

### 2.1.6 Prerequisites

For the test mainly two components are necessary: one component to generate the web service requests and another one to analyze the system condition. The first component may be separated in several computer systems and network connections, being able to produce an amount of data which is enough to stress the web service server. The analyzing component should also be separated in tools running on the tested system itself to log the system status and those outside the system, measuring parameters like answering delays or the amount of resources the system consumes.

### 2.1.7 Test Execution

To accomplish the test, the component for generating web service requests should start sending requests to the web service with increasing rates. There are many possible variants to mash up authorized und unauthorized messages, changing the payload size, building messages with a wrong structure or to diversify other message parameter. As always security tests demand a certain level of creativity to find out rather unexpected vulnerabilities.

### 2.1.8 Post conditions

It is obvious that every web service application has a limit for processing incoming requests. But one post condition is that the system should never reach a stage, when it does not react on incoming messages after the attack has been stopped. In other words, it should never completely hang up and should recover itself from overload situations. During overload conditions it should be impossible to access any information not accessible during normal circumstances.

## 2.1.9 Test results and interpretation

Perhaps even more important than stressing the system with messages creatively is to control the system accurately for undesirable behavior. This can be done during the test with automated tools and afterwards by inspecting the log files und recorded messages manually.

## 2.1.10 Test coverage and completeness

A test for denial of service attacks is very useful to inspect the stability of the implementation under load situations and the efficiency of implemented counter measures like authorization mechanisms. But it is nearly impossible to predict the system behavior under real attack situations exactly, because there are many other dynamic system parameters which might influence the system reaction.

## 2.2 Capture Replay Attack

### 2.2.1 Attack Description

#### 2.2.1.1 Initial Situation

Web services and web application expect requests in a specific form, which often is easy to analyze and to emulate by an attacker.

#### 2.2.1.2 Vulnerability

A web service respectively the application does not ensure incoming request are new and genuine.

#### 2.2.1.3 Attack

The attacker records a request message on the network and sends it to the service without manipulation later again, possibly many times. Another variant is to manipulate specific parts of the message and change for example the names of operations, values of parameters or the user-id.

#### 2.2.1.4 Threat

The replayed request will be processed again and potential costs will be charged to the original message sender. Manipulated messages may result in answers containing disclosed information. Many recurrences may overload the server system.

#### 2.2.1.5 Threatened Protection Goals

Under certain conditions all security goals may be threatened.

#### 2.2.1.6 Countermeasures

- All incoming request should have a distinct identifier and a timestamp, which makes it possible to compare incoming messages with those already received.
- The messages should be secured against unauthorized manipulation, for example by the use of digital signatures.

### 2.2.1.7 Comment

Man-in-the-Middle attacks as described in the following section are often based on capture replay attacks.

## 2.2.2 Test Goal

This test examines the countermeasures against Capture Replay Attacks like unique identifiers for all message exchanges and authentication mechanisms. It tries to simulate different variants of this attack and analysis the system behavior.

## 2.2.3 Components under Test

The test focuses on the web service implementation, in particular web service security components providing reliable message exchange and authorization.

## 2.2.4 Testing phase

Tests against capture and replay attacks can be done at an early stage of the development process. They help to decide which types of countermeasures are necessary for the particular type of application and to inspect the final configuration.

## 2.2.5 Test Personnel

The test Personnel should have knowledge to interpret and modify the exchanged SOAP messages.

## 2.2.6 Prerequisites

To accomplish a simulated attack, the test system should be placed at a position to capture messages from the original sender and to send these messages to the server again. It is not essential to pretend the identity of the original sender, unlike the following Man-in-the-Middle Attack scenario.

## 2.2.7 Test Execution

At first you have to collect a set of regular service request messages. Then simply try to resend the messages to the service and observe the corresponding service reaction. An analysis of the collected messages can be the basis for additional test with modified request messages.

## 2.2.8 Post conditions

After test execution, the system should be in a stable state and the integrity of consumer data should be unaffected.

## 2.2.9 Test results and interpretation

It should not be possible to use captured messages for requesting any type of unauthorized information, which is not available for the attacker anyway. Furthermore, the replayed

requests should not cause system changes which are assigned to the original sender, especially if they trigger money or other types of value transfers.

### 2.2.10 Test coverage and completeness

The test tries to simulate a capture replay attack maybe accomplished by an attacker on the runtime system later. It is virtually impossible to cover all possible attack variants, so the completeness depends on the efforts to simulate as much attacks as possible. Of course deep knowledge of the system and message formats can help to identify the worst scenarios.

## 2.3 Man-in-the-Middle Attack

### 2.3.1 Attack Description

#### 2.3.1.1 Initial situation

Web services and web application expect requests in a specific form, which often is easy to analyze and to emulate by an attacker.

#### 2.3.1.2 Vulnerability

A web service respectively the application does not ensure incoming request are new and genuine.

#### 2.3.1.3 Attack

Similar to capture replay attacks the attacker captures requests from authorized service consumers and manipulates them before he passes them on to the service. Additional knowledge about vulnerabilities of the used protocols and applications allows him to place himself between the service and the consumer. He leads both peers to believe that he is the originally addressed communication partner.

#### 2.3.1.4 Threat

Both service and service consumer believe that they use a confidential and authentic channel although the attacker is able to read and manipulate all messages.

#### 2.3.1.5 Threatened Protection Goals

- All

#### 2.3.1.6 Countermeasures

- Cryptographic keys to exchange data using a secure cipher channel
- A trusted third party to verify or exchange initial cryptographic keys
- Digital signatures to protect the integrity of messages

### 2.3.2 Test Goal

There are many ways for a potential attacker to bring himself in a position between to communication partners. For example he can use a weakness of underlying network protocols, which are not in the focus of web service security at all. This test tries to detect, if

the protection mechanisms on web service layer are working and if the attacker is able to harm the integrity, authenticity and confidentiality of SOAP messages.

### 2.3.3 Components under Test

The test targets the security components of web services on both network (SSL etc) and message layer (SOAP encryption).

### 2.3.4 Testing phase

The test can be executed independent from a specific web service implementation, because it examines the message channel on an abstract layer.

### 2.3.5 Test Personnel

Test Personnel should have knowledge on both network security and SOAP message level security.

### 2.3.6 Prerequisites

The attack system should be placed at a position between the service and the consumer in such a way as the system can capture messages, modify them, and send them to the service at any time. It is negligible how an attacker might get this position in real-life scenarios; we are just pretending that he might be able to reach it.

### 2.3.7 Test Execution

The test execution can be separated in three phases, targeting the goals to break confidentiality, integrity and authenticity.

The first step is to analyze the network traffic only and to try to find any type of confidential information in it. This may also include trials to decrypt encrypted data fields to control that the used algorithms are implemented well and key material is good enough.

Afterwards, test how the system reacts on modified messages. It should be impossible to exchange any type of information in the message undetected.

The third phase comprehends the attempt to get full control over the message conversation. This means to attempt both peers to believe that the attack system is the originally addressed communication partner. Therefore you should try to break the authorization data. If session based protocols are used which try to establish a secure tunnel, the initiation phase should be analyzed in detail.

### 2.3.8 Post conditions

It should be impossible to harm the integrity, authenticity and confidentiality of the exchanged SOAP messages.

### 2.3.9 Test results and interpretation

If an attacker is able to reach a position at the network between two communication peers, he is able to disturb the communication anyhow. This can be prevented on underlying network protocols only.

The result of this test has to be, that the attacker is unable to disclose secret information and that all attempts to modify the message exchange are being detected and notified by both service and consumer system.

### 2.3.10 Test coverage and completeness

Naturally tests of cryptographic data are very time-consuming. It is not the primary objective to detect weak algorithms or wrong implementations here, but to verify if they are applied well. Just to give a simple example: the strongest algorithm may be useless, if the web application uses the same symmetric key material for every message exchange. Our goal is to detect these small but far-reaching mistakes which make man-in-the-middle-attacks possible.

## 2.4 WSDL and Access Scanning

### 2.4.1 Attack Description

#### 2.4.1.1 Initial Situation

A WSDL document contains information about the provided operations of a web service, their input and output parameters and the URI to address them. Often WSDLs get generated automatically by the server system and are accessible over the network.

#### 2.4.1.2 Vulnerability

If WSDL files get generated, they are generated for all the web service operations most times. Maybe even for those which should not be accessible in public. The operations are also still accessible if their URI is wiped out from the WSDL file.

#### 2.4.1.3 Attack

The attacker tries to guess additional web service operations based on the information provided in WSDL files. For example, if there is an operation called “getItem”, there might be another hidden one called “setItem”.

#### 2.4.1.4 Threat

The attacker can execute unauthorized operations which are not intended to be public. He may use the web service operations to gain access to confidential information or to modify data of applications behind the web service.

#### 2.4.1.5 Threatened Protection Goals

- Potentially all, depending on the accessible operations

#### 2.4.1.6 Countermeasures

- Implement effective access control mechanisms for the web service operations to enforce access control policies. Wiping out WSDL information is not a sufficient solution.
- Control information provided in WSDL files and compare them to the access control policy.

## 2.4.2 Test Goal

The test mainly contains a comparison between the provided WSDL information, the reachable web service interfaces and the desired access policy.

## 2.4.3 Components under Test

The analyzed components are the WSDL definition and the web service interfaces.

## 2.4.4 Testing phase

A final test can be done if all interfaces are implemented and the WSDL file intended to be published is generated.

## 2.4.5 Test Personnel

The test Personnel should have knowledge to understand the WSDL definitions and to audit the interfaces provided by the web service server system.

## 2.4.6 Prerequisites

The prerequisites are rather simple, compared to the other tests described in this document. The WSDL analysis can be done offline, for the interface examination access to server administration tools may be useful.

## 2.4.7 Test Execution

At first it is very important to know the intended access policy. It should be absolutely clear, which interfaces ought to be provided to whom. Tools to model formal policies can be helpful for these definitions.

Then compare the policies with WSDL document and the provided interfaces. To get a general idea of the server interfaces you can use the server admin tools. Supplementary tests, if unauthorized requests are getting rejected, should be done in addition.

## 2.4.8 Post conditions

All efforts to reach interfaces unauthorized have to be rejected.

## 2.4.9 Test results and interpretation

The test should result with the conclusion, that there are no differences between the implemented interfaces, the corresponding WSDL definitions and the desired access policy. It is absolutely not a good solution to use WSDL files to enforce access policies by wiping out reachable interfaces.

## 2.4.10 Test coverage and completeness

The effort for this test highly depends on the complexity of the tested system. It might be helpful to divide the system in different parts and to test as early as possible during the system design. In this way, a full coverage of the system interfaces is possible.

## 2.5 External Entity Attacks

### 2.5.1 Attack Description

#### 2.5.1.1 Initial Situation

Nearly all protocols in the web service world are based on XML. SOAP messages, WSDL descriptions, UDDI entries and often system configuration files are XML documents. XML documents have the ability to reference other XML documents on the network using an URI.

#### 2.5.1.2 Vulnerability

The integrity of URIs inside WSDL, SOAP or other XML-document is not protected or the referenced location is not protected against manipulation.

#### 2.5.1.3 Attack

An attacker manipulates the URI or the document referenced with the URI to insert malicious content.

#### 2.5.1.4 Threat

The attack may result in access to confidential information or as entry point for denial of service attacks.

#### 2.5.1.5 Threatened Protection Goals

- Confidentiality
- Availability

#### 2.5.1.6 Countermeasures

- Generally be careful when referencing external XML-documents.
- Protect the integrity of referenced documents using XML-signatures.

### 2.5.2 Test Goal

The test tries to determine if a web service application is vulnerable to external entity attacks.

### 2.5.3 Components under Test

Target of this test are all components which open and interpret URIs inserted in messages. This may be the web service application but also the xml parser itself.

### 2.5.4 Testing phase

The XML parser implementation can be tested independent from an implementation at any time, the implementation itself early during development.

### 2.5.5 Test Personnel

The test personnel should have knowledge of selective URI modifications and the XML interpretation techniques.

## 2.5.6 Prerequisites

The first step is to analyze which message parts of web services requests contain URIs and the preparation of a test system which sends generated or captured and modified messages to the web service.

## 2.5.7 Test Execution

During the test, try to find out how the service reacts on modified URIs inside web service requests. Variants are for example malformed URIs, URIs with unreachable destinations, malformed documents or documents containing other malicious content. Compare this to the following tests.

## 2.5.8 Post conditions

The system should reject requests with malformed URIs or URIs containing links to non suitable documents.

## 2.5.9 Test results and interpretation

If messages do not have any kind of integrity protection, it is easy for an attacker to modify messages containing URIs. So the first step to avoid URI manipulation is to integrate an integrity protection, for example using signatures.

To automatically decide if an inserted URI links to the originated or a malicious document can be very ambitious. Some simple filter rules inside the web service application may prevent the system from interpreting and executing unintended activities.

## 2.5.10 Test coverage and completeness

The test can cover all provided services which except requests that contain URIs. But it is not easy to avoid an external entity attack in some cases, when the use of cryptographic methods is not suitable. The next tests help to limit the threat of successful inserted XML code.

## 2.6 XML Bomb Attack

### 2.6.1 Attack Description

#### 2.6.1.1 Initial Situation

XML is the language of the web services. Service descriptions, messages, registry entries and more are specified as XML documents.

#### 2.6.1.2 Vulnerability

Some XML parsers have a weakness in their DTD handling that might be utilised by an attacker to incapacitate the parser. Sending SOAP requests to such parsers with special inline DTDs might cause enormous memory consumption.

### 2.6.1.3 Attack

The attacker sends an XML requests to the web service that carries a bad DTD description. This description contains element references that explode exponentially to the number of references when interpreted by an XML parser that is vulnerable to this kind of attack.

### 2.6.1.4 Threat

Using rather small messages an attacker is able to consume huge parts of the memory on the service side and thus can undermine the services availability.

### 2.6.1.5 Threatened Protection Goals

- Availability

### 2.6.1.6 Countermeasures

- Use SOAP stacks with XML parsers which do not interpret DTD code in XML documents.

## 2.6.2 Test Goal

This test aims to verify that the XML parser under test is not vulnerable to harmful DTD code with exponential growing memory consumption.

## 2.6.3 Components under Test

XML Parser of the web service stacks.

## 2.6.4 Testing phase

Since this test aims at the XML parser and since it does not involve any specific web service or web service feature, it can be executed very early in the software lifecycle. This test can take place in the evaluation phase before the design phase when existing technologies and software platforms are reviewed.

## 2.6.5 Test Personnel

The test personnel do not need special skills and training.

## 2.6.6 Prerequisites

If the XML parser can be run stand alone or if it has an API, no additional set up is necessary. Otherwise set up the web service runtime environment.

## 2.6.7 Test Execution

- 1 Build a simple SOAP document with an empty header and body and feed it into the XML parser.
- 2 Do the same with the XML document at the end of this section.
- 3 Take the time for the execution of both parsing processes.

The following XML fragment contains the XML bomb. If the XML parser respects DTD the entity reference in the SOAP:envelope element will be expanded to 2 power 100 times the word BIG.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE SOAP-ENV:Envelope [
  <!ENTITY x0 "BIG">
  <!ENTITY x1 "&x0&x0">
```

```

<!ENTITY x2 "&x1&x1">
. .
<!ENTITY x99 "&x98&x98">
<!ENTITY x100 "&x99&x99">]>
<SOAP:Envelope entityReference="&x100"
  xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP:Body>
    . . .
  </SOAP:Body>
</SOAP:Envelope>

```

### 2.6.8 Postconditions

If the XML parser is not susceptible for DTD bombs the execution of each parser invocation takes constant amount of time.

### 2.6.9 Test results and interpretation

Test Result		Interpretation
Time for test execution takes constant amount of time.	Ok	XML parser is not susceptible for XML bombs
Time for parsing the SOAP document with the XML bomb is huge.	Fail	The parser is vulnerable to XML bombs.

### 2.6.10 Test coverage and completeness

The test goal to verify or falsify if the given XML parser is vulnerable to DTD bombs is fully covered by this test. This test does not cover other Denial-of-Service attacks on the XML parser or the web service stack.

## 2.7 Test XML Parser on Resilience against large XML Payloads

### 2.7.1 Attack Description

#### 2.7.1.1 Initial Situation

SOAP messages have to be completely parsed by the XML parser to validate if they are well-formed. This consumes computation power and memory, especially if a DOM parser is used.

#### 2.7.1.2 Vulnerability

Some XML parsers build a DOM object while they are parsing a new XML document. This can be a very expensive operation concerning time and memory.

#### 2.7.1.3 Attack

The attacker sends many SOAP requests of huge size to the server.

#### 2.7.1.4 Threat

The attacker is able to undermine the service availability with very simple means. Since some parsers need to load the complete XML document into the parser before they check its validity such an attack may even be successful for an unauthorized attacker who sends wrong and meaningless messages.

### 2.7.1.5 Threatened Protection Goals

- Availability

### 2.7.1.6 Countermeasures

- Use a SAX parser if possible.
- Configure the service to only accept SOAP messages with at least one small header field that has to be signed isolated from the rest of the messages. Use WS policy constructs to demand such a field and such a signature right at the beginning of the SOAP message. This way unauthorized messages can be detected early without the need of parsing the whole document.

## 2.7.2 Test Goal

This test aims to verify that the XML parser under test is able to deal with large XML payloads.

## 2.7.3 Components under Test

Test the XML parser of the web service stack.

## 2.7.4 Testing phase

Since this test aims at the XML parser and since it does not involve any specific web service or web service feature, it can be executed very early in the software lifecycle. This test can take place in the evaluation phase before the design phase when existing technologies and software platforms are reviewed.

## 2.7.5 Test Personnel

The test Personnel does not need special skills and training.

## 2.7.6 Prerequisites

A script to generate arbitrary XML documents or SOAP messages is needed. A system monitoring tool that can monitor the CPU-time and memory consumption of single processes helps to evaluate the behavior of the XML parser. Furthermore you have to set up the web service runtime environment if you can not access the XML parser in a standalone fashion.

## 2.7.7 Test Execution

- 1 Generate valid and invalid XML documents or SOAP messages of different sizes from small to huge.
- 2 Feed these documents into the parser; respectively send the messages to a dummy web service in the web service stack.
- 3 Monitor the consumption of CPU-time and memory that the XML parser process has for the various document sizes.

## 2.7.8 Post conditions

If the XML parser is not susceptible for large payload attacks its resource consumption will grow proportional to the increased document size but it will not show disproportionate or even exponential growth. Especially the additional memory consumption will not attain a multiple of the length of the document.

## 2.7.9 Test results and interpretation

Test Result		Interpretation
Memory and CPU-time consumption grows slowly with size of the XML documents.	Ok	XML parser is not susceptible for large payload attacks.
Time for test execution grows disproportional with increased DTD size.	Fail	The parser is vulnerable to large payload attacks.

## 2.7.10 Test coverage and completeness

The test goal is to verify or falsify if the given XML parser is vulnerable to large payload attacks. A single test or a group of tests can hardly deliver enough and significant data to draw a clear conclusion. This is due to the fact any XML parser as any other component, too, can be pushed to and over its capabilities. How fast and when this borderline is reached not only depends on the parsers implementation and memory management but also on other factors like the available system memory and its computation capabilities as well as what other processes are competing for resources at the same time. Therefore only a comparing study of many different XML parsers working in the same environment under the same constraints can provide the right information to decide which XML parser offers the best protection against large payload attacks.

## 2.8 XPath Injections

### 2.8.1 Attack Description

#### 2.8.1.1 Initial Situation

The XML Path language (XPath) is a language to form quick queries over XML documents without the need to parse the whole structure into the memory. Web applications often construct XPath query strings from user input.

#### 2.8.1.2 Vulnerability

The input parameters submitted by the user via SOAP messages are accepted unchecked. Especially, the parameters are not checked to be free of escape characters and escape sequences that precede the code fragments the attacker wants to have executed.

#### 2.8.1.3 Attack

The attacker uses skillfully formed input parameters to infiltrate own XPath code into the query code on the server side. The input strings contain escape characters that separate the parameter values from the code fragments. These code fragments may be treated as server side code when the input parameters are just copied into the query string by the application.

#### 2.8.1.4 Threat

When the infiltrated code fragments are well-engineered their execution may produce service responses that reveal confidential information.

#### 2.8.1.5 Threatened Protection Goals

- Confidentiality

### 2.8.1.6 Countermeasures

- Use whitelists and filter methods to only allow valid input data, respectively to check input parameters for escape sequences.
- Establish code review and static code analysis to discover unfiltered parameter flow from the client side into the XPath queries.

## 2.8.2 Test Goal

This test aims to verify that a web service or the application behind the web service interface is not vulnerable to XPath code injection.

## 2.8.3 Components under Test

The test checks the functionality of the input validation component of the web service respectively the application if such a component exists.

## 2.8.4 Testing phase

The earliest this test can and should be executed is in the software development phase when the access interface to the application behind the web service is programmed. Such a test can be written by the developers themselves as unit tests that assert that no escape characters are accepted by the input validation methods. In this case the test can be written as a white box test.

As a black box test this test can be executed during the system integration phase of the software development process.

## 2.8.5 Test Personnel

The test Personnel should have some knowledge about XPath and code injection attacks in general to perform meaningful tests. It is necessary to construct several different XPath code fragments if the test is supposed to find out what information might be revealed if XPath code injection vulnerability exists. Especially, if this test is to be executed as a black box test.

If the vulnerability in general is to be tested no special skills are needed. See the section 2.8.7.

## 2.8.6 Prerequisites

Here, we only consider black box tests that check for the general vulnerability of the application. The web service has to be running and accessible and the client side offers either a programmatic or a web interface that accepts string input parameters.

## 2.8.7 Test Execution

We describe the test execution for the general vulnerability test. More sophisticated test goals like trying to gain access to certain fields of the XML document need special tester skill to handcraft input parameters.

We also assume that the input parameters are strings that at the server side will be compared against some attributes of some elements in the XML document.

1 Provide some strings that contain string escape characters and attribute field delimiters.

Examples may be

```
' ] or 1=1 or /something[arbitrary='  
*' or ''='
```

It is important to copy that quote marks since these escape characters work as delimiters for the strings that will be concatenated at the server side. You can also try to replace the single quotes with double quotes.

- 2 Pass these strings as input parameters to the client side interface and send the corresponding request separately to the web service.
- 3 Check the results and interpret them conforming to the table in the section *Test results and interpretation*.

### 2.8.8 Post conditions

If the application on the server side or its interface with the web service validates incoming parameters an error message should be returned in turn. Depending on the implementation of the validation process the response may even be suppressed and illegal requests silently be dropped. That means no response may be a valid post condition.

It is very important that the mentioned error message does not contain data that reveals information about the internals of the application. This would be the case if the stack trace of an exception was returned. Thus it is advisable to return a general error message like *Illegal Argument* or alike.

### 2.8.9 Test results and interpretation

If the test fails for just one of the given input parameters the web service is vulnerable to an XPath code injection attack.

Test Result		Interpretation
A general error message like <i>Illegal Argument</i> that indicates the parameter value has been validated.	OK	The server side has validated the input and filters input parameters with escape sequences.
No response at all.	OK	This very likely means that the validation procedure has recognized illegal operands. But it can also mean that the XPath implementation can not deal with syntactically wrong query strings and crashed.
Error message that says a syntactically invalid query has been proposed.	Fail	This means the escape characters have not been filtered out but passed to the XPath query engine that could not handle the query.
Response message, eventually with meaningless content.	Fail	The web service respectively the application do not check input parameters for escape sequences and thus for code injection. Check section 2.8.10 for possible exceptions from this interpretation.

### 2.8.10 Test coverage and completeness

The test goal is to verify or falsify if the given web service, respectively the application behind the web service interface is vulnerable against XPath code injection attacks. A test result of *Fail* does not necessarily mean that the server side does not validate the input. It is possible that escape sequences with empty strings are just ignored without raising an error. This case cannot be easily detected with a black box test.

## 3 Supporting Tools

### 3.1 Parasoft SOAtest

:

SOAtest is a commercial test suite for web service applications. Until 2005 the application was offered under the name SOAPtest, but it was renamed because of its wider range of supported protocols. It can be used for functional testing of web services as well as looking for security vulnerabilities.

Key features for web service security testing are:

- **Policy Enforcement**  
Management of web service policies, including analysis of WSDLs and SOAP artifacts as well as semantic validation tests.
- **Load and Performance Testing**  
Simulation of virtual users to produce extreme loads and tests from different locations.
- **Security Penetration Testing**  
SOAtest automatically generates tests to perform security penetration testing. It simulates attacks and can help to search for vulnerabilities like XPath injections, XML bombs, external entities or large XMLs payloads.
- **Test Management**  
Support for test case management correlated with requirements within a requirements management or a bug tracking system and support for automated test executions.

More Information can be found at the developer web site:

<http://www.parasoft.com/jsp/products/home.jsp?product=SOAP&itemId=101>

### 3.2 SoapUI

SoapUI is a free and open source desktop application for functional, load and compliance testing of web services. It supports testing both interactively in soapUI or automated during build or integration processes using the soapUI command-line tools.

The following key features for inspecting web services are currently available:

- Import of WSDLs and automatic generation of requests from associated schema
- Create test cases containing unlimited number of requests to imported web service and multiple service endpoints.
- Support for Basic, Digest, WS-Security authentication.
- SOAP response can be asserted for associated schema compliance, XPath expression content matching, etc.
- Values can be transferred between requests using XPath expressions (for example to transfer session IDs).

- Creation of load tests for test cases, containing configurable load strategies.
- Assertion of load test results continuously for performance and functionality surveillance.
- Behavioral diagrams allow real time analysis of performance statistics.

SoapUI can be used as a standalone application, but there are also IDE-plugins available for eclipse, IntelliJ IDEA, NetBeans and a specialized eclipse-plugin for JBossWS.

Developer web site:

<http://www.soapui.org/>

### 3.3 Conclusion

Both tools, SOAtest and soapUI, help to accomplish the tests described in chapter 2. They assist the test personnel with automated generation of test cases and present a basis of mechanisms for web service security testing. Integrated, ready-to-use security test cases may help to detect the most common web service vulnerabilities under certain conditions, but they can not replace manually adapted test cases.

The development of a secure web services requires an analysis of the application itself already during design time, to identify possible vulnerabilities early and to integrate solutions and counter measurements. Test tools can be used most effective if the test cases are designed to the need of the specific implementation. They can help to confirm the defined security requirements and that the security policies are implemented as desired. Furthermore, black box tests or attack simulations help to find out if the implemented security solutions are sufficient and help to reduce the risk of unconsidered vulnerabilities.

## 4 References

[Secologic05a] Projekt Secologic, Programmiersprachen und Sicherheit, 2005-06,  
<http://www.secologic.org/languages>

[Secologic05b] Projekt Secologic, Angriffe und Lösungen, 2005,  
<http://www.secologic.org/attacks>

[Secologic07] Web Service Security – Best Practice Guide, Projekt Secologic, Februar 2007,  
<http://www.secologic.org/languages#webservices>