

# Secologic

## Java Web Application Security

### Best Practice Guide

Document Version 3.0 – September 2006

**NEW:** With **Test Procedures** for every Category

**CHANGED:** 2 Category Names:

- *Ressource Injection* to ***Ressource Tampering***
- *Logic Errors* to ***Missing Input Validation***



---

This material is provided by SAP AG for informational purposes, without representation or warranty of any kind. This material was created 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)

## Contents

<i>Purpose</i>	6
<i>Target Group</i>	6
<i>About this Document</i>	6
<i>Test Procedures in this document</i>	8
<i>Platform solutions described in this document</i>	10
<i>Classification table</i>	10
<b>1 SQL Code Injection</b>	<b>12</b>
1.1 <i>Introduction</i>	12
1.1.1 <b>Variants</b>	13
1.1.2 <b>Damage in the case of Non-Compliance</b>	15
1.2 <i>Prevention, Countermeasures, Solutions</i>	16
1.2.1 <b>Description</b>	16
1.2.2 <b>Platform Limitations or Extensions</b>	16
1.2.3 <b>Solution Variants</b>	16
1.3 <i>Test Procedure</i>	20
1.3.1 <b>Outline</b>	20
1.3.2 <b>Test Execution</b>	21
1.3.3 <b>Interpretation of Test Results</b>	21
1.3.4 <b>Special Tool Support</b>	21
1.4 <i>References</i>	22
<b>2 Cross Site Scripting (XSS)</b>	<b>23</b>
2.1 <i>Introduction</i>	23
2.1.1 <b>Variants</b>	23
2.1.2 <b>Damage in the case of Non-Compliance</b>	26
2.2 <i>Prevention, Countermeasures, Solutions</i>	27
2.2.1 <b>Description</b>	27
2.2.2 <b>Platform Limitations or Extensions</b>	28
2.2.3 <b>Solution Variants</b>	28
2.3 <i>Test Procedure</i>	33
2.3.1 <b>Outline</b>	33
2.3.2 <b>Test Execution</b>	33
2.3.3 <b>Interpretation of Test Results</b>	34
2.4 <i>References</i>	35
<b>3 Cookie Security</b>	<b>36</b>
3.1 <i>Introduction</i>	36
3.1.1 <b>Variants</b>	36
3.1.2 <b>Damage in the case of Non-Compliance</b>	40
3.2 <i>Prevention, Countermeasures, Solutions</i>	40
3.2.1 <b>Description</b>	40
3.2.2 <b>Platform Limitations or Extensions</b>	40
3.2.3 <b>Solution Variants</b>	40
3.3 <i>Test Procedure</i>	43

3.3.1	<b>Outline</b>	43
3.3.2	<b>Test Execution</b>	44
3.3.3	<b>Interpretation of Test Results</b>	45
3.4	<i>References</i>	45
<b>4</b>	<b>Resource Tampering</b>	<b>46</b>
4.1	<i>Introduction</i>	46
4.1.1	<b>Variants</b>	46
4.1.2	<b>Damage in the case of Non-Compliance</b>	50
4.2	<i>Prevention, Countermeasures, Solutions</i>	50
4.2.1	<b>Description</b>	50
4.2.2	<b>Platform Limitations or Extensions</b>	50
4.2.3	<b>Solution Variants</b>	50
4.3	<i>Test Procedure</i>	54
4.3.1	<b>Outline</b>	54
4.3.2	<b>Test Execution</b>	55
4.3.3	<b>Interpretation of Test Results</b>	55
4.4	<i>References</i>	56
<b>5</b>	<b>Code Injection</b>	<b>57</b>
5.1	<i>Introduction</i>	57
5.1.1	<b>Variants</b>	57
5.1.2	<b>Damage in the case of Non-Compliance</b>	59
5.2	<i>Prevention, Countermeasures, Solutions</i>	59
5.2.1	<b>Description</b>	59
5.2.2	<b>Platform Limitations or Extensions</b>	59
5.2.3	<b>Solution Variants</b>	59
5.3	<i>Test Procedure</i>	61
5.3.1	<b>Outline</b>	61
5.3.2	<b>Test Execution</b>	61
5.3.3	<b>Interpretation of Test Results</b>	62
5.4	<i>References</i>	62
<b>6</b>	<b>Information Disclosure</b>	<b>63</b>
6.1	<i>Introduction</i>	63
6.1.1	<b>Variants</b>	63
6.1.2	<b>Damage in the case of Non-Compliance</b>	64
6.2	<i>Prevention, Countermeasures, Solutions</i>	65
6.2.1	<b>Description</b>	65
6.2.2	<b>Platform Limitations or Extensions</b>	65
6.2.3	<b>Solution Variants</b>	65
6.3	<i>Test Procedure</i>	66
6.3.1	<b>Outline</b>	66
6.3.2	<b>Test Execution</b>	67
6.3.3	<b>Interpretation of Test Results</b>	68
6.4	<i>References</i>	68
<b>7</b>	<b>Unreleased Resources</b>	<b>69</b>

---

7.1	<i>Introduction</i>	69
7.1.1	<b>Variants</b>	69
7.1.2	<b>Damage in the case of Non-Compliance</b>	71
7.2	<i>Prevention, Countermeasures, Solutions</i>	71
7.2.1	<b>Description</b>	71
7.2.2	<b>Platform Limitations or Extensions</b>	71
7.2.3	<b>Solution Variants</b>	71
7.3	<i>Test Procedure</i>	74
7.3.1	<b>Outline</b>	75
7.3.2	<b>Test Execution</b>	75
7.3.3	<b>Interpretation of Test Results</b>	76
7.3.4	<b>Special Tool Support</b>	76
7.4	<i>References</i>	77
<b>8</b>	<b>Missing Input Validation</b>	<b>78</b>
8.1	<i>Introduction</i>	78
8.1.1	<b>Variants</b>	78
8.1.2	<b>Damage in the case of Non-Compliance</b>	79
8.2	<i>Prevention, Countermeasures, Solutions</i>	79
8.2.1	<b>Description</b>	80
8.2.2	<b>Platform Limitations or Extensions</b>	80
8.2.3	<b>Solution Variants</b>	80
8.3	<i>Test Procedure</i>	82
8.3.1	<b>Outline</b>	82
8.3.2	<b>Test Execution</b>	83
8.3.3	<b>Interpretation of Test Results</b>	83
8.4	<i>References</i>	83

## Purpose

This document is a collection of best practice guides for several security topics with a focus on Java web applications and, more precisely, Java Servlets and JSPs. It describes common security errors and weaknesses to watch out for as well as approved procedures so that your application functions “securely”.

## Target Group

The target group of this documentation are Java developers with security concerns for dealing with several security topics, like cross site scripting, input validation etc.

## About this Document

Every section will introduce a security vulnerability by showing corresponding insecure Java code and possible attacks. This is necessary for a better understanding of attacks on an application’s security. Furthermore, for every security topic discussed, this document explains the best practice solution for preventing corresponding security failures.

Some solutions apply not only to web applications or Servlets but to Java in general, like the prevention of “SQL Code Injection”. See the section “Platform solutions described in this document” for more details.

In the following discussion, we classify security topics by common attack names. The name stands for a security category like “Cross Site Scripting”. Every category represents a separate section of this document. We consider that this is the best way to summarize several well-known security topics into one category.

As this cannot cover the entire variety of topics in the area of IT security we have also constructed a **Classification table**. This table provides a clear overview of the correlation of well-known topics on IT security to the sections of this document. Moreover, the table includes our definitions of the treated topics which include the categories (e.g., cookie security) in the document. We only use these categories throughout the entire document with the given meaning.

Every section has the following structure and subsections:

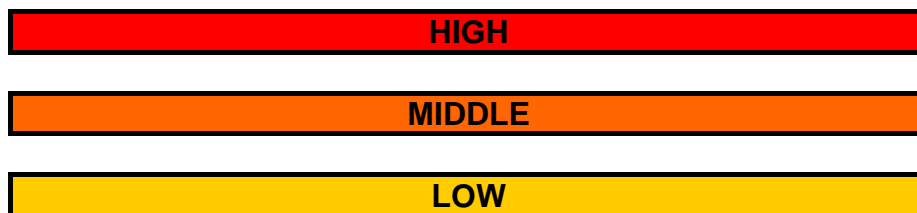
X	Category	
X.1	Introduction	
X.1.1	Variants	
X.1.1.1	Attack Variant A	
X.1.1.2	...	
X.1.2	Damage in the case of Non-Compliance	
X.2	Prevention, Countermeasures, Solutions	
X.2.1	Description	
X.2.2	Platform Limitations or Extensions	
X.2.3	Solution Variants	
X.2.3.1	Solution A	
X.2.3.1.1	Requisites	
X.2.3.1.2	Procedure	
X.2.3.1.3	To be avoided	
X.2.3.1.4	More information	
X.2.3.1.5	Example of good code	
X.2.3.2	...	
X.3	Test Procedure	
X.4	References	

(Red: New in Version 3)

Each section starts with a summary table which gives a brief overview of the corresponding security topic. The table looks like this:

Field	Meaning
Brief description of related security attacks	Summarizes briefly the main problems of the security topic
This topic is related to	Lists general programming areas (e.g., “database access”) which are affected by the topic
Degree of severity	<b>SEE BELOW</b>
Consequential and potential damage	Enumerates possible damage if the security topic is ignored for the application development
Affected packages (among others)	Lists the most important Java packages which are affected by the topic
WHEN NOT TO READ	Lists the cases for which the topic is clearly not relevant

The degree of severity of the security flaw provides a first impression of the general importance of the section’s topic. In this document, we use a simple scale with three levels. This scale is not universally binding and is only based on our general experience:



The summary table is followed by the first subsection which contains an introduction to the topic and a description of the topic in general. Important and known attack variants are highlighted by examples. This includes variants for bad source code and examples of resulting attack(s) (Introduction). After this, we give a general introduction about countermeasures and solutions which are the best practices to prevent the respective vulnerabilities. In addition, we describe the requirements for concrete solutions and we describe each solution’s procedure in detail, to help prevent the corresponding security flaws in applications. This includes also the demonstration of implementation strategies and examples of good source code as far as possible (Prevention, Countermeasures, Solutions). The next subsection contains test instructions which allow the review of existing application code to ensure that a certain application is not vulnerable against attacks described. Please note that we describe this section in more detail below (Test Procedure). Each section closes with a subsection which contains all relevant references for the respective topic (References).

### ***Test Procedures in this document***

Every section of this document describes a security topic, corresponding vulnerabilities and attacks, as well as countermeasures and solutions to enable the development of secure applications. We then present several test procedures for each topic which make it possible to evaluate Java applications and Java code with respect to the best practices described. We recommend that the test procedures are executed by security experts or experienced developers following the best practices described in this document.

Each test procedure consists of short instructions to check and verify an application's source code against the respective security flaws. Thus, these test procedures are conceived to be performed as manual code reviews. Nevertheless, most development projects nowadays provide several tools for developers which can be used to assist and support the execution of a test procedure, such as source code scanners. However, we only briefly mention possible tool support for a test procedure, as the use of tools occurs on very individual circumstances, in practice. Developers can feel free to implement the given test procedures in their individual tools, but for general usability reasons, the test procedures are designed as manual code reviews.

Please note that we do not consider dynamic test procedures in this document. For instance, penetration tests on web applications, or the development of JUnit test cases, etc., are omitted completely.

The test procedures of each section have the following structure and subsections:

- X.3 Test Procedure
  - X.3.1 Outline
  - X.3.2 Test Execution
    - X.3.2.1 Special Cases in Practice
  - X.3.3 Interpretation of Test Results
  - [X.3.4 Special Tool Support]

Every test procedure of the respective security topic is treated in a separate subsection (Test Procedure). Each test procedure starts with an outline which briefly summarizes the purpose of the test. This subsection includes a table which summarizes the most important test attributes (Outline). The next subsection contains the test instructions (Test Execution). Since the instructions are designed for any kind of Java web application, the instructions are kept general. However, we try to explain circumstances which might possibly occur and which might be relevant for the test in practice (Special Cases in Practice). The next subsection describes the relevant issues in the test results and how to proceed with these results (Interpretation of Test Results). Some test procedures include an additional subsection called "Special Tool Support". This subsection describes briefly the use of a freely available Open Source static analyzer for Java, called "FindBugs"<sup>1</sup>, to support the corresponding test procedure. FindBugs provides a few bug detectors for security.

---

<sup>1</sup> <http://findbugs.sourceforge.net/>

The table of the outline subsection which shows the most important test attributes looks like this:

Field	Meaning
Platform	The Java platform the test is applicable to
Operating Systems	The operating system the test is applicable to
Type of test	Describes briefly the kind of test which has to be performed according to the test procedure
When should be tested	<p>Here, we suggest when a test procedure should be executed during the development phase of an application and what we regard as the earliest possible point in time for a test execution.</p> <p>The tests are not applicable without existing source code. Thus, they cannot be executed in the design phase of an application, for instance. Usually, all tests can be applied from the initial to the final implementation phase. For instance, the test procedures might be executed on the first prototype of an application which even cannot be compiled (initial implementation status). They also might be executed during implementation, when the application becomes compiled or is considered to be completed (final implementation status).</p>
Tool support	Describes briefly our assessment of the possibilities of using additional and individual tools to support the execution of the respective test procedure. The most common possibility here might always be to implement the corresponding test instructions in individual tools.

## Platform solutions described in this document

Category	Solution offered	
	Java	J2EE
Code Injection	O	O
Cookie Security	O	O
Cross Site Scripting	O	O
Information Disclosure	O	O
Missing Input Validation	O	O
Resource Injection	O	X
SQL Code Injection	X	X
Unreleased Resources	X	X

### Legend

- O: Extra action necessary
- X: Solution exists and is provided by the corresponding platform

## Classification table

In the following we define and describe common categories of IT security. These are only used in the given meaning in the entire document. Moreover, the chosen categories (e.g., cookie security) of this document and thus the document's sections and their relation to other security topics are explained.

Category	Definition for the area of IT security	Related to
<i>Code Injection (COD)</i>	The injection of system and script commands into a web application or an application's server. This kind of attack mostly applies to server side script languages like PHP or Perl.	INP PAT RES
<i>Cookie Security (COO)</i>	This category includes several security vulnerabilities based on cookies, e.g., unfiltered cookie content, cookie poisoning, and flow injection via cookies. In a broader sense, this section is related to session management.	INP
<i>Cross Site Scripting (XSS)</i>	Here, the attacker inserts code into a URL or link. The malicious URL must be executed by a web application's user to have an effect. Misleading users to execute such URLs is supported by the URL itself which looks like a trustworthy URL to the application. This only works when the application is vulnerable to XSS. The result can be, e.g., the execution of malicious script (e.g., JavaScript) commands on the client side.	INP

Category	Definition for the area of IT security	Related to
Directory Browsing	<i>see Path Traversal</i>	
Directory Traversal	<i>see Path Traversal</i>	
Flow Injection (FLO)	This category is a creation of our own. It is a special case of Missing Input Validation and is usually not detectable by security scanners. This vulnerability is based on setting application states which depend on untrustworthy user data. Thus, the control flow of an application's code could be influenced by an attacker.	MIV
<i>Information Disclosure (INF)</i>	An information disclosure security flaw can be defined as the emission of data or information which is not intended to become available to the public. This can be internal or private data. There are several issues in this category which are not programming errors, like the wrong or public storage of sensitive data.	
Input Validation (INP)	Usually any input/external data – not only from users – of an application has to be checked to see whether it conforms to intended formats or properties. Such procedures usually also involve data filtering (sanitization) and adequate <i>output encoding</i> . If input validation, filtering, and output encoding are missing or incomplete, this can enable a variety of attacks.	COD COO RES SQL XSS
<i>Missing Input Validation (MIV)</i>	All programming errors, but also errors in system design or specification, which cannot be classified in another security category are called Missing Input Validation. Thus, these errors are not typical programming errors. Moreover, it is usually not possible to test for resulting security flaws.	FLO
Path Browsing	<i>see Path Traversal</i>	
Path Traversal (PAT)	Can be generally defined as unintended access to application files or directories by injecting (sub) paths and filenames. The injection, for instance, can take place into application URLs. This topic will be discussed in detail in section 4	COD INP RES
<i>Resource Tampering (RES)</i>	We define Resource Tampering flaws as a category of security vulnerability related to unintentional access to system resources via the application layer, like in the case of path traversal.	COD INP PAT
<i>SQL Code Injection (SQL)</i>	Results of successful attacks of this category are the execution of arbitrary SQL statements and commands on the application's database backend(s).	INP
<i>Unreleased Resources (UNR)</i>	Some program resources, which are, e.g., variables and class instances (objects), have to be explicitly unloaded for freeing application memory. If they are not released properly and not caught by the Java garbage collector, they might lead to increased memory consumption. Thus, in a broader sense, unreleased resources can enable "Denial of Service" attacks and are a concern for an application's security.	

**Legend**

- *Expression* (blue italics): Chosen sections of document Version 1.0
- *Expression* (red italics): New sections of document, Version 2.0
- Please note: Path Browsing, Path Traversal, Directory Browsing, and Directory Traversal are used synonymously throughout the document

# 1 SQL Code Injection

Brief description of related security attacks	<ul style="list-style-type: none"> <li>• Injection of arbitrary SQL commands</li> <li>• Manipulation of existing SQL queries</li> </ul>
This topic is related to	<ul style="list-style-type: none"> <li>• Dealing with HTTP request parameters</li> <li>• Input filtering, validation, and encoding</li> <li>• Secure access to databases via JDBC</li> </ul>
Degree of severity	<b>HIGH</b>
Consequential and potential damage	<ul style="list-style-type: none"> <li>• Complete and possibly unperceived access to the whole application database</li> <li>• Manipulation of data, including deletion of tables and databases</li> <li>• Access to the database system's host / server system (depending on the functionality provided by the database system's SQL)</li> </ul>
Affected packages (among others)	<code>java.sql.*</code>
WHEN NOT TO READ	<ul style="list-style-type: none"> <li>• If the application does not deal with database access</li> <li>• If the application does not deal with user data</li> </ul>

## 1.1 Introduction

Today, most web applications have to deal with dynamic content. Usually, three-tier architectures are used for realizing these web applications. A web browser (client), the application code base (server), and a database system (same or another server) are the architecture's components. Applications mainly use SQL for read and write access to the database. The web application client has no direct access to the database system. However, as the client is communicating with the application, it triggers access and operations of the database executed by the application. Although this is an intended functionality, it gives the client's user the opportunity to manipulate data, which has an impact on the database behavior. Input data, like user inputs within web forms, can contain SQL code or commands. Thus, the application has to ensure that no processed input data can manipulate the SQL queries it is using. In the following, we describe SQL injection vulnerabilities from a technical point of view with regard to Java. SQL queries within Java code are simple character strings. They are transmitted to the application's database backend via the JDBC API. SQL injection vulnerabilities arise because different parts of a query (substrings) are merged to a final query string within the application code. The set of used substrings includes unfiltered user data which comes, e.g., from HTTP requests. This can result in injections of arbitrary SQL commands from outside the application. Thus, the security flaws in this category can be primarily traced back to insecure or nonexistent input validation.

### 1.1.1 Variants

In the following, we will discuss attack examples in more detail, based on two concrete programming variants.

#### 1.1.1.1 SQL queries via Statement

In general, the JDBC `Statement` interface can be used in a secure way. However, there are only two scenarios for a secure implementation:

- Only constant SQL queries are executed
- All input data which becomes part of the SQL query is filtered by a rigorous external input validation filter

In the context of Java source code, constant SQL queries are constant character strings. Thus, once initialized, they are never changed and, especially, they cannot be changed depending on user data.

In the following we show a typical SQL injection example. Strings with user input coming from HTTP request parameters are appended to an UPDATE query.

#### Example of bad code

We assume a web application with a simple input form for changing a user's password. The form takes the user's name, its old password, and a chosen new password. Submitting the form transfers the filled-in user data via HTTP to an application Servlet. This Servlet executes a password change for the corresponding user by updating the application's user table `T_Customer`. The update of the affected data record is performed by an insecure SQL query.

```
String strInpName = request.getParameter("username");
String strInpPWNew = request.getParameter("newpass");
String strInpPWOld = request.getParameter("oldpass");

// Preparing database connection "con"

Statement stm = con.createStatement();
stm.executeUpdate("UPDATE T_Customer SET password=' "
    + strInpPWNew
    + " ' WHERE (name=' "
    + strInpName
    + " ') AND (password=' "
    + strInpPWOld
    + " ');");
```

Please note that this source code and the above mentioned application context are only a simple example. Furthermore, please note that it is irrelevant at which part of the query the injection is performed. This implies injections are possible at all components of an SQL query, e.g., the column choice (SELECT-clause), the table choice (FROM-clause), or the constraint (WHERE-clause). Moreover, injections are not limited to SELECT-statements. They are possible for all statement types, like INSERT, UPDATE, DELETE, CREATE TABLE etc. Other popular injection variants are the appending of an additional SQL statement to an existing one, which implies

both are executed, or to amputate an existing statement by using SQL comment characters (--).

### Sample attack

In the following, we want to briefly show how a concrete attack on the example shown above could be performed. We assume the Servlet of the example can be accessed by HTTP GET requests. A URL call which is actually intended to be the result of the input form could look like the following. Please note that, for the sake of readability, we do not use escaped HEX encoding for the URL in this example, which would be necessary in a real-world scenario:

```
http://www.myserver.com/changePassword?username=pumpkin05
&oldpass=notgood&newpass=b3TT3r
```

The Servlet of the code example would create the following SQL query:

```
UPDATE T_Customer
SET password='b3TT3r'
WHERE (name='pumpkin05') AND (password='notgood');
```

An attacker with information about the query's original structure (e.g., gained by trial-and-error or provided by an insider) could create, e.g., this request:

```
http://www.myserver.com/changePassword?username=doesntmatter
&oldpass=bla&newpass=myGoodOne';--
```

Now, the resulting SQL query at the Servlet would be:

```
UPDATE T_Customer
SET password='myGoodOne' ;--'
WHERE (name='doesntmatter') AND (password='bla');
```

As -- is the SQL special character for comments, the Servlet would execute this query:

```
UPDATE T_Customer
SET password='myGoodOne' ;--
```

This would obviously imply that the attacker successfully overwrites all user passwords with his/her own password. Furthermore, this could imply, for instance, that the attacker is able to access all application user accounts, including administration accounts.

For the practical execution of this example, an attacker would only need to edit the web site, including the input form, offline in most cases. Otherwise, an attacker could also use an HTTP request manipulation tool. Please note that these kinds of attacks can be executed not only for HTTP GET but for all HTTP and HTTPS request types. In the case of, e.g., HTTP POST, an attacker would need a request manipulation tool, but the attack still works in a similar way.

### 1.1.1.2 SQL queries via PreparedStatement

In this section we refer to the example of section 1.1.1.1. We assume that a developer has realized that s/he had used the Statement interface in an insecure way, as shown above. The PreparedStatement interface could be indeed an adequate solution. However, PreparedStatement can also be used wrongly.

#### Example for bad code

An easy but wrong way to fix the usage of Statement in the example of section 1.1.1.1 would be to use PreparedStatement:

```
String strInpName = request.getParameter("username");
String strInpPWNew = request.getParameter("newpass");
String strInpPWOld = request.getParameter("oldpass");

// Preparing database connection "con"

PreparedStatement ps = con.prepareStatement("UPDATE
    T_Customer SET password=' "
    + strInpPWNew
    + " ' WHERE (name=' "
    + strInpName
    + " ') AND (password=' "
    + strInpPWOld
    + " ');");
ps.executeUpdate();
```

The code is now freed of the Statement interface. However, this code is still insecure and contains the same vulnerabilities as the example in section 1.1.1.1. This shows that using PreparedStatement can be delusive. The correct usage of PreparedStatement is described in section 1.2.

#### Sample attack

The same as in the example in section 1.1.1.1

### 1.1.2 Damage in the case of Non-Compliance

The general result of existing SQL injection vulnerabilities is the opportunity to inject arbitrary SQL commands or manipulate existing SQL commands processed by the application's database back end. Possible consequences can be summarized as follows:

- Unperceived and unauthorized access to information and data theft
- Data and database structure manipulation
- Data loss
- Access to the database system's host system. This depends on the functionality provided by the database system's SQL. This can again result in new threats, like the installation of Trojan horses or access to other sensitive and internal hosts or workstations reachable via network.

## 1.2 Prevention, Countermeasures, Solutions

Besides the dangerous implications of SQL injection vulnerabilities described in section 1.1.2, one can find much detailed information about other potential threats. The following material should be useful for application developers with security concerns when dealing with SQL injection faults. We recommend material provided by the “Open Web Application Security Project” (OWASP [1]) and the “Web Application Security Consortium” (WASC [2]). Especially the OWASP “Top Ten” [3] and the “Web Security Threat Classification” [4] provide a minimum standard for web application security. [4] includes a documentation of most of the security threats and also gives typical code examples.

### 1.2.1 Description

The general guideline for protecting application code from SQL injection vulnerabilities is: “Do not trust any data coming from outside your application code.” Thus, the general countermeasure is adequate input validation, filtering, and encoding. Of course, this is also true for several other security topics [3] [4].

Input validation has to be performed in particular on all HTTP request parameters. In general, it has to be performed on all data which could be potentially tainted with user data, e.g., database records.

Moreover, we want to highlight that input validation and filtering has to be performed on the server-side. This is necessary because filtering mechanisms on the client-side can easily be disabled by attackers.

### 1.2.2 Platform Limitations or Extensions

All solutions described here are built-in platform solutions. Thus, no extensions are needed. No limitations are known.

### 1.2.3 Solution Variants

Overview of “standard” solutions:

Solution	Platform	Libraries
<i>Prepared Statements</i>	Java in general (all versions)	java.sql
<i>Stored Procedures</i>	Database systems with corresponding functionality (e.g., MS SQL Server)	java.sql

#### 1.2.3.1 PreparedStatement interface

Encoding user data according to the database properties with consideration of all database special characters is the most important issue when creating a secure SQL query. JDBC provides the `PreparedStatement` interface, which enables the developer to do this without further knowledge about database properties and features. In a manner of speaking, the `PreparedStatement` interface provides automatic and secure encoding of data, when it is used correctly.

It enables the application developer to distinguish between the SQL command structure and the user input data. This distinction can be achieved by using corresponding class methods and does not require extra knowledge about the database back end. Similar solutions (always called prepared SQL statements) can also be found in other development platforms, like Perl, PHP, and also Microsoft ADO. Please note that the concrete implementation of the interface depends on which JDBC database driver is used.

#### 1.2.3.1.1 Requisites

Technical Requisites	Platform Release	Features/Interfaces to be used
Database connections via JDBC	Java (all)	JDBC's PreparedStatement interface

#### 1.2.3.1.2 Procedure

For using prepared SQL queries, the following JDBC interfaces have to be used:

- java.sql.Connection
- java.sql.PreparedStatement
- java.sql.ResultSet

The "standard" way to start is by creating a Connection object out of a valid JDBC driver provided for the corresponding database system. There are several ways of doing this. As this is not important for the case of SQL injection, we only provide one example:

```
Class.forName("org.postgresql.Driver");
           // Or, e.g., "org.gjt.mm.mysql.Driver"
Connection con;
con = DriverManager.getConnection(url, username, password);
```

The database connection is used for creating a prepared SQL query. This is done by creating an instance of a PreparedStatement out of the previously created Connection object. Furthermore, to create the PreparedStatement object, the content of the SQL query can be committed as string parameter. The following procedure is valid for all kinds of SQL queries, e.g., UPDATE, DELETE, and SELECT, as well as DROP TABLE, or ALTER TABLE, etc.:

```
PreparedStatement ps = con.prepareStatement("UPDATE T_Customer
SET password=? WHERE (name=?) AND (password=?);");
```

or

```
PreparedStatement ps = con.prepareStatement("SELECT T_Customer
WHERE password=? AND name=?;");
```

Please note that all parameters of the query which have to be set on a value coming from the outside are combined with a ?. Such a question mark stands for an arbitrary value of an arbitrary data type (possible quotation marks must not be set).

The quotation marks can be set in order of appearance by using the corresponding set methods of the `PreparedStatement` interface, like the following – taken from the JDBC API documentation:

```
void setInt(int parameterIndex, int x)
    Sets the designated parameter to the given Java int value.
void setLong(int parameterIndex, long x)
    Sets the designated parameter to the given Java long value.
void setString(int parameterIndex, String x)
    Sets the designated parameter to the given Java String value.
```

In the first query example, the following instructions could follow the creation of the `PreparedStatement` object. The method `setString` is used for setting the SQL statement's respective parameters:

```
ps.setString(1, "NewPass");
ps.setString(2, "Username");
ps.setString(3, "OldPass");
```

By choosing a specific set method, one is also setting the corresponding data type of the respective parameters. For instance, the value for password in the SQL query example above could be a character string like `NewPass`. In this case, the internal representation of `password=?` would be the substring `password='mypass'`.

Please note: prevention of SQL code injection takes place at this point. The set methods encode every input value, e.g., a character string, according to the underlying database system. For instance, `setString` would transform the potentially dangerous input string `'--` (compare with section 1.1.1.1) which has been set as the value for the parameter `password=?` into the internal representation `password='\'\-\-`. As the set methods are implemented in the database driver, they are responsible for the secure encoding of the parameter values, which could be user input data.

If there is nothing to set because the SQL query is a constant string (e.g., `DELETE * FROM T_TMP`), the set methods are not used. In this case, there will be no quotation mark in the query string and thus no parameter to be set.

To execute the SQL query, the corresponding execution method of the `PreparedStatement` has to be used. It could be, e.g., one of the following – taken from the JDBC API documentation:

```
ResultSet executeQuery()
    Executes the SQL query in this PreparedStatement object and returns the
    ResultSet object generated by the query.
int executeUpdate()
    Executes the SQL statement in this PreparedStatement object, which must be
    an SQL INSERT, UPDATE or DELETE statement; or an SQL statement that
    returns nothing, such as a DDL statement.
```

If a `SELECT` statement is executed, the execution method will return the corresponding JDBC `ResultSet` for browsing the respective data records. Executing database changes, e.g., committing new data, will not return a `ResultSet` but rather a status value stating whether the execution was successful or not. This behavior is equal to the behavior of the corresponding methods of the insecure `Statement` interface.

#### 1.2.3.1.3 To be avoided

In section 1.1.1.2 we showed one of the major faults which could appear when using `PreparedStatement`, and which results in the complete loss of the security actually provided by this interface. The fault is to nevertheless create an SQL query string by appending substrings, like in the case of using JDBC's `Statement` interface. Thus, all (external) query parameters have to be set by corresponding methods, e.g. `setString`, provided by `PreparedStatement`.

#### 1.2.3.1.4 More information

Please see the Java API documentation for more details about the `PreparedStatement` interface.

#### 1.2.3.1.5 Example of good code

Referring to the code example in section 1.1.1.1, we now show how to make the same code secure without changing the semantics. This example shows the correct usage of the `PreparedStatement` interface:

```
// ...
// Preparing database connection "con"

PreparedStatement ps = con.prepareStatement("UPDATE T_Customer
      SET password=? WHERE (name=?) AND (password=?);");
ps.setString(1, strInpPWNew);
ps.setString(2, strInpName);
ps.setString(3, strInpPWOld);
ps.executeUpdate();
```

As described above, using this kind of notation for the SQL query guarantees the separation of commands and input data. Inputs to the query are placed at the location of the `?`s and only there. The `?`s are clearly addressed by sequential numeration inside of the `setString` method calls. As the `PreparedStatement` object in the example correctly encodes all data set via the `setString` method, no manipulation of the existing SQL query is possible.

Another practical advantage of this solution is that developers do not have to care about correct SQL encoding inside of query strings. They only have to use the appropriate set method for the corresponding Java data type, e.g., `setString`, `setInt`, or `setTimestamp`. The actual SQL query is one simple string and, moreover, no appending of substrings has to be performed.

### 1.2.3.2 Stored database procedures

Using stored SQL procedures on the database server system usually avoids SQL injection vulnerabilities. This is because the database knows the stored procedure's structure and "automatically" encodes input data correctly. Thus, the command structure of an SQL query cannot be manipulated from the outside.

This variant has another practical impact. The management of SQL queries is swapped out of the Java source code. This implies that SQL queries are not created using Java character strings. Often, the latter leads to a development overhead, because SQL statements have to be encoded according to the Java string format.

A possible disadvantage of this variant could be the compatibility of stored procedures which have been developed. Depending on the database system used, it is possible that stored procedures cannot easily be adapted when the application's database system is exchanged.

#### 1.2.3.2.1 Requisites

The general requisite for using stored database procedures is that the database system provides such functionality. Moreover, in the case of Java applications, there has to be a corresponding JDBC driver for the database which allows access to stored procedures via SQL or Java.

## 1.3 Test Procedure

Please note that the section Test Procedures in this document gives an introduction to the usage of the test procedures described.

### 1.3.1 Outline

Usually, the creation of SQL queries in Java is realized via JDBC. JDBC provides two kinds of SQL statements:

- `java.sql.PreparedStatement`
- `java.sql.Statement`

If SQL queries are created by simply assembling character strings which contain external/user inputs (compare 1.1.1), SQL code injection vulnerabilities might be created as well. This test procedure will ensure that SQL injection vulnerabilities are prevented in source code.

Test Attributes	
Platform	J2EE/Java
Operating Systems	All, no restrictions
Type of test	Code Review
When should be tested	From initial to final implementation phase

Test Attributes	
Tool support	Assisting usage of source code scanners and static analysis tools is possible for more sophisticated courses of action. Additionally, it is possible to use the Open Source tool FindBugs to perform this test procedure automatically (see section 1.3.4 for more details).

### 1.3.2 Test Execution

Basically, there are two methods which allow the preparation of the execution of an SQL query:

- `PreparedStatement` `prepareStatement(String sql)` of `java.sql.Connection`
- `ResultSet` `executeQuery(String sql)` of `java.sql.Statement`

Please note that the method `prepareStatement` also has more signatures [5] and that `java.sql.Statement` also provides other execute methods for SQL queries [6].

Any occurrences of these methods have to be found. If the parameter which contains the SQL string is not hard-coded or read from a constant, a vulnerable piece of code has been located.

#### 1.3.2.1 Special Cases in Practice

n/a

### 1.3.3 Interpretation of Test Results

Whenever a code line which contains one of the methods listed above calls this method with a parameter (the SQL string) which is not hard-coded or read from a constant, the piece of code has to be reengineered.

Please note that a secure Java built-in procedure exists to create SQL queries (compare to 1.2.3.1). We want to highlight that the use of the set methods, which are provided by `java.sql.PreparedStatement`, for adding external data to an SQL query, such as user inputs, is more secure than the use of proprietary filter methods. Moreover, `java.sql.PreparedStatement` fully replaces `java.sql.Statement` for most applications.

### 1.3.4 Special Tool Support

The Open Source tool FindBugs provides security bug detectors which cover the test procedure given above. Thus, this test procedure can be executed automatically by applying FindBugs to the corresponding application. In the following, we list the necessary steps:

- The current archive which contains the FindBugs Java program can be downloaded from <http://findbugs.sourceforge.net/downloads.html>.
- The archive has to be unpacked.

- It has to be ensured that an up-to-date Java Runtime Environment (JRE) is installed on the system which is to run FindBugs.
- FindBugs can be started via the batch script `bin\findbugs.bat` (Windows) or `bin/findbugs` (UNIX).
- When FindBugs is running, a new project has to be opened by choosing the menu “File” – “New Project”. A new project window will open. Here, FindBugs can be applied to the corresponding Java application by providing the path to the compiled application, the path to the application’s source code (not required), and the application’s classpath entries. Then, the FindBugs analysis can be started.
- The result screen which follows might also show other bugs, since FindBugs includes many bug detectors for Java. SQL injection vulnerabilities are marked by the term “SQL” at the beginning of a line which contains a bug report. Additionally, the corresponding line will contain a message like this: “Non-constant string passed to execute method on an SQL statement”.

#### 1.4 References

- [1] The Open Web Application Security Project (OWASP), <http://www.owasp.org>
- [2] The Web Application Security Consortium (WASC), <http://www.webappsec.org/>
- [3] The OWASP Top Ten, <http://www.owasp.org/documentation/topten.html>
- [4] The Web Security Threat Classification, <http://www.webappsec.org/projects/threat/>
- [5] <http://java.sun.com/j2se/1.4.2/docs/api/java/sql/Connection.html>
- [6] <http://java.sun.com/j2se/1.4.2/docs/api/java/sql/Statement.html>

## 2 Cross Site Scripting (XSS)

Brief description of related security attacks	<ul style="list-style-type: none"> <li>• Injection and execution of arbitrary script commands on web application clients (e.g., JavaScript)</li> <li>• Redirecting web application clients to arbitrary locations</li> </ul>
This topic is related to	<ul style="list-style-type: none"> <li>• Dealing with HTTP (GET/POST) request parameters</li> <li>• Output encoding and input validation/filtering</li> <li>• Storing request data in the application's database</li> </ul>
Degree of severity	<b>HIGH</b>
Consequential and potential damage	<ul style="list-style-type: none"> <li>• Redirecting of clients for further attacks (e.g., exploiting web browser vulnerabilities)</li> <li>• Identity theft and impersonation</li> <li>• Session hijacking</li> </ul>
Affected packages (among others)	<code>javax.servlet.* , javax.servlet.http.*</code>
WHEN NOT TO READ	<ul style="list-style-type: none"> <li>• If the application does not deal with HTTP requests</li> </ul>

### 2.1 Introduction

Cross-Site Scripting (XSS) attacks manipulate HTML pages by injection of malicious script code or by other indirect techniques, such as redirection to another server, or logical attacks, e.g., replacing images, or changing style sheets. Attackers look for HTML pages where user input is written back to the HTML page, e.g., during a logon failure, the logon screen is displayed a second time. These examples demonstrate the potential security vulnerabilities for XSS attacks, if user's input is written back to the HTML page.

Since HTML is based on tags, the browser may even interpret and execute JavaScript or ActiveX controls, which might contain malicious script commands. These commands are executed by a user's browser when the user opens a manipulated HTML page.

#### 2.1.1 Variants

In the following, we will discuss attack examples in more detail, based on two common variants: Client-side and server-side XSS. Please note that XSS attacks can occur as output

- between tags,
- inside of tags,
- inside a script context.

##### 2.1.1.1 Client-side XSS

The definition of client-side XSS attacks is as follows. The attacker injects script code into the vulnerable web application. The malicious code becomes part of the application's original HTML page(s). If the corresponding manipulated web site,

which includes the malicious code, is opened by an application client, the attacker's code is executed by the client's browser.

### Example of bad code

In the following we give a simple example for a client-side XSS vulnerability of a Java Servlet or JSP. We neglect the actual functionality or context for a corresponding web application.

```
// Servlet or JSP
// "request" is the HttpServletRequest object
// "response" is the HttpServletResponse object

String param1 = request.getParameter("formparam1");
String param2 = request.getParameter("formparam2");
// ...

// Computing, generating HTML output, appending strings, ...
// paramX still contains the original content

response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<html><body>MyOutput");
// ...
out.println(param1);
// ...
out.println("<br><br><br>Good bye!");
out.println("</body></html>");
```

Please note that the execution of this application code is triggered by sending an HTTP request to a certain URL of the application. In this example the content of the request's parameters, which are character strings, is loaded into local `String` variables. After the operations of the application's Servlet/JSP, these string values are written into the application's HTML response. We assume this happens without any filtering or output encoding of the input strings. This allows several attacks, because every input of the parameters is written into the HTML page generated by the web application without adequate (HTML) output encoding. This enables attackers to directly inject script code into the application's HTML page which is interpreted by the client's browser. As the malicious script code is not encoded when written into the HTML page (output encoding), it becomes part of the page's original HTML.

### Sample attack

We give only one example for a client-side XSS attack. We assume a vulnerable web application (similar to the example above) has a search function which can be used via the following URL:

```
http://www.mytrustworthywebapp.com/pages/search
```

Furthermore, we assume the content of the page behind this URL depends on one parameter which includes the keyword to search for. This parameter is transferred via HTTP GET and thus within the URL. A regular URL call could be:

```
http://www.mytrustworthywebapp.com/  
search?keyword=telephone%20ceo
```

The corresponding output could be:

```
MyOutput  
  
You searched for: telephone ceo  
  
Results: ...
```

As we assumed the web application is vulnerable – because every input of the parameter is written into the HTML response page – an attacker could create a URL call including JavaScript. To improve readability we neglect the usage of escaped HEX encoding for the URL. Normally, all special characters have to be HEX encoded:

```
http://www.mytrustworthywebapp.com/  
search?keyword=<script>alert(42)</script>
```

If the attacker is able to entrap an application user to, e.g., click on this URL, the JavaScript code would be executed on the user's web browser. This is a result of the Servlet/JSP's property of writing every input data of its parameter into its HTML page response. Of course, a JavaScript alert box is a minor attack, but the example shows that every conceivable combination of JavaScript commands is possible. This includes the creation and forwarding of hidden forms or access to the client's cookie for this URL/web application or the redirection of the client to an attacker's web sites, etc.

Please note that, depending on the amount of HTML tags and, e.g., JavaScript commands, the web application allows a large set of possible attack variants. For instance, there are several HTML tags where injections are possible. Moreover, HTML includes so-called event handlers, normally used for including dynamic content on web pages. Examples for possible valid statements, again for alert(), are:

- `<a href="javascript:alert();">Click me!</a>`
- ``
- `<textarea onchange="alert();">`
- ...

### 2.1.1.2 Server-side XSS

The definition of server-side XSS attacks is as follows: The attacker successfully injects commands into the vulnerable web application and the application executes those commands. Again, this only occurs when the corresponding manipulated web site, which includes the malicious code, is opened by the client. The difference compared to client-side XSS is that the client does not execute the commands

injected by the attacker. The client receives a direct manipulated response from the web application, like a redirect.

#### Example of bad code

We now give one classic example of server-side XSS: HTTP redirects are manipulated by the attacker. We assume the application performs redirects by this URL:

```
http://www.mytrustworthywebapp.com/goto?url=/pages/mypage.html
```

The corresponding Servlet/JSP could look like this:

```
// Servlet or JSP
// "request" is the HttpServletRequest object
// "response" is the HttpServletResponse object

String param1 = request.getParameter("url");
// ...

// Other operations

response.sendRedirect(param1);
```

#### Sample attack

In this case, the attacker is able to create URLs like the following. We again neglect HEX encoding, which would be necessary for the parameter data:

```
http://www.mytrustworthywebapp.com/goto?url=
http://attackersitelookstrustworthy.org
```

A click on this URL which looks like the URL to the user's web application would redirect the client to the attacker's web site. Via *its* site, the attacker could try further attacks, such as exploiting web browser vulnerabilities, etc.

### **2.1.2 Damage in the case of Non-Compliance**

In the following we summarize some specific aspects related to the potential damage caused by XSS. In the case of JavaScript or VBScript, potential attacks could:

- Redirect the browser to a different page by overwriting the current document
- Access all user's inputs and send them to a rogue server
- Access the user's cookies (e.g., session hijacking, cookie manipulation)
- Insert new script tags as output between tags, which, for example, create new event handlers that are executed when certain events occur

The consequences of ActiveX attacks are:

- The hacker might access and modify files on any accessible drives and memory
- Application actions might be executed under other user's privileges
- The hacker might install other applications, like Trojan horses

## 2.2 Prevention, Countermeasures, Solutions

Besides the dangerous implications of XSS vulnerabilities described in section 2.1.2 and above, much detailed material about their potential threat is available. The following material should help application developers with security concerns when dealing with XSS faults – similar to the case of SQL injection. We recommend material provided by the “Open Web Application Security Project” (OWASP [1]) and the “Web Application Security Consortium” (WASC [2]). Especially the OWASP “Top Ten” [3] and the “Web Security Threat Classification” [4] provide a minimum standard for web application security. [4] includes a documentation of most of the security threats, and also gives typical code examples.

### 2.2.1 Description

The general guideline for protecting applications from XSS vulnerabilities is: “Do not trust any data coming from outside your application code.” Thus, the general countermeasure is adequate input validation, input filtering, and – especially – output encoding (HTML, JavaScript) of input values. Of course, this is also true for several other security-related topics [3] [4].

Input validation has to be performed especially on all HTTP request parameters. In general, it has to be performed on all data which could be potentially tainted with user data, e.g., database records, and which is used inside of generated HTML output. It must be absolutely clear which data is filtered and must be encoded for the output.

Moreover, we want to highlight that input validation and filtering has to be performed on the server-side. This is necessary because filtering mechanisms on the client-side (e.g., JavaScript code validating form inputs) can easily be disabled or bypassed by attackers.

If developers have to allow HTML and JavaScript code as user inputs for web applications, we recommend using a whitelist procedure for validating and filtering inputs instead of a pure blacklist procedure. This provides the advantage of allowing an amount of secure code commands (whitelist) instead of only determining which command of the whole command set could lead to attacks or, more generally, to security faults (blacklist). Moreover, a combination of both is possible: starting with a whitelist procedure which then is followed by a blacklist procedure applied to the user inputs. In practice, the performance loss of the input filtering procedure (sanitization) always has to be considered as well as the trade-off against output encoding.<sup>2</sup>

It should be noted that the countermeasures for XSS apply also to applications based on a Three Tier Architecture. This implies that not every XSS attack works via a URL created by an attacker using, e.g., JavaScript injection into the URL’s request parameters. This is the case in the example in section 2.1.1.1. There, the attacker’s JavaScript code is written directly to the HTML page which is returned when calling the URL. Hence, the attacker’s code is interpreted immediately by the client when opening the corresponding page of the vulnerable web application. It is also possible that an attacker is able to place and *store* its malicious code on a certain page of the web application. This is true when the application uses a database for storing user

---

<sup>2</sup> We will publish a best practice document for input validation within the “Secologic” project.

inputs and generates its output pages out of this database. In this case, the attacker would only need a standard URL without malicious code to the web application's page where its malicious code is opened. It is not necessary to inject the code inside of the URL.

There are two possibilities for preventing XSS attacks in this case. Firstly, filtering and (output) encoding of corresponding user inputs can be performed before database updates or inserts, which implies before writing data to the database. Secondly, output encoding of every data of the database can be performed when it is used to generate the application's output pages. The first possibility offers the advantage that the application database is kept "clean" and thus kept free of injection fragments of XSS attempts. However, in practice the second case (output encoding) always has to be performed because applications do not only consist of user data. Performing both procedures can lead to a performance loss, thus an adequate trade-off for the respective use case has to be found for each individual case.

### 2.2.2 Platform Limitations or Extensions

All solutions described here are based on encoding user inputs. The J2EE platform has no built-in solution for this. Thus, there are two possibilities: Developing encoding procedures or using external Java libraries/packages.

### 2.2.3 Solution Variants

Overview of "exemplary" solutions:

Solution	Platform	Libraries
<i>Encoding</i>	Java in general (all versions)	n/a; individual procedures are necessary
<i>Encoding</i>	Java extension (for all versions)	Apache Jakarta Commons Lang [5]

#### 2.2.3.1 Basic rules

In this section we present the basic rules for preventing XSS vulnerabilities in web applications. The main task is to implement a complete input validation. The following aspects have to be considered when implementing input validation and the corresponding filtering procedures:

- Constrain input and input fields (e.g., database fields, String variables etc.)
- All input has to be validated
  - Define a codepage (e.g., character set ISO-8859-1) to clearly decide which character encoding is used<sup>3</sup>
  - Filter special characters and meta-characters (e.g., tag commands in HTML, like <)
  - Restrict variables to those characters that are explicitly allowed (whitelist)

<sup>3</sup> In practice, it has to be remembered that web browsers might be able to redefine the character set of HTTP requests.

- If users are allowed to enter a URL or external links within input fields, restrict the domain of the URL and permit only the selection of approved URLs. Compare with server-side XSS in section 2.1.1.2.
- Validation in general involves the following aspects:
  - Field length (e.g., character strings)
  - Data types
  - Range (e.g., numbers in general, dates, postcode, etc.)

Another important aspect in the context of generated HTML code is to always enclose input values in quotation marks. In this case, attacks are only possible if the context of the input value – which is then embedded in quotation marks – is left by using (additional) quotation marks. Thus, malicious input can simply be detected by filtering quotation marks (") within input values.

Omitting the quotation marks will make an XSS attack easier, because attackers do not have to leave the context by setting any ". Therefore, it is much more difficult to filter malicious code out of HTML pages like the following:

```
<form name=HUGO>
  <input type=text name=user value=hello>
</form>
```

A better version of this HTML code fragment would be:

```
<form name="HUGO">
  <input type="text" name="user" value="hello">
</form>
```

### 2.2.3.2 Encoding procedures

As most issues related to the basic rules for preventing XSS vulnerabilities (see section 2.2.3.1) depend on the application context (e.g., input field ranges), the filtering of special characters is application-independent. In the case of XSS via HTML code, the most important characters are:

Character	Name	HTML entity	HTML char code
"	Quotation mark	&quot;	&#34;
'	Apostrophe		&#39;
&	Ampersand	&amp;	&#38;
<	Less-than	&lt;	&#60;
>	Greater-than	&gt;	&#62;

An overview of all special characters in HTML and their HTML char codes can be found in [6]. Although it is always necessary to replace “natural” input values, in order to display them in HTML again, it is not always performed. For instance, a “natural” input value in German for a street within an address field would be “Lindenstraße”. As the “ß” is not visible in all web browsers all over the world, it has to be converted

into HTML encoding. Thus, displaying this input value on an HTML page, requires its conversion into “Lindenstra&szlig;e”.

Moreover, some of the special characters are meta-characters. Such characters have specific semantics and are part of the HTML syntax, like < and > in the case of HTML tags (e.g., <a href="...>). These characters are used for the XSS attack discussed in section 2.1.1.1. As in the case of the German “ß”, the meta-characters can also be an optional part of visible HTML output. Therefore, they have to be used in the form of HTML char codes and thus no longer have an effect on the HTML syntax. This also implies that meta-characters within user inputs cannot be used for XSS attacks when they are converted into their HTML char codes.

It has to be pointed out that no “automatic” HTML output encoding procedures are part of the J2EE. Thus, in this section (especially in subsection 2.2.3.2.2), we show an exemplary “manual” replacement strategy for input validation in the context of XSS prevention.

### 2.2.3.2.1 Requisites

Technical Requisites	Platform Release	Features/Interfaces to be used
n/a	Java (all)	String, Character for manual encoding

### 2.2.3.2.2 Procedure

The following example replaces the HTML meta-characters <, >, and " inside of an input string into their HTML char codes (entities) &lt;, &gt;, &quot;. This is a simple example of how to prevent the XSS attack shown in section 2.1.1.1. The prevention is implemented using output encoding in the context of HTML. We first show the implementation of the replacement method:

```
public String HTMLencode(String x){
    String ret="";
    for(int i=0; i<x.length(); i++){
        char c = x.charAt(i);
        switch(c){
            case '"':
                ret=ret + "&quot;";
                break;
            case '<':
                ret=ret + "&lt;";
                break;
            case '>':
                ret=ret + "&gt;";
                break;
            default: ret=ret + Character.toString(c);
        }
    }
    return ret;
}
```

```
}
```

To make the vulnerable Servlet of section 2.1.1.1 secure against the sample attack, we have to use the method inside the Servlet. Please note that the method is not a solution for XSS attacks in general. It is only provided for our sample scenario.

```
// ...

PrintWriter out = response.getWriter();

out.println("<html><body>MyOutput");
out.println(HTMLencode.HTMLencode(param1));
out.println("</html></body> ");
```

This will prevent the XSS attack described above. For instance, an input like "><SCRIPT>" would be converted into "&gt;&lt;SCRIPT&gt;", which is again displayed as "><SCRIPT>" on an HTML page.

Another and more general approach for meta-character replacement uses HTML char codes for HTML encoding. This is more practical, because it is possible to give a simple list (in this example an array) of meta-characters to be filtered. Please note that the HTML char codes are supported at least by the web browsers Internet Explorer and Mozilla.

```
public String HTMLencode2(String x){

    // BEGIN List of characters to filter
    char[] myChars={'"', '<', '>'};
    // END List of characters to filter

    String ret="";
    boolean set=false;
    for(int i=0; i<x.length(); i++){
        char c = x.charAt(i);
        for(int ii=0; ii<myChars.length; ii++){
            if(c==myChars[ii]){
                ret=ret + "&#" + (int)c + ";";
                set=true;
            }
        }
        if(!set)ret=ret + Character.toString(c);
        else set=false;
    }
    return ret;
}
```

This will also prevent the XSS attack described above. The input "><SCRIPT>" would be converted to "&#34;&#62;&#60;SCRIPT&#62;", which is also displayed as "><SCRIPT>" on an HTML page.

### 2.2.3.2.3 To be avoided

Derived from the basic rules for XSS vulnerability prevention (see section 2.2.3.1), the following issues have to be avoided during the development of web applications. These can be summarized into the topic “incomplete input validation”:

- Ignoring certain inputs (input fields like form fields or hidden fields and request parameters in general)
- Ignoring certain meta-characters of the HTML or JavaScript syntax

### 2.2.3.2.4 More information

Please see 2.4 References for more information.

### 2.2.3.3 External encoding libraries

We will now briefly refer to implementations of encoding procedures which can be easily used within Java. Such implementations are well-discussed and thus usually offer automatic and complete encoding. These Java packages or libraries can be applied to the user’s input values similarly to the method `HTMLEncode` of the short example in section 2.2.3.2.2.

First, we want to refer to the Apache Jakarta Commons Lang package [5]. It includes the `org.apache.commons.lang.StringEscapeUtils` which provides the following encoding methods – taken from the API documentation:

```
static String escapeHtml(String str)
    Escapes the characters in a String using HTML entities.
static String escapeJava(String str)
    Escapes the characters in a String using Java String rules.
static String escapeJavaScript(String str)
    Escapes the characters in a String using JavaScript String rules.
static String escapeXml(String str)
    Escapes the characters in a String using XML entities.
```

The “unescape” methods are omitted here. Further information about the Lang package and the `StringEscapeUtils` can be found in [5], including the complete API documentation. Please note that this class also provides encoding (“escape”) procedures not only for HTML but also for Java, JavaScript and XML.

Second, we also draw attention to the `Utils` class by Purple Technology. The `com.purpletech.util.Utils` provides the following methods for encoding – taken from the API documentation [7]:

```
static java.lang.String htlescapse(java.lang.String s1)
    Turns funky characters into HTML entity equivalents.
static java.lang.String htmlunescape(java.lang.String s1)
    Given a string containing entity escapes, returns a string containing the actual
    Unicode characters corresponding to the escapes.
```

The source code for these methods can be found in [8].

### 2.2.3.3.1 Requisites

Integration of the Apache Commons Lang Java library into the individual application development and deployment environment.

### 2.2.3.3.2 To be avoided

See section 2.2.3.2.3.

## 2.3 Test Procedure

Please note that the section Test Procedures in this document gives an introduction to the usage of the test procedures described.

### 2.3.1 Outline

Whenever the response of a Servlet/JSP is based on former user inputs, these user inputs must not be written into the response without being encoded adequately. If such inputs are written into a Servlet's/JSP's output without being encoded, XSS attacks become possible (compare with 2.1.1.1 and 2.1.1.2). Moreover, certain web applications might not allow all kinds of inputs. This requires that corresponding filter methods be applied to the application's inputs.

Test Attributes	
Platform	J2EE/Java
Operating Systems	All, no restrictions
Type of test	Code Review
When should be tested	From initial to final implementation phase
Tool support	Assisting usage of source code scanners and static analysis tools is possible for more sophisticated courses of action

### 2.3.2 Test Execution

The occurrences of the following methods are considered to be most relevant for this test procedure:

- All output methods of the classes
  - `javax.servlet.ServletOutputStream`
  - `java.io.PrintWriter`

*Note:* Within a Servlet/JSP, corresponding objects of these output classes are obtained, for instance, by the following methods of the class `javax.servlet.ServletResponse`

  - `getOutputStream()`
  - `getWriter()`
- The following methods of the class `javax.servlet.http.HttpServletResponse`
  - `addCookie(Cookie cookie)`

- `addHeader(java.lang.String name, java.lang.String value)`
- `sendRedirect(String location)`
- `setHeader(java.lang.String name, java.lang.String value)`

Any occurrences of these methods have to be examined to see whether character strings which are written into the web application's response have been encoded adequately (compare with 2.2.3).

### 2.3.2.1 Special Cases in Practice

Please note that it might be possible that the character strings written into the response are filtered manually. For instance, it is possible that the developer is testing for a whitelist of allowed strings (e.g., allowed HTML tags). It might also be possible that, in individual circumstances, application developers are bound to use specific or proprietary filter methods.

Thus, finding all relevant code lines can be done automatically, e.g., with the use of an editor's search function, etc., or more sophisticated tools. However, when a special kind of trustworthy filter method is being used for an application, the search for potential vulnerabilities possibly has to be done manually in consideration of these methods.

Please note that we do not assess the security and trustworthiness of such individual filter methods in this test procedure. However, please also note that such filters are very critical for the security of an application. Thus, the filters to be used should also be tested in another test procedure.

### 2.3.3 Interpretation of Test Results

The interpretation of the test results, and thus the assessment of the potentially vulnerable code pieces which have been found, depend on the origin of the character strings which are written into the response. It is necessary to check whether these strings come from:

- a constant
- an external input, such as a user input within a request parameter
- the application's database

The following cases are considered to be secure:

- The string is a constant.
- The string is part of an external input but it is output encoded and/or filtered adequately (compare with 2.1.1.1 and 2.1.1.2).
- The string comes from the application's database and the database contains only encoded and/or filtered data. Otherwise, strings which come from the database have to be treated as external inputs and thus have to be output encoded and/or filtered adequately.

Please note that the methods listed above in 2.3.2 allow testing for client-side and server-side XSS vulnerabilities. If the occurrences which have been found according

---

to the instructions above cannot be considered secure, the corresponding pieces of code have to be reengineered.

## 2.4 References

- [1] The Open Web Application Security Project (OWASP), <http://www.owasp.org>
- [2] The Web Application Security Consortium (WASC),  
<http://www.webappsec.org/>
- [3] The OWASP Top Ten, <http://www.owasp.org/documentation/topten.html>
- [4] The Web Security Threat Classification,  
<http://www.webappsec.org/projects/threat/>
- [5] Apache Jakarta Commons Lang, <http://jakarta.apache.org/commons/lang/>
- [6] HTML 4.01 Entities Reference,  
[http://www.w3schools.com/tags/ref\\_entities.asp](http://www.w3schools.com/tags/ref_entities.asp)
- [7] Purple Technology “Utils” library,  
<http://www.purpletech.com/code/doc/index.html>
- [8] Purple Technology “Utils” class source code,  
<http://www.purpletech.com/code/src/com/purpletech/util/Utils.java>

## 3 Cookie Security

Brief description of related security attacks	<ul style="list-style-type: none"> <li>• Stealing of cookies</li> <li>• Poisoning of cookies</li> <li>• Code injection via cookies</li> <li>• XSS attacks via cookies</li> </ul>
This topic is related to	<ul style="list-style-type: none"> <li>• Dealing with cookies (within HTTP requests)</li> <li>• Session management</li> <li>• Input filtering, validation, and encoding of cookie data</li> </ul>
Degree of severity	<b>HIGH</b>
Consequential and potential damage	<ul style="list-style-type: none"> <li>• Identity theft and impersonation</li> <li>• Session hijacking</li> <li>• Unauthorized access to the application and its data</li> <li>• Deception of application users</li> </ul>
Affected packages (among others)	<code>javax.servlet.*</code> , <code>javax.servlet.http.*</code>
WHEN NOT TO READ	<ul style="list-style-type: none"> <li>• If the application does not deal with cookies and thus neither set nor read cookies</li> </ul>

### 3.1 Introduction

HTTP is a stateless protocol. In 1994, Netscape invented a mechanism called “cookie” as a method for session tracking. A cookie is a small piece of information usually created by the web server and stored in the client’s web browser. Each time the client contacts the web server, the cookie’s data is passed back to the server. The cookie contains information used by web applications to persist and pass variables back and forth between the client and the web application. Two types of cookies are known:

- Persistent cookies, which are stored in a file on the client-side until an expiry date
- Session cookies, which are only stored by the client until the current session with the web application is finished

As a result of the cookie structure and their usage in practice, all data stored in a client cookie could be easily read and manipulated. The risk of data tampering and even information disclosure is very high.

#### 3.1.1 Variants

In the following, we will discuss some examples of attacks related to flaws in cookie security. These variants differ widely, as cookie security affects many areas of web application security.

##### 3.1.1.1 Cookie poisoning

Attacks of this category arise because parts of the content of the client’s cookie can be written through application URLs. This provides an opportunity for easy cookie manipulation not only to application users but also to external attackers, which could

mislead users to execute malicious URL calls. Depending on the severity of the application's vulnerability, such attacks could lead to XSS, stealing of cookies, and session hijacking.

#### Example of bad code

We assume a web application uses URLs to write client cookies. For instance, a URL like

```
http://myapp.eu/shopping/calcbill?param1=12%2E95
```

sets a cookie containing the information 12.95. The corresponding piece of code could look like this:

```
String cookcontent = request.getParameter("param1");  
// ...  
Cookie newcookie = new Cookie("testcookie1", cookcontent);  
newcookie.setMaxAge(1800);  
response.addCookie(newcookie);
```

Further, we assume that another page of the application displays the cookie content. At this point, e.g., XSS attacks as shown in section 2.1.1.1 become possible. Similar to the example there, an attacker is able to create application URLs containing malicious XSS code. If an attacker misleads an application user to execute those URLs, the XSS code is stored within the user's cookie. The difference compared to the example in 2.1.1.1 is that the XSS code is only executed if the user calls up another application page which embeds the cookie content into its HTML code. Assuming, for instance, that validating and filtering of the cookie content is never done, the XSS code will be executed on the client side when the corresponding HTML page is opened.

#### Sample attack

Compare with section 2.1.1.1.

### **3.1.1.2 Cookie manipulation and Missing Input Validation**

Security flaws of this category occur because developers assume a cookie's content cannot be changed by the client. If a cookie is regarded as a trustworthy part of the application, consequences can range from Missing Input Validation inside of the web application's procedures to code injection attacks. Two causes of vulnerabilities of this category can be identified:

- Too much information is stored in the client's cookie, instead of storing the information on the server-side
- Possible manipulations of a client's cookie by a (regular) web application user are disregarded

#### Example of bad code

We assume a Servlet allows access to specific operations or data only to privileged and thus specific application users, e.g., administrators. After the login procedure, the application sets a client cookie which includes information about the user's privileges.

Thus, the client sends its cookie for all further communication with the application and the application identifies the user and its privileges by the cookie. A vulnerable Servlet could look like this:

```
boolean isadmin = false;          // Status of the user
String cookcontent = "";

Cookie[] cookie = request.getCookies();
if (cookie!=null){
    Cookie myCookie=null;
    if ((cookie[0].getName().equals("testcookie1"))){
        myCookie = cookie[0];
        cookcontent = myCookie.getValue();
        if (cookcontent.equals("ADMIN=yes"))
            isadmin=true; // Setting the user status
    }
}

if (isadmin){
    // Privileged operations here
}
else{
    // Non-privileged operations here
}

// ...
```

### Sample attack

We assume the Servlet above sends the following cookie after a non-privileged user successfully logged into the web application:

```
Cookie: ADMIN=no
```

As the cookie is stored on the client-side, the user is able to modify this cookie, e.g., as follows:

```
Cookie: ADMIN=yes
```

Now, the Servlet would allow this user access to its privileged operations or data. Hence, it is obvious that the Servlet code succumbs to a logic error which is the assumption that the client cookie's content can be trusted and cannot be modified.

To make this point clear, we give another simple example. We assume that a web shop application stores shopping cart information, including the pricing, in client cookies. The cookie content for the web shop could look like this:

```
item1_ID=12369&item1_pr=27.95&item2_ID=10334&item2_pr=19.95
```

Obviously, the total amount of both items is 47.90. Let us assume the checkout page calculates the final price by using the amounts inside of the client's cookie. Again, the client's user could manipulate the cookie's content, e.g., in the following way:

```
item1_ID=12369&item1_pr=0.95&item2_ID=10334&item2_pr=1.95
```

In this scenario, this would result in a total amount of 2.90 for all items calculated by the web shop.

#### 3.1.1.2.1 Code injection via cookies

Depending on the web application, cookie manipulations can lead not only to attacks due to Missing Input Validation (see section 3.1.1.2). Another possibility is code injection attacks via cookies. We provide an example for the case of SQL code injection. Vulnerabilities of this category of cookie security basically reflect the same vulnerabilities as (SQL) code injection in general. However, cookies are a specific way of transferring the attacker's code into the vulnerable application.

##### Example of bad code

This short Servlet code example is similar to the examples in section 1.1.1.

```
Cookie[] cookie = request.getCookies();
String cookcontent = "";

if (cookie!=null){
    Cookie myCookie=null;
    if ((cookie[0].getName().equals("testcookie1"))){
        myCookie = cookie[0];
        cookcontent = myCookie.getValue();
    }
}

// Preparing the database connection
// Preparing the Statement object s

ResultSet rs = s.executeQuery("SELECT "
    + cookcontent
    + " FROM testtable1"
    + " WHERE ... ;");
// At this point SQL injection is possible
// via the cookie's content
// ...
```

##### Sample attack

Compare with section 1.1.1.

#### 3.1.1.3 Information disclosure by cookies

In case persistent or session cookies set on the client contain sensitive information – because the web application has written such information into the client's cookie – an attacker might be able to steal this information. This is possible, e.g., by a client-side

XSS attack via JavaScript code (compare with section 2.1.1.1), as JavaScript allows access to client cookies. As there is no need for a web application to store such sensitive data on the client side, information disclosure can be avoided.

Nevertheless, if client-side XSS is possible, an attacker may be able to steal the users' cookies. When a cookie contains only session information, the attacker still might be able to perform session hijacking (see also section 3.2.3.1).

### 3.1.2 Damage in the case of Non-Compliance

The damage in the case of insufficient cookie security has been described in general in the previous sections. We now summarize these issues as follows:

- Access the user's cookies
- Cookie manipulation
- Cookie poisoning
- Session hijacking
- Information disclosure
- Stealing of cookies
- Code injection via cookies
- Identity theft and thus impersonation and access to the web application

## 3.2 Prevention, Countermeasures, Solutions

We want to refer to the external information in [1] [2] [3] [4], which provides a minimum standard for web application security. Besides the dangerous implications of flaws in cookie security described in section 3.1.1, it must be remembered that cookie security is related to other attacks, like XSS and code injection. This interrelationship was demonstrated in the subsections of 3.1.1.

### 3.2.1 Description

The general guideline for protecting applications from vulnerabilities regarding cookie security is similar to the case of XSS: "Do not trust any data coming from outside your application code." Thus, the general countermeasure is again adequate input validation and filtering. This also applies to several other security topics [3] [4].

Moreover, there are some specific guidelines for cookie security regarding the attack variants described in section 3.1.1. These guidelines will now be introduced.

### 3.2.2 Platform Limitations or Extensions

All solutions described here are built-in platform solutions. Thus, no extensions are needed. No limitations are known.

### 3.2.3 Solution Variants

Overview:

Solution	Platform	Libraries
<i>Basic rules have to be considered</i>	Java Servlets/JSPs (all versions)	n/a

### 3.2.3.1 Rules for cookie security and session management

The best practice to avoid security vulnerabilities via cookies is to be suspicious of data stored in cookies. Thus, the most important guideline for web applications is: Do not store ANY data in a client cookie except session IDs. Attackers can easily manipulate client-side cookies. All of the information which the web application server needs for the client communication should be stored on the server-side. Cookies only should be used for maintaining session IDs.

Thus, if data is stored in a client cookie, the application developer should always question whether the application needs more information within a client's cookie than only a session ID. As an alternative to cookies, the developer could consider whether URL rewriting could be used for managing session IDs on clients. Please see section 3.2.3.1.2 for an introduction to the J2EE's session management interface.

In case the web application really needs specific information inside of a client cookie, aspects related to information disclosure threats have to be taken into account. Therefore, never store any confidential data in a cookie, e.g.:

- Non-public IP addresses of target servers
- Host names
- System IDs

Many web applications need to store session information on the server-side to provide their functionality (e.g., web shops). However, even if only session IDs are maintained on the client-side, the application developer should be aware of the possibility of session hijacking attacks. Session hijacking can be the result of two scenarios:

- The attacker guesses the victim's (an application client) current session ID
- The attacker is able to steal a client's session ID (e.g., by reading the client's cookie via XSS)

The first scenario can be avoided by using non-computable session IDs. The second scenario can only be mitigated, as stealing of session IDs on the client cannot be prevented by the web application in general. To mitigate session hijacking attacks in this scenario, we have to refer to the corresponding Java web application server being used, as the server is responsible for creating session IDs. Developers should be aware of the information the server uses for creating session IDs (e.g., random number generator, hash value of the client's IP address etc.), because this is important for security in the second scenario: If the "stolen" session ID can only be used by the original client, even in the second scenario no session hijacking attacks would be possible.

Another basic rule for web applications is to limit the session time as far as possible while still respecting the application's usability. Adequate procedures are:

- Use session cookies instead of persistent cookies
- Use idle timeouts for applications that expose private data or that may cause identity theft if left open
- Offer a logout mechanism to the user, to manually shorten the time until a session timeout will end the session automatically

### 3.2.3.1.1 Requisites

Technical Requisites	Platform Release	Features/Interfaces to be used
n/a	Java (all)	Interface <code>javax.servlet.http.HttpSession</code> , class <code>javax.servlet.http.Cookie</code>

### 3.2.3.1.2 Procedure

As the basic rules described in section 3.2.3.1 mostly refer to the guideline to store only session IDs on the client-side and to store all other session information on the server-side, we want to explain how to do this in Java. The J2EE standard provides session tracking mechanisms which manage:

- The storage of session information at the server
- The mapping of the session information to a certain client

Mapping is realized by tracking a client and storing a single session ID. The Java tracking mechanisms allow two methods for storing IDs:

- Setting client cookies
- Rewriting URLs

Moreover, the storing method is chosen automatically, depending on the settings on the client-side. For instance, if cookies are disabled at the client's browser, URL rewriting is used.

The `HttpSession` interface of the J2EE allows tracking sessions with an object residing only on the server side. This implies that all information relevant to a session can be stored inside of an `HttpSession` object. The correlation between the specific object and the client is performed automatically.

We now present a brief example for the usage of `HttpSession`. We assume a web application includes a proprietary class called `MyShoppingCart`. An instance of this class may take all important information of one session of a certain client.

As every Servlet provides access to the `HttpServletRequest`, e.g., by an object called `request`, it automatically provides access to the corresponding `HttpSession` object. The session object can easily be read out, as well as created and stored. It also takes the `MyShoppingCart` object and thus stores the session information. These procedures are demonstrated in the following code example:

```
import javax.http.servlet.HttpSession;

// ...

// Accessing the session object for this request
HttpSession session = request.getSession(true);

// Reading the session information to this client's request
```

```
MyShoppingCart cartobject =
    (MyShoppingCart)session.getAttribute("TheShoppingCart");

// ...
// Creating/Setting new session information
if (cartobject==null){
    cartobject = new MyShoppingCart();
    session.setAttribute("TheShoppingCart", cartobject);
}

// ...
```

For more information please refer to the J2EE session tracking API [5].

#### 3.2.3.1.3 To be avoided

Sometimes cookies may contain personal information, if programmers ignore the advice never to store any confidential data in a cookie. This has to be avoided at all costs.

The extent of cookie manipulation ranges from session tokens to arrays that make authorization decisions. Thus, developers should only store as much information in a client's cookie as is absolutely necessary. Do not trust any data coming from the client because it could be manipulated. This implies that input validation has the same importance here as in the case of, e.g., XSS or SQL injection attacks (compare with sections 1 and 2). Please note that the data inside of cookies could also be used for, e.g., SQL injections (see section 3.1.1.2.1). If the web application is vulnerable to SQL injections, cookies are only another entry point into the application.

If client cookies could be set by URLs, attackers may be able to mislead application user's to execute malicious URL calls. Thus, external attackers might also be able to attack the vulnerable web application by poisoning the clients' cookies.

#### 3.2.3.1.4 More information

Please see Sun's Java documentation [5] [6] for more details, especially on the usage of the `javax.servlet.http.HttpSession` interface and the `javax.servlet.http.Cookie` class.

### 3.3 Test Procedure

Please note that the section Test Procedures in this document gives an introduction to the usage of the test procedures described.

#### 3.3.1 Outline

For testing all issues in cookie security, the three kinds of vulnerabilities described in section 3.1.1 have to be considered:

- cookie poisoning
- cookie manipulation
- code injection via cookies

For the latter, please refer to sections 1.3 and 5.3 and perform the corresponding test procedures, but with cookies considered as input source for potential injections. The following procedure deals with the first two kinds of vulnerabilities.

Test Attributes	
Platform	J2EE/Java
Operating Systems	All, no restrictions
Type of test	Code Review
When should be tested	From initial to final implementation phase
Tool support	Assisting usage of source code scanners and static analysis tools is possible for more sophisticated courses of action

### 3.3.2 Test Execution

The occurrences of the following methods of `javax.servlet.http.Cookie` inside a Servlet/JSP are considered to be most relevant for this test procedure:

- the constructor `Cookie(String name, String value)`
- `setValue(String newValue)`
- `String getValue()`

Character strings written into a cookie might be used for cookie poisoning when these strings come from an unfiltered user input. Character strings read from a cookie might be used for manipulation attacks when these strings are not filtered adequately.

#### 3.3.2.1 Special Cases in Practice

Please note that it might be possible that character strings written into a cookie or read from a cookie might be filtered manually. For instance, it is possible that the developer is testing for a list of allowed cookie contents. It might also be possible that, in individual circumstances, application developers are bound to use specific or proprietary filter methods.

Thus, finding all relevant code lines can be done automatically, e.g., by the use of an editor's search function, etc., or more sophisticated tools. However, when a special kind of trustworthy filter method is being used for an application, the search for potential vulnerabilities possibly has to be done manually in consideration of these methods.

Please note that we do not assess the security and trustworthiness of such individual filter methods in this test procedure. However, please also note that such filters are very critical for the security of an application. Thus, the filters to be used should also be tested in another test procedure.

### 3.3.3 Interpretation of Test Results

It is secure to write content into a cookie that comes from a constant or that comes from user inputs which have been filtered previously. In all other cases, pieces of code which have been found in 3.3.2 have to be reengineered.

All character strings which are read from a cookie have to be filtered for allowed cookie content. Since allowed content depends on the individual application, we cannot give more information here. However, all pieces of code which process unfiltered cookie content have to be reengineered.

### 3.4 References

- [1] The Open Web Application Security Project (OWASP), <http://www.owasp.org>
- [2] The Web Application Security Consortium (WASC),  
<http://www.webappsec.org/>
- [3] The OWASP Top Ten, <http://www.owasp.org/documentation/topten.html>
- [4] The Web Security Threat Classification,  
<http://www.webappsec.org/projects/threat/>
- [5] Servlet API documentation, <http://java.sun.com/products/servlet/2.2/javadoc/>
- [6] The J2EE 1.4 Tutorial, <http://java.sun.com/j2ee/1.4/docs/tutorial/doc/>

## 4 Resource Tampering

Brief description of related security attacks	<ul style="list-style-type: none"> <li>Unauthorized access to application resources, like files or folders</li> </ul>
This topic is related to	<ul style="list-style-type: none"> <li>Dealing with HTTP requests</li> <li>Input filtering and validation</li> <li>Web application deployment</li> </ul>
Degree of severity	<b>HIGH</b>
Consequential and potential damage	<ul style="list-style-type: none"> <li>Information disclosure</li> <li>Unauthorized modification of application data</li> </ul>
Affected packages (among others)	<code>javax.servlet.*</code>
WHEN NOT TO READ	<ul style="list-style-type: none"> <li>If the application does not deal with HTTP requests at all</li> </ul>

### 4.1 Introduction

A web application usually consists not only of Servlets, JSPs, and a database backend, but also of resources like style sheets, image files, HTML page fragments or a function for browsing directories. There are two basic possibilities for realizing access to those resources:

- Implementation inside of the application code
- Using resource handling functions of the web server (deployment)

Web servers generally are set up to restrict public access to a specific portion of the web server's file system. In a Resource Tampering attack, like path traversal, an attacker manipulates a URL in such a way that the web server reveals the content of a file anywhere on the server. Thus, this applies also to files which reside outside of the web application's root directory. Path traversal attacks take advantage of special-character sequences in URL input parameters, cookies, or HTTP request headers. Even if a web server properly restricts path traversal attempts in the URL path, any application that exposes an HTTP-based interface is also potentially vulnerable to such attacks.

#### 4.1.1 Variants

In the following we want to demonstrate the most common attack in the category of Resource Tampering – path traversal. As the deployment of a web application is very individual (e.g., choice of platform or web server) we have created an example for resource handling inside of a Servlet.

##### 4.1.1.1 Path Traversal

A common path traversal attack uses the `../` character sequence to alter the document or resource location requested in a URL. Most web servers block this method by using escaping sequences, but, generally, alternate encodings of the `../`

sequence can also bypass basic security filters and thus have to be taken into account.

The following example shows a simple handling function inside of a Java Servlet where even specific sequences like `../` or different encodings are not needed for a successful attack.

#### Example of bad code

We assume a web application should be easily deployed independent of its web server environment. Thus, the application provides its own resource handling function:

```
// ...
// Resource handling inside of the application code

// URL requests resource in "param1"
String filename = request.getParameter("param1");

InputStream in = new BufferedInputStream(new
                                         FileInputStream(filename));
String ct = URLConnection.guessContentTypeFromStream(in);

response.setContentType(ct);
byte arbitraryfile[] = new byte[in.available()];
in.read(arbitraryfile); // 1)

OutputStream outbin = response.getOutputStream();
// The resource's content (binary, plaintext) is the response
outbin.write(arbitraryfile); // 2)
```

#### Sample attack

We assume the web application is run on a UNIX server and the resource handling function above is available via the URL `getfiles`. An attack for the Servlet example above would be the following. We omit HEX encoding for the URL:

```
http://www.myapp.com/getfiles?param1=/etc/passwd
```

As a result of the URL call, the web application would return the content of the file `/etc/passwd` to the client. This happens because, in line 1), the Servlet reads in the corresponding text file, and, in line 2), the Servlet writes the file's content into its HTTP response.

#### **4.1.1.1 Attack variants of path traversal**

Web applications can be deployed in Windows and UNIX systems. Different characters can have specific meanings in these systems. For instance, in UNIX systems, the slash character is used for addressing paths or files (e.g., `/etc/passwd`), while in Windows systems, the backslash character is used (e.g., `C:\autoexec.bat`) for this purpose.

Moreover, web applications are addressed by URLs which have to be escaped HEX encoded. Usually, this is only obvious when special characters like `"` or `/` are part of,

e.g., a URL parameter's content. For instance, the sample URL above usually would be called in this form:

```
http://www.myapp.com/getfiles?param1=%2Fetc%2Fpasswd
```

The following table summarizes the most important ASCII characters which are usually used in path traversal attacks (also depending on the server system) including their corresponding escaped HEX encoding:

ASCII character	Escaped HEX encoding
NUL	%00
Space	%20
%	%25
.	%2E
/	%2F
:	%3A
\	%5C

In case the web application has implemented security functions for validating path access via URLs, several attack variants could be of interest to an attacker wanting to bypass those functions. The most important issues which could be part of attacks are:

- For addressing paths, both UNIX and Windows systems offer the possibility of addressing the current directory by `./` and `.\` as well as addressing the parent directory by `../` and `..\`
- URLs could contain different encodings, for instance
  - ASCII characters without escaped HEX encoding
  - Escaped HEX encoding of special characters (e.g., `%2e%2e%2f` → `../`)
  - Double escaped HEX encoding (e.g., `%255C` → `%5C` → `\`)
  - Unicode encoding

In the following we provide some *sample* attacks for these issues.

#### Sample attack: Relative directories

We assume a web application allows directory browsing but the folder `information` should never be accessible from the outside. Thus, the folder is not linked by any pages of the web application. An attacker might be able a) to guess the name of or b) to discover (e.g., by browsing) the folder. A path traversal attack using `../` could look like this:

```
http://mywebapp.tv/../../../../information/disclosure.mdb
```

Usually, web servers filter the `../` or `..\` input. However, this is not always true for web applications which manage directory browsing via their own functionality.

### Sample attack: Escaped HEX encoding

A similar attack could use escaped HEX encoding (%2F → /). A successful attack would rely on the fact that input validation of the URL is done *before* the translation of the escaped characters and thus misses the meaning of the %2F characters:

```
http://mywebapp.tv/..%2F..%2F..%2Finformation/disclosure.mdb
```

Moreover, other special characters could be used for path traversal attacks within URLs. For instance, attacks could also use the escaped encoded NUL character (%00). We assume a web application checks and verifies the extension of a requested file for returning only .html or .css files. This would prevent the application from displaying any other files of the server's file system and thus, e.g., script code of the application. An attack using the NUL character would be the creation of a URL that looks like the request for a .html or .css file. For instance, this could be a regular application URL:

```
http://mywebapp.tv/cgi-bin/show.cgi?web/info.html
```

An attack for gaining information about the CGI script could look like this:

```
http://mywebapp.tv/cgi-bin/show.cgi?../  
cgi-bin/show.cgi%00.html
```

Assuming the check for the valid file extensions (.html and .css) is performed first, the attack would be successful when the web application (in this case the CGI script itself) would stop reading the parameter string as soon as it reaches the NUL character. If so, the script would return its source code.

### Sample attack: Double escaped HEX encoding

The success of an attack using double escaped HEX encoding is due to the fact that a) the application's web server (unintentionally) translates escaped encoded characters twice or b) the web server's operating system understands escape codes. We reuse the first attack example ("Relative directories") with double escaped HEX encoding:

```
http://mywebapp.tv/..%252F..%252F..%252F/  
information/disclosure.mdb
```

In case a), ..%252F would be translated to ..%2F and finally to ../. In case b), ..%252F only would be translated to ..%2F but the web application passes this URL to the web server's file system which would understand this escaped encoded request. Both cases assume that input validation for the URL is done after the *first* translation. Only if this is true, would the path traversal attack be successful.

### Sample attack: CGIs for resource handling

The last example demonstrates how complex the prevention of path traversal attacks can be. We assume that a web application does not allow direct access to the web server's file system. Thus, no directory browsing functions are activated and URLs can never address server resources directly. We assume further that the application uses static HTML pages on the file system for building the actual application pages,

e.g., by adding header and footer to the static HTML pages. This is done by a Servlet:

```
http://mywebapp.tv/showfiles?web/info.html
```

The Servlet behind this URL reads the request parameter and returns the corresponding HTML page it has produced. An obvious path traversal attack on a UNIX system could be the following, perhaps followed by several guesses:

```
http://mywebapp.tv/showfiles?../../../../etc/passwd
```

This implies that the Servlet must provide functionality for validating the inputs which are given by the URL's parameter. It has to detect that, e.g., the file `etc/passwd` must not be returned. This implies, further, that all possible attack variants for path traversal must be completely filtered by the Servlet itself.

#### 4.1.2 Damage in the case of Non-Compliance

The main damage takes place in the area of information disclosure. This implies that information is accessed which is not intended to be accessed. This information could be, for instance:

- Other users' data that is accessed via Resource Tampering attacks by users without corresponding access rights given by the application
- All application data in general, e.g., sensitive user inputs, by unauthorized access to application resources, like database files on the application server, or source code to JSPs and scripts, etc.
- The application's deployment environment, like the web server's file system

## 4.2 Prevention, Countermeasures, Solutions

We refer to the external information in [1] [2] [3] [4], which provides a minimum standard for web application security.

### 4.2.1 Description

The general guideline for protecting applications from vulnerabilities regarding Resource Tampering is similar to the case of XSS: "Do not trust any data coming from outside the application code." Thus, the general countermeasure is again adequate input validation and filtering. This is also valid for several other security topics [3] [4].

Moreover, there are some specific guidelines for preventing Resource Tampering vulnerabilities and these will now be introduced.

### 4.2.2 Platform Limitations or Extensions

All solutions described here are built-in platform solutions. Thus, no extensions are needed. No limitations are known.

### 4.2.3 Solution Variants

Overview of the "exemplarily" described solutions:



Solution	Platform	Libraries
<i>Using Resource Handlers</i>	Web servers	n/a

#### 4.2.3.1 Rules

We want to give some general recommendations to prevent Resource Tampering attacks:

- The implementation of file access functionality that is based on user inputs should be omitted unless there is no other alternative
- The access rights of the web application on the web server should be restricted to prevent access to resources other than the web application's

If developers have to build individual security functions for validating resource and path inputs to the web application, then the best way to do so is "normalization". This implies that the given path string has to be normalized (compare with section 4.1.1.1.1). A normal path string has:

- No empty segments (e.g., occurrences of //)
- No segments equal to .
- No segments equal to ..

Moreover, if developers must allow user input for file access, the following aspects have to be considered:

- Constrain the user input to a list of allowed files and paths (whitelist)
- Define a codepage (e.g., character set ISO-8859-1) to clearly decide which character encoding is being used
- Filter the input for malicious meta-characters (compare with section 4.1.1.1.1)

#### 4.2.3.2 Resource handling

Resource handling is a functionality provided by several web servers to allow web applications access to file resources. This is an easy way for implementing and deploying web applications in a secure way when considering Resource Tampering attacks. We now show such functionality by introducing two web servers exemplarily.

##### 4.2.3.2.1 Requisites

Technical Requisites	Platform Release	Features/Interfaces to be used
Web servers' resource handlers	Java (all)	

##### 4.2.3.2.2 Procedure

To describe the resource handling functionality of web servers, we exemplarily chose two common Open Source servers: Apache Tomcat [5] and Jetty [6]. Both are HTTP Servers as well as Java Servlet containers.

When deploying web applications, one has to be aware of the web server's configuration in general. Especially with respect to Resource Tampering attacks, the

setting of the web application's runtime environment is critical. For instance, the Servlet container Tomcat allows enabling and disabling its "directory listing" feature [7]. Similar options can be found in any web server, e.g., also in Microsoft's Internet Information Server (IIS).

In the following we describe exemplarily the deployment of a web application and the usage of resource handlers. If not otherwise noted, the following is true for Apache Tomcat and Jetty. A Java web application is usually set up inside of a `WEB-INF` directory. We assume that the package of an application called `MyWebApp` includes the following directories and files:

```
MyWebApp\:  
  style.css  
MyWebApp\images\  
  arrow1.jpg  
  arrow2.jpg  
MyWebApp\WEB-INF\  
  classes\MyWebApp\Main.class  
  lib\mysql-connector-java-3.1.10-bin.jar  
  web.xml
```

We assume further that these are all files of the single web application and thus no other files are needed for running `MyWebApp`. The configuration file `web.xml` is the most important file for setting up the properties of the Servlet-based application. Please note that application properties could also be managed by central configuration files of Tomcat or Jetty. Moreover, it is possible to set general properties for all web applications running on a server. These considerations are omitted here.

By convention, the `WEB-INF` folder and its content is not readable outside the web server and thus never accessible for an application user or attacker. The `web.xml` file could look like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>  
  
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web  
  Application 2.3//EN"  
  "http://java.sun.com/dtd/web-app_2_3.dtd">  
  
<web-app>  
  <servlet>  
    <servlet-name>MyWebApp</servlet-name>  
    <servlet-class>  
      MyWebApp.Main  
    </servlet-class>  
  
    <init-param>  
      <param-name>InstallDir</param-name>  
      <param-value>  
        /webapps/MyWebApp/WEB-INF
```

```
        </param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>MyWebApp</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>
</web-app>
```

This would allow access the web application, e.g., on the server `example.com` via this URL:

```
http://example.com/MyWebApp
```

To explicitly deactivate the directory browsing function of Tomcat for this web application (which is the standard setting), we would have to add the following piece inside the `servlet` tag:

```
    <init-param>
        <param-name>listings</param-name>
        <param-value>>false</param-value>
    </init-param>
```

Now, the setup of the web application is complete. The application resources in `MyWebApp\*` are automatically accessible. This implies that *only* the following URLs are valid for this web application:

```
http://example.com/MyWebApp/style.css
http://example.com/MyWebApp/images/arrow1.jpg
http://example.com/MyWebApp/images/arrow2.jpg
```

In the path shown above, traversal attacks are unsuccessful and prevented by the server. Please note that the application's Servlet is running under certain system access rights. This implies that if the Servlet contains additional functionality for accessing system resources, it is able to do so. Moreover, if it is returning system resources within its responses to clients depending on user requests, and thus user inputs, Resource Tampering attacks are possible. For instance, if the application owns access rights for the system file `/etc/passwd`, it is able to return it to its users.

It is also possible to restrict the access rights of the Servlet by using the Java J2EE built-in Security Enhancement functionalities. These are also provided by Tomcat within its policy configuration, e.g.:

```
grant {
    permission java.io.FilePermission "-", "read,write";
}
```

Please see [9] and [5] for details for setting up Java Security Enhancements for Servlets.

Jetty offers another possibility for creating resources and resource handlers for Java web applications. Jetty provides a set of Java methods which enable the application developer to create resources within his or her Java code. Please refer to [8] for more information about the following Jetty classes:

- `org.mortbay.http.handler.ResourceHandler`
- `org.mortbay.util.FileResource`
- `org.mortbay.util.Resources`

#### 4.2.3.2.3 To be avoided

Building individual resource handling functions should be neglected (compare with the code sample of section 4.1.1.1).

#### 4.2.3.2.4 More information

Please see 4.4 References for more details about the possibilities of configuring resource handlers of web servers.

### 4.3 Test Procedure

Please note that the section Test Procedures in this document gives an introduction to the usage of the test procedures described.

#### 4.3.1 Outline

File access in Java is provided by the following classes:

- `File`
- `RandomAccessFile`
- `FileInputStream`

The constructor of the respective class expects a string parameter which contains the corresponding filename, including the full path to the file. This parameter is not dangerous if it is filled by a constant or has been previously filtered. However, if the parameter contains unfiltered input, an attacker might be able to access arbitrary files.

Test Attributes	
Platform	J2EE/Java
Operating Systems	All, no restrictions
Type of test	Code Review
When should be tested	From initial to final implementation phase
Tool support	Assisting usage of source code scanners and static analysis tools is possible for more sophisticated courses of action

### 4.3.2 Test Execution

Any occurrences of the constructors of `File`, `RandomAccessFile`, and `FileInputStream` have to be found. If the filename parameter is not hard-coded or not read from a constant, or unfiltered, a potentially vulnerable piece of code has been located.

#### 4.3.2.1 Special Cases in Practice

Please note that it might be possible that the filename parameter is filtered manually. For instance, it is possible that the developer is testing for a list of allowed filenames. It might be also possible that, in individual circumstances, application developers are bound to use specific or proprietary filter methods.

Thus, finding all relevant code lines can be done automatically, e.g., by the use of an editor's search function, etc., or more sophisticated tools. However, when a special kind of trustworthy filter method is being used for an application, the search for potential vulnerabilities possibly has to be done manually in consideration of these methods.

Please note that we do not assess the security and trustworthiness of such individual filter methods in this test procedure. However, please also note that such filters are very critical for the security of an application. Thus, the filters to be used should also be tested in another test procedure.

### 4.3.3 Interpretation of Test Results

The interpretation of the test results and thus the assessment of the potentially vulnerable code pieces which have been found depend on the origin of the character strings which are used to construct the filename parameter. It is necessary to check whether these strings come from:

- a constant
- an external input, such as a user input within a request parameter
- the application's database

The following cases are considered to be secure:

- The string is a constant.
- The string is part of an external input but it is filtered adequately (compare with 4.2.3.1).
- The string comes from the application's database and the database contains only filtered data, including filtered filenames. Otherwise, strings which come from the database have to be treated as external inputs and thus have to be filtered adequately.

If the occurrences which have been found according to the instructions above cannot be considered as secure, the corresponding pieces of code have to be reengineered.

---

#### 4.4 References

- [1] The Open Web Application Security Project (OWASP), <http://www.owasp.org>
- [2] The Web Application Security Consortium (WASC),  
<http://www.webappsec.org/>
- [3] The OWASP Top Ten, <http://www.owasp.org/documentation/topten.html>
- [4] The Web Security Threat Classification,  
<http://www.webappsec.org/projects/threat/>
- [5] Apache Tomcat, <http://tomcat.apache.org/>
- [6] Jetty, <http://jetty.mortbay.org>
- [7] Apache Tomcat FAQ, <http://tomcat.apache.org/faq/misc.html>
- [8] Jetty API documentation, <http://jetty.mortbay.org/javadoc/index.html>
- [9] Java Security Enhancements,  
<http://java.sun.com/j2se/1.5.0/docs/guide/security/index.html>

## 5 Code Injection

Brief description of related security attacks	<ul style="list-style-type: none"> <li>• Injection and execution of arbitrary system commands on web application servers</li> <li>• Attacking internal or sensitive hosts via compromised web servers</li> </ul>
This topic is related to	<ul style="list-style-type: none"> <li>• Dealing with HTTP (GET/POST) request parameters</li> <li>• Input validation and filtering</li> </ul>
Degree of severity	<b>HIGH</b>
Consequential and potential damage	<ul style="list-style-type: none"> <li>• Tampering of web servers and web applications</li> <li>• Unauthorized access to application data and data theft</li> <li>• Compromising internal or sensitive hosts</li> </ul>
Affected packages (among others)	<code>java.lang.Runtime</code>
WHEN NOT TO READ	<ul style="list-style-type: none"> <li>• If the application does not deal with system command calls</li> </ul>

### 5.1 Introduction

The domain of code injection vulnerabilities usually belongs to script languages like Perl or PHP. In this case, a successful code injection attack on a web application (based on script languages) leads to the execution of the attacker's script code on the server-side. This also includes commands which allow the manipulation of the host system and the web server.

In the case of Java, in general this kind of attack is of minor importance, as additional Java code usually cannot be inserted into a running application. However, Java allows calling commands provided by the host system. Java applications usually avoid such procedures because Java's native platform independence could be lost for the application. However, in some cases it might be necessary to call system commands, e.g., to access a mail transfer agent on the web server to enable the application to send e-mails.

#### 5.1.1 Variants

In the following, we give a typical example for system command injection via Java's `Runtime` class.

##### 5.1.1.1 Injection of system commands via Runtime

We return to the example addressed in the introduction: a web application has to send e-mails to its registered users. A practical example of this case is the implementation of a "mail me my password" function in an application. There are many ways to implement such functionality. We only provide one bad example.

Please note that we do not discuss general security aspects of such functions. However, those functions are obviously very critical.

### Example of bad code

We assume an application consists of a Java Servlet which processes a web form of the application. This form takes a username and an e-mail address, so that the application can verify if the user is registered and can compare the entered value of the e-mail address with the value that was stored when the user registered. If input validation for the address field was completely omitted, several system commands might be invoked by an attacker.

In the following, we assume the Servlet runs on a UNIX system. The system includes a script for sending e-mails which is called `mail.sh`. This script takes three parameters: The e-mail's subject line (-s), content (-c), and recipient (-r). The Servlet could look like this:

```
// ...

String mailaddress = request.getParameter("mail");
String pw = ""; // The password which shall be mailed
// Loading the password from the application's database
// ...

// Building the command for sending e-mails via the script
String cmd = "mail.sh -s \"Password reminder\" ";
cmd = cmd + "-c \"Your password is: " + pw + "\" ";
cmd = cmd + "-r " + mailaddress;

Runtime.getRuntime().exec(cmd);
```

The execution of the command string (cmd) is done by Java's Runtime class. The intention of building the command string inside of the Servlet is to create a system command as follows:

```
mail.sh -s "Password reminder" -c "Your password is: XY"
-r me@myisp.com
```

If this command is executed, the web server creates a corresponding e-mail out of the given parameters and transfers the e-mail to its recipient.

### Sample attack

Since we assume input validation for the e-mail address is not executed, an attacker is able to append command calls to the original call of `mail.sh`. A sample attack URL could look like this:

```
http://www.mywebapp.com/acc?mail=jane@doe.tld; rm -rf /var/log
```

In this case the Servlet would create the following system command:

```
mail.sh -s "Password reminder" -c "Your password is: XY"
-r jane@doe.tld;
rm -rf /var/log
```

If the web application possesses system executable access rights, the attack would have the following effect: Instead of performing the intended creation and transfer of

an e-mail only, the system would also perform the removal (rm) of its log files (/var/log).

### 5.1.2 Damage in the case of Non-Compliance

If a Java web application is vulnerable to code injection attacks, because, e.g., it executes system commands via Runtime, the following consequences can be identified:

- Gaining control of the application's server system
- Manipulation of the web server
- Manipulation of the web application via the compromised server system
- Unauthorized access to arbitrary application data
- Moreover, it might be possible to attack other computers which can be accessed via network by the web server. These could be third party computers (e.g., Internet servers) and even computers of the web server's private subnet. The attacker might be able to send injected network commands from the web server to those computers.

## 5.2 Prevention, Countermeasures, Solutions

In addition to the dangerous implications of code injection vulnerabilities described above, much detailed information about other potential threats is available. The following material should be useful for an application developer with security concerns, also if platforms other than Java are used. We recommend material provided by the "Open Web Application Security Project" (OWASP [1]) and the "Web Application Security Consortium" (WASC [2]). Especially the OWASP "Top Ten" [3] and the "Web Security Threat Classification" [4] provide more material on code injection vulnerabilities in general.

### 5.2.1 Description

In the following, we give some general rules to protect web applications against code injection vulnerabilities by using input validation.

### 5.2.2 Platform Limitations or Extensions

Since the usage of system commands inside of Java code is performed in individual cases, the Java platform provides no built-in solution for the prevention of code injection vulnerabilities.

### 5.2.3 Solution Variants

Overview:

Solution	Platform	Libraries
<i>Basic rules have to be considered</i>	Java in general (all versions)	n/a; individual procedures are necessary

### 5.2.3.1 Rules

The basic rule to protect applications against code injection vulnerabilities is to avoid the call of system commands from the application side, unless it is absolutely necessary. If system commands have to be called by a web application, input validation has to be performed according to the individual circumstances of the application context. For instance, for the case of the “mail me my password” function described in section 5.1.1.1, the following procedure would be adequate:

- 1) When users are registered, the specified e-mail address is validated. This implies that the input given as e-mail address is checked to see whether it is a valid e-mail address format and contains only allowed characters. The e-mail address is then stored in the application’s database for the corresponding user’s username. (Additionally, it is possible to verify the e-mail address by sending an e-mail to it, and only to accept the registration if the user replies to the application’s e-mail.)
- 2) If the “mail me my password” function is requested by users, the application reads the corresponding e-mail address for the specified user out of its database. Since this e-mail address is valid, it is secure to pass it on to the mailing script used, via Runtime (compare with section 5.1.1.1).

In general, the following rules for input validation have to be considered when system commands within a Java application are used:

- Developers have to make sure that no more and no commands other than the intended ones are executed by the application. This implies, for instance, that it must be impossible to append additional commands to the intended one. For this purpose, it is necessary to be aware of the application’s host system’s attributes (e.g., in the case of UNIX systems, different commands might be combined by ; or pipe character/|). If the injection of an additional command could be detected by the input validation, the execution of the whole request should be refused.
- Developers have to make sure that no additional command parameters or options are added to the command except the intended ones. For this purpose, the structure of the corresponding command call must be known when implementing the input validation, like the number of parameters used.
- All inputs given to a system command as parameters have to be validated to see whether these are in a valid format for the command. For instance, this could imply checking for a valid e-mail address, a valid number range, or allowed characters (compare with section 2.2.3.1).

#### 5.2.3.1.1 To be avoided

Developers should avoid using system commands of the web application’s host unless they absolutely have to. This applies to the usage of Java’s `Runtime` class or similar system classes. If applications really need to execute system commands, the rules for input validation described above have to be considered.

### 5.2.3.1.2 More information

Please see the References for more details about code injection vulnerabilities.

## 5.3 Test Procedure

Please note that the section Test Procedures in this document gives an introduction to the usage of the test procedures described.

### 5.3.1 Outline

The method `java.lang.Runtime.exec(String command)` allows the execution of system commands. Please note that this method has more signatures which can be found in [5]. The command is not dangerous if the command variable is read from a constant or is hard-coded. However, if this value is filled from a user input, or from a non-trustworthy environment, an attacker might be able to execute arbitrary system commands, such as `rm -rf /*` (compare with 5.1).

Test Attributes	
Platform	J2EE/Java
Operating Systems	All, no restrictions
Type of test	Code Review
When should be tested	From initial to final implementation phase
Tool support	Assisting usage of source code scanners and static analysis tools is possible for more sophisticated courses of action

### 5.3.2 Test Execution

Any occurrences of the string `Runtime.exec(...)` have to be found. If the parameter naming the command to be executed is not hard-coded, or read from a constant, a potentially vulnerable piece of code has been located.

#### 5.3.2.1 Special Cases in Practice

Please note that it might be possible that the input variable is filtered manually. For instance, it is possible that the developer is testing for a list of allowed commands (compare with 5.2.3.1). It might be also possible that, in individual circumstances, application developers are bound to use specific or proprietary filter methods. Thus, finding all code lines containing system calls as described above can be done automatically, e.g., by the use of an editor's search function, etc., or more sophisticated tools. However, when a special kind of trustworthy filter method is being used for an application, the search for potential command injection vulnerabilities possibly has to be done manually in consideration of these methods.

Please note that we do not assess the security and trustworthiness of such individual filter methods, in this test procedure. However, please also note that such filters are very critical for the security of an application. Thus, the filters to be used should also be tested in another test procedure.

### 5.3.3 Interpretation of Test Results

The interpretation of the test results, and thus the assessment of the potentially vulnerable code pieces found, is very individual. For instance, the use of Java's `Runtime.exec` method might be not allowed at all for certain application developers. This would imply that every piece of code which uses `Runtime.exec` has to be reengineered.

In all cases, we regard every piece of code as insecure – and thus as “to be reengineered” – which uses the `Runtime.exec` method in the following way: the command string is not hard-coded and not validated against a list of allowed commands. The latter might be realized by proprietary filter methods.

### 5.4 References

- [1] The Open Web Application Security Project (OWASP), <http://www.owasp.org>
- [2] The Web Application Security Consortium (WASC), <http://www.webappsec.org/>
- [3] The OWASP Top Ten, <http://www.owasp.org/documentation/topten.html>
- [4] The Web Security Threat Classification, <http://www.webappsec.org/projects/threat/>
- [5] <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Runtime.html>

## 6 Information Disclosure

Brief description of related security attacks	<ul style="list-style-type: none"> <li>• Systematic elicitation of application error messages</li> </ul>
This topic is related to	<ul style="list-style-type: none"> <li>• Information handling</li> <li>• Error message handling</li> </ul>
Degree of severity	<b>MIDDLE</b>
Consequential and potential damage	<ul style="list-style-type: none"> <li>• Emission of sensitive system and application information</li> </ul>
Affected packages (among others)	<code>java.lang.Exception</code>
WHEN NOT TO READ	n/a

### 6.1 Introduction

In general, the term “information disclosure” describes the emission of data or information which is not intended for public distribution. This includes also, e.g., the inadvertent public storage of sensitive data, like private documents or password files. We omit these topics here, because they are not relevant for the area of secure programming of Java web applications and thus not relevant for the focus of this document. However, application developers should be aware of the range of implications of the term “information disclosure”.

In the programming context “information disclosure” usually means the emission of internal application structures in the form of internal error messages. These might include sensitive information, like libraries used, database structures, or other application characteristics which could be of interest to an attacker.

#### 6.1.1 Variants

In the following, we give an example for the emission of internal error messages inside of a Java Servlet.

##### 6.1.1.1 Emission of internal error messages

We assume a Java web application which produces a runtime error (`NullPointerException`). This is an internal error which usually would create an error log entry on the application’s host system. The following code example inserts the corresponding Java error message into the Servlet’s response text.

### Example of bad code

Please note that the following code is only a simple example of an information disclosure. It demonstrates how internal errors might become available to the public.

```
String output = "<html>";
PrintWriter out = response.getWriter();

try{
    // Servlet operations
    output = output + ("<h1>...</h1>");
    if (request.getParameter("param") != null){
        String temp = request.getParameter("noparam"); // 1
        boolean test = temp.equals("anything"); // 2
    }
    // ...
}
catch (Exception ex){
    output = output + ex.toString(); // 3
}
finally{
    output = output + ("</html>");
    out.println(output);
}
```

We assume this code is part of a Servlet which creates an HTML output. The corresponding HTML page's content is appended to the character string `output` within the whole Servlet. The actual output of the HTML page occurs inside of the `finally` statement, where the `output` string is written into the application's response.

In line 2), the Servlet produces an error, a Java exception, because of a null pointer. This is caused by line 1) where the Servlet assigns the content of a non-existing request parameter to the string `temp`. Thus, in line 3), the application inserts the exception message directly into its HTML response. Please note that also in the case of an exception, Java executes the `finally` statement.

### Sample attack

The following URL only triggers the error described above. We want to highlight that this is not an attack. However, in other circumstances an attacker might be able to gain much more information about an application's characteristics by causing errors systematically.

```
http://www.mywebapp.com/shopping?param=XYZ
```

## 6.1.2 Damage in the case of Non-Compliance

In the introduction we distinguished between two types of "information disclosure" in general:

- 1) The emission of sensitive data in the form of documents or files which have unintentionally become available to the public.

- 2) The emission of sensitive internal application characteristics, like libraries used or database structures.

The damage in the first case depends on the individual importance of the documents. In the second case, direct damage cannot be identified. However, the information which might be gained by an attacker could be useful for further attacks, like cross site scripting or SQL code injection.

There is an overlap between cases 1 and 2, if the files mentioned in case 1 are source code files of the web application. The emission of an application's source code can be interpreted as a disclosure of internal application structures. However, if, for instance, Open Source software is used for the web application, this consideration is irrelevant.

## 6.2 Prevention, Countermeasures, Solutions

In addition to the dangerous implications of information disclosure vulnerabilities described above, much information about other threats in the context of web application security is available. The following material should be useful for application developers with security concerns. We recommend material provided by the "Open Web Application Security Project" (OWASP [1]) and the "Web Application Security Consortium" (WASC [2]). Especially the OWASP "Top Ten" [3] and the "Web Security Threat Classification" [4] provide a minimum standard for web application security. [4] includes a documentation of most of the security threats.

### 6.2.1 Description

In the following, we give some general rules to protect web applications against information disclosure vulnerabilities.

### 6.2.2 Platform Limitations or Extensions

Since information disclosure flaws depend on individual circumstances, the Java platform provides no built-in solution for the prevention of information disclosure vulnerabilities.

### 6.2.3 Solution Variants

Overview:

Solution	Platform	Libraries
<i>Basic rules have to be considered</i>	Java in general (all versions)	n/a; individual procedures are necessary

#### 6.2.3.1 Rules

The rules listed in the following derive from the aspects which have to be avoided in the context of information disclosure. We do not list aspects like the avoidance of publishing sensitive documents or application source code (if applicable), as this is obvious.

In the case of the emission of internal application characteristics, it has to be considered which application components are able to produce information or error messages of interest to an attacker. These might be, for instance:

- The application's platform itself (e.g., Java/J2EE engine)
- Components connected to the application, like database systems  
These components might create messages via:
  - Interfaces which are used to connect a component to the application
  - Their output stream which is provided by the application for the public (e.g., database records)
- The application's runtime environment and host system

The following data and application characteristics, which might become available to attackers via (error) messages, should be suppressed as far as possible:

- The application's host system's configuration, like Java engine type, operating system, web server, database system etc. as well as corresponding configurations and versions
- External libraries used by the application
- If a database is present, the application database's structure, like table definitions, field names etc.

This implies any error message and information which is not of interest to a user must not be addressed to application users (especially information listed above). For instance, the content of the Java runtime stack, which is usually printed in system logs or on the system console when a Java application crashes, has to be kept internally. This is a normal procedure and usually Java's default setting.

In general, a good user interface does not provide internal system errors for users, e.g., like raw errors of a database system. An application catches these errors and provides corresponding help messages to its users through the user interface. Thus, for instance, messages of interest to users are wrongly filled in form fields, like wrong e-mail addresses, etc. The user even should be informed about the non-availability of an application by the application's host system without providing raw system error messages.

#### 6.2.3.1.1 More information

n/a

### 6.3 Test Procedure

Please note that the section Test Procedures in this document gives an introduction to the usage of the test procedures described.

#### 6.3.1 Outline

When a Java exception is thrown, the corresponding application can supply the stack trace and an error message related to the exception. The class `Exception` provides the following methods to output this information:

- `getCause`
- `getMessage`
- `getStackTrace`

- `printStackTrace`
- `toString`

The output of these methods allows a developer to find the reason for the exception. If an attacker can access this information, s/he might use it to exploit a program. Thus, it is extremely important for an application's security that this kind of information related to exceptions should never be written to the application's HTML response.

Test Attributes	
Platform	J2EE/Java
Operating Systems	All, no restrictions
Type of test	Code Review
When should be tested	From initial to final implementation phase
Tool support	Assisting usage of source code scanners and static analysis tools is possible for more sophisticated courses of action

### 6.3.2 Test Execution

Since Java exceptions are only processed in `catch` blocks, all relevant pieces of code for this test procedure are to be found in `catch` blocks. Please note that all kinds of exceptions are relevant. This means that not only occurrences of the class `Exception` but of all of its subclasses, such as `IOException`, are relevant to the search for code pieces.

The next step is to check the exception variable's name from the `catch` block's signature. For instance, a `catch` block might start like this: `catch (Exception e)`. Thus, all occurrences of the variable `e` inside of the `catch` block have to be examined.

First, it is necessary to check whether one of the output methods of the following classes is called inside of the `catch` block:

- `javax.servlet.ServletOutputStream` (e.g., `print(String s)`)
- `java.io.PrintWriter` (e.g., `print(String s)`)

Inside of a Servlet/JSP, corresponding class instances are obtained by the following methods of the class `javax.servlet.ServletResponse`:

- `getOutputStream()`
- `getWriter()`

Second, it is necessary to check whether one of these output methods, such as `print(String s)`, takes a parameter which includes information retrieved by calling one of the following methods of the exception object `e`:

- `getCause()`
- `getMessage()`
- `getStackTrace()`

- `printStackTrace(...)`
- `toString()`

### 6.3.2.1 Special Cases in Practice

n/a

### 6.3.3 Interpretation of Test Results

The two steps given in section 6.3.2 lead to the identification of an information disclosure vulnerability. If information gained from the methods listed above of an exception object is written into a Servlet's/JSP's HTML response, we suggest reengineering the corresponding piece of code. For instance, it would be more appropriate to create user-friendly error messages inside of the web application (compare with section 6.2.3).

## 6.4 References

- [1] The Open Web Application Security Project (OWASP), <http://www.owasp.org>
- [2] The Web Application Security Consortium (WASC), <http://www.webappsec.org/>
- [3] The OWASP Top Ten, <http://www.owasp.org/documentation/topten.html>
- [4] The Web Security Threat Classification, <http://www.webappsec.org/projects/threat/>

## 7 Unreleased Resources

Brief description of related security attacks	<ul style="list-style-type: none"> <li>• Flooding an application with requests to overload the application and its server system</li> </ul>
This topic is related to	<ul style="list-style-type: none"> <li>• Releasing application resources</li> <li>• Garbage collection</li> </ul>
Degree of severity	<b>LOW</b>
Consequential and potential damage	<ul style="list-style-type: none"> <li>• Denial-of-service</li> </ul>
Affected packages (among others)	all
WHEN NOT TO READ	<ul style="list-style-type: none"> <li>• If the application does not deal with external resources</li> <li>• If the application does not deal with objects which have the ability to become released manually</li> </ul>

### 7.1 Introduction

The security relevance of this category is related to denial-of-service attacks only. However, we want to highlight that unreleased resources are not the only reason for possible denial-of-service attacks on web applications.

In the case of unreleased resources, denial-of-service vulnerabilities occur when system resources are not available but necessary for further application procedures. If an attacker is able to discover the vulnerability, s/he might be able to execute a systematic attack by causing several application faults, which finally result in a denial-of-service.

In general, affected application resources are system resources, like main or hard disk memory, as well as external components, like libraries or databases. Java provides two possibilities for developers to release these resources: a) by (automatic) garbage collection of unused objects and b) by explicit unloading of objects (if possible). In the case of insufficient main memory, the occurrence of denial-of-service vulnerabilities is obvious. In the case of unreleased external components, denial-of-service vulnerabilities might occur, for instance, if the number of possible database connections is limited or a data file is opened in an exclusive mode. Thus, further access to these resources by the application generates runtime errors which might lead to a crash of the whole application.

#### 7.1.1 Variants

In the following, we give an example for an unreleased database resource inside of a Java Servlet.

##### 7.1.1.1 Abandonment of releasing resources

The code example below contains a database connection via JDBC. Thus, three objects are created:

- `db`, a `JDBC Connection` for providing a database connection
- `ps`, a `PreparedStatement` for executing an SQL query

- `rs`, a `ResultSet` for browsing through the fetched data

### Example of bad code

We assume the following code is inside of a web application based on Java Servlets:

```
// Preparation of database connection data

Connection db = null;
PreparedStatement ps = null;
ResultSet rs = null;

try{

    // Setting up the database connection
    Class.forName(conDriver);
    db = DriverManager.getConnection(conURL,
                                    conName, conPass);

    // Setting up the SQL query
    ps = db.prepareStatement("SELECT * FROM testtable1
                              WHERE ...;",
                             ResultSet.TYPE_SCROLL_SENSITIVE,
                             ResultSet.CONCUR_READ_ONLY);

    // Getting the selected database records
    rs = ps.executeQuery();

// Database operations
// ...
// End of code

}
catch (Exception ex){System.out.println(ex.toString());}
```

Every time the application server gets a URL call of the Servlet, it creates a new instance of the Servlet. Thus, if the Servlet manages database connections in the way shown above, every Servlet instance creates a new database connection. Please note that the code example contains no procedure for an explicit release of the objects related to the database connection (compare with section 7.2.3.1.4). In this case, the developer trusts Java's garbage collector which should release the created database objects on time.

However, in practice, two aspects might cause threats to the application's security:

- The garbage collection is too busy for releasing resources on time
- The Servlet crashes inside the try-clause for some reason

Hence, several scenarios for entire application crashes can be identified when the application's resources are not suitably released. These scenarios might be a result of the two aspects above:

- A resource is opened exclusively and thus becomes blocked, like a file handler
- A resource has a limited amount of access, like connections to a database

- Memory consumption might become critical

For the code example, the latter two scenarios are possible. This causes denial-of-service vulnerabilities. An attacker could exploit them, e.g., by systematic creation of Servlet calls for the purpose of overloading the application's amount of possible database connections.

### 7.1.2 Damage in the case of Non-Compliance

If an application includes unreleased resource vulnerabilities, denial-of-service attacks become possible. If resources, such as database systems, are shared among different applications, even applications other than the vulnerable one might be attacked.

Moreover, if an application uses the security functions of external libraries, the application developer has to consider an application procedure for the detection of the non-availability of the corresponding libraries and security functions. In case this is neglected, an attacker might be able to bypass security mechanisms of the application by launching denial-of-service attacks.

## 7.2 Prevention, Countermeasures, Solutions

In addition to the dangerous implications of vulnerabilities related to unreleased resources described above, more information about threats related to denial-of-service attacks on web applications is available. The following material should be useful for application developers with security concerns. We recommend material provided by the "Open Web Application Security Project" (OWASP [1]) and the "Web Application Security Consortium" (WASC [2]). Especially the OWASP "Top Ten" [3] and the "Web Security Threat Classification" [4] provide a minimum standard for web application security. [4] includes a documentation of most of the security threats.

### 7.2.1 Description

In the following, we give some general rules to protect web applications against vulnerabilities related to unreleased resources.

### 7.2.2 Platform Limitations or Extensions

Since unreleased resource vulnerabilities depend on the individual circumstances, described above, the Java platform provides no built-in solutions.

### 7.2.3 Solution Variants

Overview:

Solution	Platform	Libraries
<i>Basic rules have to be considered</i>	Java in general (all versions)	n/a; individual procedures are necessary

### 7.2.3.1 Rules

Besides trusting Java's garbage collector, a developer should consider the following general programming rules when dealing with external resources:

- An application requires procedures for the case that an external resource is not available to ensure a safe application workflow. Then the application does not crash when resources become unavailable. Moreover, in case an application uses security functions within an external resource, the developer has to ensure that the application enters a secure state when the corresponding resource becomes unavailable. This also applies if an application requires an external resource inside of its own security function. For instance, we assume an application uses access control functions from a third party library for certain application content. For some reason, this library is not available during application runtime. In this case, it has to be ensured that the application does not run without performing access control on its content. Thus, either the application runs an "emergency plan" in this situation, or it forbids access to the complete content.
- Developers have to make sure that, in all application flows, no resource which could be needed in further processes (e.g., file handlers) becomes blocked.
- In case of limited resources, like database connections, are being used, it has to be guaranteed that sufficient resources are available during application deployment. Especially the application's maximum load in practice has to be considered.
- External resources should be explicitly released as soon as possible in the application context. Explicit releasing, (e.g., by using the Java object's `close()` method, if available) ensures that the garbage collector releases a resource on time. The automatic garbage collection routine might otherwise release a resource too late. For instance, in the case of a Servlet which uses a database connection as described in section 7.1.1.1, a new database connection is created for each HTTP request. If the garbage collector is too busy, the database system's maximum number of allowed connections might be exceeded.

Releasing resources also has to be guaranteed by the application in case of runtime errors or crashes. For instance, if a Servlet's database connections are not closed correctly, the database system might hold some open connections which are not in use anymore. Again, the database system's maximum number of allowed connections might then be exceeded unintentionally.

In general, releasing a resource inside of a Java application depends on the Java object, which stands for the corresponding resource, itself. A "standard" procedure is to call the `close()` method of the corresponding object, if it provides one.

Moreover, to also guarantee the release of resources even when (runtime) errors occur, it is appropriate to use Java's `finally` statement for releasing resources in general. In section 7.2.3.1.4 we demonstrate the usage of `close()` and `finally` but also highlight some problems of their usage in practice.

### 7.2.3.1.1 Requisites

Technical Requisites	Platform Release	Features/Interfaces to be used
Java resources with the ability to become explicitly released	Java (all)	n/a

### 7.2.3.1.2 To be avoided

External application resources should never be used blindly. The application has to provide procedures in case resources are not available or only available with restrictions. If resources are limited, their availability during the application runtime must be considered.

### 7.2.3.1.3 More information

Please see the Java documentation for more details on Java's garbage collection.

### 7.2.3.1.4 Example of good code

In the following we provide an adjusted code fragment, in contrast to the code example in section 7.1.1.1. The difference is additional code at the end (marked by // NEW):

```
// Preparation of database connection data

Connection db = null;
PreparedStatement ps = null;
ResultSet rs = null;

try{

    Class.forName(conDriver);
    db = DriverManager.getConnection(conURL,
                                    conName, conPass);
    db.setAutoCommit(false);

    ps = db.prepareStatement("SELECT * FROM testtable1
                            WHERE ...;",
                            ResultSet.TYPE_SCROLL_SENSITIVE,
                            ResultSet.CONCUR_UPDATABLE);
    rs = ps.executeQuery();
    rs.beforeFirst();
    rs.next();

// Database operations
// ...
// End of code
```

```
}  
catch (Exception ex){System.out.println(ex.toString());}  
finally{ // NEW  
    try{  
        rs.close();  
        ps.close();  
        db.close();  
    }  
    catch (Exception ex){System.out.println(ex.toString());}  
}
```

The new `finally` statement realizes the release and deallocation of the three database-related objects by their corresponding `close()` method. This is an example of the general procedure described in section 7.2.3.1.

Please note that the `finally` statement is always executed. This applies if an error occurs between the first `try...catch` statement and for the case that no error occurs at all. Thus, in all cases, the database resources are released when the Servlet terminates.

We want to highlight that, in practice, several conceptual problems might also occur when the `finally` statement is used:

- In practice, a `finally` statement could consist of complex code. It might be possible that a runtime error occurs inside of a `finally` statement. If so, the rest of the code inside the statement will not be executed. For instance, in the code example above, the execution of `rs.close()` might trigger an error. This implies that the other two lines of the `finally` statement are not executed. Thus, in fact, another `finally` statement would be necessary for executing the remaining two lines of code. Hence, in practice, a trade-off and an individual solution must be found for implementing the release of an application's resources in an appropriate way.
  - We want to highlight that errors within `finally` statements might also occur because of the usage of a release method itself. For instance, the execution of `rs.close()` might trigger an error because the database is "not ready" for some reason. Again, these cases must be solved in an appropriate way in the application context.
- In many scenarios, resources within applications are defined globally, because this kind of implementation might be very easy and efficient. However, these scenarios make it difficult to decide when a resource is no longer required and thus may be released. For instance, if one resource, like a database connection, is used within a whole Java class or even a whole package, it might be difficult to decide when and where in the application code the resource's release might be executed.

### 7.3 Test Procedure

Please note that the section Test Procedures in this document gives an introduction to the usage of the test procedures described.

### 7.3.1 Outline

Usually, operations on resources in Java are framed in two basic steps:

- the resource requested is opened (e.g., open a file)
- the resource is released (e.g., close a file)

One of the developer's tasks is to minimize the time during which a resource is opened, especially if the resource is requested exclusively. As shown above, unreleased resource vulnerabilities can be used for DoS attacks.

This test ensures that all resources are released after their usage. Therefore, this test checks for method pairs which should be called before and after the use of a resource. Usually, resources should be released in a `finally` block to ensure their closing/release under all circumstances (e.g., also on errors).

Test Attributes	
Platform	J2EE/Java
Operating Systems	All, no restrictions
Type of test	Code Review
When should be tested	From initial to final implementation phase
Tool support	Assisting usage of source code scanners and static analysis tools is possible for more sophisticated courses of action. Additionally, it is possible to use the Open Source tool FindBugs to perform this test procedure automatically (see section 1.3.4 for more details).

### 7.3.2 Test Execution

Please note that we cannot list all relevant resources of the security category “unreleased resources”, here. A corresponding list of resources would be incomplete, since all possible external resources (compare with 7.2.3.1) of an arbitrary application are basically relevant for this test procedure. Thus, every developer has to identify the relevant resources which are being used within their application individually. In the following, we take two important kinds of resources as examples: files and databases. Other resources might be, for instance, network sockets or database transactions.

Usually, the code lines which have to be located for this test procedure depend on the respective kind of resource:

- File access: Each file which is opened has to be closed. Usually files are opened by using the constructor of the class which holds the file, like `new File(...)` or `new RandomAccessFile(...)`. After access, a file is usually released/closed via the method `close()`.
- Database access: In Java, database connections are managed via JDBC. A database connection `con`, for instance, is created via the method `DriverManager.getConnection()`. To close the database connection, the `con` object, which holds the connection, provides the `close()` method.

Corresponding method pairs should be identifiable for all kinds of resources. However, there are no obligations to, for instance, label a method for a resource's release `close()`. Thus, methods for creating and releasing resources have to be identified individually.

### 7.3.2.1 Special Cases in Practice

n/a

### 7.3.3 Interpretation of Test Results

It is necessary to verify whether all resources which are created within an application – this corresponds to the code lines identified in 7.3.2 – are released appropriately. For this purpose, as mentioned above, the method pairs for creating and releasing an individual resource in Java have to be identified. If the code contains missing resource releases, it has to be reengineered.

Please note that a resource's release has to be implemented for every path in the application's data flow. For many cases, the use of a `finally` block is appropriate. However, this is not always possible. Thus, individual release strategies might have to be identified under particular circumstances.

### 7.3.4 Special Tool Support

The Open Source tool FindBugs provides security bug detectors which cover the test procedure given above. Thus, this test procedure can be executed automatically by applying FindBugs to the corresponding application. As mentioned above, there are many kinds of external resources which might be used within Java programs. Please note that this document's category of "unreleased resources" does not correspond to a single bug detector of FindBugs. Instead, FindBugs provides detectors which can be used for detecting two important kinds of unreleased resources solely:

- "Open Stream" (OS)
- "Open database resource" (ODR)

In the following, we list the steps necessary for applying FindBugs to a Java application for detecting unreleased resources:

- The current archive which contains the FindBugs Java program can be downloaded from <http://findbugs.sourceforge.net/downloads.html>.
- The archive has to be unpacked.
- It has to be ensured that an up-to-date Java Runtime Environment (JRE) is installed on the system which is to run FindBugs.
- FindBugs can be started via the batch script `bin\findbugs.bat` (Windows) or `bin/findbugs` (UNIX).
- When FindBugs is running, a new project has to be opened by choosing the menu "File" – "New Project". A new project window will open. Here, FindBugs can be applied to the corresponding Java application by providing the path to the compiled application, the path to the application's source code (not required), and the application's classpath entries. Then, the FindBugs analysis can be started.

- The result screen which follows might also show other bugs, since FindBugs includes many bug detectors for Java. Unreleased resources are marked by the term “OS” or “ODR” at the beginning of a line containing a bug report. In the case of unreleased database resources, the corresponding line will contain an additional message like: “method X may fail to close database resource”.

#### 7.4 References

- [1] The Open Web Application Security Project (OWASP), <http://www.owasp.org>
- [2] The Web Application Security Consortium (WASC), <http://www.webappsec.org/>
- [3] The OWASP Top Ten, <http://www.owasp.org/documentation/topten.html>
- [4] The Web Security Threat Classification, <http://www.webappsec.org/projects/threat/>

## 8 Missing Input Validation

Brief description of related security attacks	<ul style="list-style-type: none"> <li>• Tampering with data for individual application processes (application functionality)</li> </ul>
This topic is related to	<ul style="list-style-type: none"> <li>• Dealing with external / user inputs</li> <li>• Input validation and filtering</li> </ul>
Degree of severity	<b>MIDDLE</b>
Consequential and potential damage	<ul style="list-style-type: none"> <li>• Unintentional and unauthorized setting of application states and data</li> </ul>
Affected packages (among others)	all
WHEN NOT TO READ	n/a

### 8.1 Introduction

This category summarizes all security errors which are related to the logical application workflow. Generally, a vulnerability of this category exists if the developer sets the state of an application and thus influences its workflow, due to wrong assumptions. Since attacks typically take place from the outside via an application's entry point, like the user interface, invalidated external inputs can be identified as the most relevant vulnerability in this category.

#### 8.1.1 Variants

In the following we give one example for a typical security error in this category. We provided a similar example in section 3.1.1.2. The difference between the examples is only the entry point: cookies in section 3.1.1.2 and URLs in the following example.

##### 8.1.1.1 Invalidated state parameters

We assume a Servlet allows access to specific operations or data only to privileged and thus specific application users, e.g., administrators. After the login procedure, the application sets a status parameter, which describes the user's privileges, via URL rewriting on the client-side.

Obviously, this parameter might be easily manipulated when its meaning is discovered. A slightly less obvious manipulation is possible if the parameter is stored inside of a cookie (compare with section 3.1.1.2). However, both cases are related to the same vulnerability and demonstrate false trust in external data. Thus, developers must never regard any external data as a trustworthy part of the application.

##### Example of bad code

A vulnerable Servlet could look like this:

```
// ...

String rawadmin = "";
Boolean isadmin = false;

if(request.getParameter("param") != null){
```

```
    rawadmin = request.getParameter("param");
}

if(rawadmin.equals("1")){
    isadmin = true;
}

// ...
// Code for privileged users only

if(isadmin){
    // ...
}
```

The (internal) variable `isadmin` is used within the application as a status flag (a Boolean value) which tells whether a user has privileged access rights or not. The variable `rawadmin` reads in the corresponding HTTP request parameter's content, which is then used for setting `isadmin`, in case `rawadmin` contains a valid value.

#### Sample attack

We assume the Servlet above performs URL rewriting for setting the status flag of a user without privileged access rights. A resulting URL could look like this:

```
http://www.mywebapp.com/manage?param=0
```

In practice, an application might use additional parameters, e.g., for storing session information or setting page IDs etc. Thus, in practice a status parameter like this might not be discovered easily. Moreover, the parameter could have more complex values than only 0 and 1. However, in our example, an attacker could easily call a URL for gaining privileged access rights as follows:

```
http://www.mywebapp.com/manage?param=1
```

Now, the Servlet would allow the user – the attacker – access to its privileged operations or data. Hence, it is obvious that the Servlet code succumbs to a logic error which is the assumption that the status parameter's content can be trusted and are probably not modified.

### **8.1.2 Damage in the case of Non-Compliance**

Since logic security errors occur in very individual circumstances, their damage can be summarized best in relation to the issues of unauthorized access to and manipulation of application data or individual application procedures.

## **8.2 Prevention, Countermeasures, Solutions**

In addition to the dangerous implications of Missing Input Validation described above, much information about other threats in the context of web application security is available. The following material should be useful for application developers with security concerns. We recommend material provided by the "Open Web Application Security Project" (OWASP [1]) and the "Web Application Security Consortium"

(WASC [2]). Especially the OWASP “Top Ten” [3] and the “Web Security Threat Classification” [4] provide a minimum standard for web application security. [4] includes a documentation of most of the security threats.

### 8.2.1 Description

In the following, we give some general rules to protect web applications against Missing Input Validation.

### 8.2.2 Platform Limitations or Extensions

Since Missing Input Validation are very individual programming flaws, the Java platform provides no general built-in solutions.

### 8.2.3 Solution Variants

Overview:

Solution	Platform	Libraries
<i>Basic rules have to be considered</i>	Java in general (all versions)	n/a; individual procedures are necessary

#### 8.2.3.1 Rules

Vulnerabilities of this category emerge when developers set application states due to invalidated external inputs. In certain contexts, vulnerabilities caused by invalidated inputs have specific characteristics, so that they can be classified into sub-categories such as cross site scripting or Resource Tampering vulnerabilities.

Since this category summarizes all security errors which are related to the logical application workflow, the following rules for protecting applications against corresponding vulnerabilities are very generic. They must be applied to those individual application procedures which rely on external inputs. Thus, the main task is to implement a complete input validation for the corresponding application context.

The following basic aspects have to be considered when implementing input validation and corresponding input filtering procedures:

- Constrain input and input fields (e.g., database fields, String variables etc.)
- All input has to be validated
  - Define a codepage (e.g., character set ISO-8859-1) to clearly decide which character encoding is used
  - Filter special characters (e.g., %) and meta-characters (e.g., tag commands in the context of HTML, like <) depending on the application context
  - Restrict variables to those characters that are explicitly allowed (whitelist)
  - Validation in general involves the following aspects:
    - Field length (e.g., character strings)
    - Data types
    - Range (e.g., numbers in general, date fields, etc.)

To protect a web application against Missing Input Validation, the following basic rules have to be followed:

- Developers should minimize the data which has to be transferred to the client and retransferred to the web application for any process.
- All data that could be stored on the server-side, like status parameters of users or other user data (compare with section 8.1.1.1 or 3.1.1.2), should be stored on the server-side by the application. For further application processing, connections between the client/user and the data stored on the server-side can be realized by temporary identification numbers. These IDs are comparable to Java's session IDs. Thus, the developer always has to consider whether it would be possible to use Java's session management to connect certain user data with the web application (compare with section 3.2.3.1.2).
- When an application process relies on external data, all external data transferred to the application has to be validated. For this purpose, all entry points of the application have to be considered. The input validation must guarantee that the corresponding application process only receives data in the valid format which it expects. Moreover, input validation has to be performed on any external data. This includes data which is generated by the application and should be automatically retransferred by a client, e.g., status parameters set via URL rewriting (compare with section 8.1.1.1).

Entry points of a web application depend on the application context. For instance, if for any reason an application processes raw IP packets, these packets and their fields and headers are an entry point. It has to be considered that an attacker might send manipulated packets to the application. In general, a web application is able to receive external data via several entry points. The following list of entry points includes overlaps. This is done on purpose, because an entry point can be understood in a variety of ways, depending on the degree of abstraction which is used within the application to process data coming from an entry point. For instance, fields of a web form are part of the application's user interface, which is obviously an entry point. However, the application receives the fields' values as HTTP request parameters. The corresponding HTTP request is triggered by the HTML web form. Entry points of an application which might deliver data which the application processes are:

- HTTP requests, including
  - HTTP headers (containing, e.g., the client's web browser identification)
  - URLs
  - Cookies
  - POST parameters
  - GET parameters (part of the URL)
- URLs
  - URL itself (compare to path traversal attacks, see section 4)
  - URL parameters
- Cookies
- User interface, including
  - Form fields
  - Dynamically generated and static web links (URLs)

- Files derived from file uploads (e.g., via HTTP PUT)
- Other protocols supported by the application (e.g., SSL, FTP, etc.)

These entry points are the external entry points of a web application and it is usually adequate to consider only these. However, in practice, it might be also relevant to validate data coming from other entry points, like third party libraries used inside of an application, other internal services of the application's host system, or utilized web services.

When data arrives at the application through an entry point, it has to be validated according to the basic rules described above. It might be practical to centralize input validation procedures for an application. However, external data must be validated before the application runs any further processing which depends on this data.

#### 8.2.3.1.1 To be avoided

In the case of logical security errors related to user inputs, the application developer has to be aware of the fact that not only well-defined vulnerabilities, like cross site scripting or SQL injection, exist. In certain application contexts, other security errors related to user inputs might be created by developers due to false assumptions (compare with section 8.1.1.1 or 3.1.1.2, for instance).

Thus, an application developer has to consider that *any* kind of data which comes from the application's clients has to be validated. Even data which is generated by the application, then transferred to the client, and then retransferred to the application might be manipulated and has to be validated. This also applies to data which is generated automatically and not intended to be changed by the client, like web page IDs, hidden form fields, or even session IDs.

#### 8.2.3.1.2 More information

n/a

### 8.3 Test Procedure

Please note that the section Test Procedures in this document gives an introduction to the usage of the test procedures described.

#### 8.3.1 Outline

As mentioned above, a vulnerability of this category exists if the developer sets the state of an application on the basis of wrong assumptions. Since attacks typically take place from the outside via an application's entry point, invalidated external inputs can be identified as the most relevant issues for the corresponding test procedure.

Test Attributes	
Platform	J2EE/Java
Operating Systems	All, no restrictions
Type of test	Code Review

---

Test Attributes	
When should be tested	From initial to final implementation phase
Tool support	n/a

### 8.3.2 Test Execution

Since this security category is very generic, its test procedure is generic, too. We do not provide a complete list of all Java methods which provide external data, because this is not possible. Application developers have to decide individually which methods might be relevant for their application. General entry points have been listed in section 8.2.3.1. Any occurrences of invalidated external input which have an influence on the application's dataflow and on application states might be a potential vulnerability.

#### 8.3.2.1 Special Cases in Practice

n/a

### 8.3.3 Interpretation of Test Results

The interpretation of the test results and thus the assessment of the potentially vulnerable code pieces which have been located are highly individual (compare with 8.2.3.1).

## 8.4 References

- [1] The Open Web Application Security Project (OWASP), <http://www.owasp.org>
- [2] The Web Application Security Consortium (WASC), <http://www.webappsec.org/>
- [3] The OWASP Top Ten, <http://www.owasp.org/documentation/topten.html>
- [4] The Web Security Threat Classification,