THE BEST-RUN BUSINESSES RUN SAP

Secologic

Java Web Application Security

Best Practice Guide

Document Version 1.0 – April 2006

These materials are provided by SAP AG for informational purposes, without representation or warranty of any kind. These 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

Contents

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 all different existing topics in the area of IT security we have also constructed a **Classification table**. This table provides a clear overview for the assignment of well-known topics of 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 in the entire document in the given meaning.

Every section has the following structure:

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	References

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	**SEE BELOW**
Consequential and potential damage	Enumerates well-known damage possibilities 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 gives 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:

HIGH

MIDDLE

LOW

The summary table is followed by 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 describe the requirements for and necessity of preventing the corresponding security flaw in applications (Prevention). The next subsection gives a general introduction to solutions and good source code. This is followed by the detailed description of concrete implementation methods and their requirements, including examples as far as possible (Solutions and Countermeasures).

Platform solutions described in this document

	Solution offered	
Category	**Java**	**J2EE**
Code Injection	O	O
Cookie Security	O	O
Cross Site Scripting	O	O
Information Disclosure	O	O
Logic Errors	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 corres-ponding 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
Code Injection (COD)	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
Cookie Security (COO)	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
Cross Site Scripting (XSS)	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
Directory	*see Path Traversal*	

Category	Definition for the area of IT security	Related to
Browsing		
Directory Traversal	*see Path Traversal*	
Flow Injection (FLO)	This category is a creation of our own. It is a special case of logic errors 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.	LOG
Information Disclosure (INF)	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 only 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 *output encoding*. If input validation, filtering, and output encoding are missing or incomplete, this can enable a variety of attacks.	COD COO RES SQL XSS
Logic Errors (LOG)	All programming errors, but also errors in system design or specification, which cannot be classified in another security category are called logic errors. Thus, these errors are not typical programming errors. Moreover, it is usually not possible to test for resulting security flaws.	FLO
Path Browsing	*see Path Traversal*	
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 Resource Injection**.	COD INP RES
Resource Injection (RES)	We define resource injection 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
SQL Code Injection (SQL)	Results of successful attacks of this category are the execution of arbitrary SQL statements and commands on the application's database backend(s).	INP
Unreleased Resources (UNR)	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 this document (Version 1.0)
- *Expression* (green italics): Chosen sections for the next version of this document
- 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	• Injection of arbitrary SQL commands • Manipulation of existing SQL queries
This topic is related to	• Dealing with HTTP request parameters • Input filtering, validation, and encoding • Secure access to databases via JDBC
Degree of severity	**HIGH**
Consequential and potential damage	• Complete and possibly unperceived access to the whole application database • Manipulation of data, including deletion of tables and databases • Access to the database system's host / server system (depending on the functionality provided by the database system's SQL)
Affected packages (among others)	`java.sql.*`
WHEN NOT TO READ	• If the application does not deal with database access • If the application does not deal with user data

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 assure 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 regarding 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 are due to the fact that 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 initialised, 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 to append 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 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.

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
Prepared Statements	Java in general (all versions)	`java.sql`
Stored Procedures	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 and so on:

```
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 compared to 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 developed stored procedures. 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 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/

2 Cross Site Scripting (XSS)

Brief description of related security attacks	• Injection and execution of arbitrary script commands on web application clients (e.g., JavaScript) • Redirecting web application clients to arbitrary locations
This topic is related to	• Dealing with HTTP (GET/POST) request parameters • Output encoding and input validation/filtering • Storing request data in the application's database
Degree of severity	**HIGH**
Consequential and potential damage	• Redirecting of clients for further attacks (e.g., exploiting web browser vulnerabilities) • Identity theft and impersonation • Session hijacking
Affected packages (among others)	`javax.servlet.*, javax.servlet.http.*`
WHEN NOT TO READ	• If the application does not deal with HTTP requests

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 trigged 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>`
- `<img src="filename" onclick="alert();">`
- `<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.[1]

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 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.

[1] We will publish a best practice document for input validation within the "Secologic" project.

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**
Encoding	Java in general (all versions)	n/a; individual procedures are necessary
Encoding	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[2]
  - Filter special characters and meta-characters (e.g., tag commands in HTML, like <)
  - Restrict variables to those characters that are explicitly allowed (whitelist)
  - 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

[2] In practice, it has to be remembered that web browsers might be able to redefine the character set of HTTP requests.

- 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 `&quot;&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 2.3 *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 htmlescape(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 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

[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	• Stealing of cookies • Poisoning of cookies • Code injection via cookies • XSS attacks via cookies
This topic is related to	• Dealing with cookies (within HTTP requests) • Session management • Input filtering, validation, and encoding of cookie data
Degree of severity	**HIGH**
Consequential and potential damage	• Identity theft and impersonation • Session hijacking • Unauthorized access to the application and its data • Deception of application users
Affected packages (among others)	`javax.servlet.*, javax.servlet.http.*`
WHEN NOT TO READ	• If the application does not deal with cookies and thus neither set nor read cookies

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 logic errors

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 logic errors 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 logic errors (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
Basic rules have to be considered	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 `javax.servlet.http.HttpSession`, class `javax.servlet.http.Cookie`

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 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 Injection

Brief description of related security attacks	• Unauthorized access to application resources, like files or folders
This topic is related to	• Dealing with HTTP requests • Input filtering and validation • Web application deployment
Degree of severity	**HIGH**
Consequential and potential damage	• Information disclosure • Unauthorized modification of application data
Affected packages (among others)	`javax.servlet.*`
WHEN NOT TO READ	• If the application does not deal with HTTP requests at all

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 injection 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-characters 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 injection – 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.

4.1.1.1.1 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.

<u>Sample attack: Relative directories</u>

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 injection 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 injection 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 injection 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**
Using Resource Handlers	Web servers	n/a

4.2.3.1 Rules

We want to give some general recommendations to prevent resource injection 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 injection 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 injection 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 injection 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 4.3 References for more details about the possibilities of configuring resource handlers of web servers.

4.3 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