Software Security (SSI)

8. Java Web Application Security – Part 1

Prof. Dr. Marc Rennhard
Institut für angewandte Informationstechnologie InIT
ZHAW School of Engineering
rema@zhaw.ch

There are several valuable resources for the content of this chapter:

- **Java Platform, Enterprise Edition 7 API Specification**: *http://download.oracle.com/javaee/7/api*
- **Java 7 EE documentation**: A very thorough introduction to Java EE 7 can be found here: *http://download.oracle.com/javaee/7/tutorial/doc/*
- **Java Servlet Specification**: *http://www.oracle.com/technetwork/java/javaee/servlet/index.html*

Content

- Introduction to Java web applications
- The Marketplace application – which serves as an example throughout the entire chapter
- Various declarative and programmatic security mechanisms offered by Java SE and EE to secure web applications
- Input validation with OWASP ESAPI
- Cross-Site Request Forgery protection by implementing our own mechanism

Goals

- You know typical security problems that can arise when developing Java web applications
- You understand the difference between declarative and programmatic security in web applications
- You understand the different technologies and methods that can be used to secure Java web applications and can apply them appropriately to secure your own applications
- You know the possibilities of OWASP ESAPI with respect to input validation and can apply them to secure your own applications
- You understand how CSRF attacks can be prevented and can implement a corresponding mechanism

Introduction

Java Web Applications

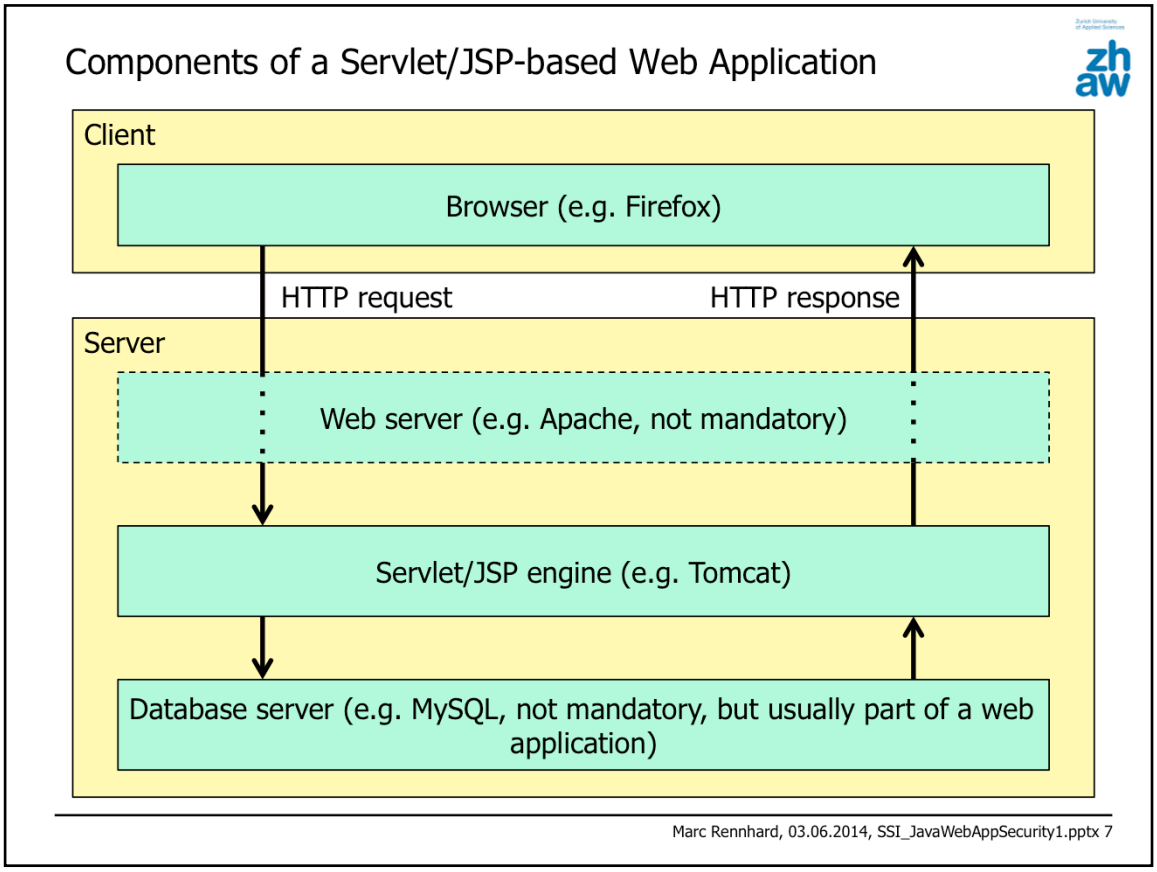
- With Java Web Applications, we mean web applications developed with Java EE (Java Enterprise Edition) technology
  - Java EE is based on the Java Standard Edition (Java SE)
- First release of Java EE in 1999, current version 7 (Java EE 7)
  - Just like with Java SE, new versions are backward compatible
  - As web applications often have long lifetimes, older versions are still frequently used
- Among the basic technologies of Java EE to develop web applications are Servlets and JavaServer Pages (JSP)
  - They also provide the basis to implement secure web applications → we focus on these technologies here
- To run web applications based on servlets/JSPs, a servlet/JSP engine (or container) is required
  - We use Tomcat here, which is the most popular servlet/JSP engine

Enterprise Java Beans

We do not discuss technologies such as Enterprise JavaBeans (EJB) and their security aspects here, as EJBs are not required to develop secure and powerful web applications

Why Using Servlets and JSPs?

- Most of you are at least somewhat familiar with these technologies
  - Using more advanced frameworks/components (JSF, Spring, Struts...) would require lots of time until we could use them
- They are very well suited to truly learn how to solve security issues in web applications
  - Modern frameworks often "solve" some security issues for you out of the box, but developers then usually don't really know what happens
  - With Servlets/JSPs, you have to solve a lot on your own, which will help you to truly understand how security issues can be solved
- Once you have mastered solving the security issues in a servlet/JSP-based web application, you should be able to transfer what you've learned to other technologies and programming languages
  - And it helps you to understand security features that are possibly available there and what their limits are
  - For those interested in JSF: check the appendix



Web Server

The web server is not required in a pure servlet/JSP application. Still, using a web server such as Apache may be reasonable to handle requests to static resources (HTML pages) or to handle request to resources written in other languages (e.g. PHP). In this course, we will only use Tomcat.

Enterprise JavaBeans

As mentioned before, we won't look at EJB here. However, if EJB are used in a Java web application, Tomcat is no longer sufficient and and EJB server or EJB container must be used (e.g. JBoss or Glassfish).

Java EE Versions

- Different Java EE versions also implies different versions of the servlet/ JSP specifications and different Tomcat versions that are necessary
- Java EE 5 includes Servlet 2.5 and JSP 2.1
  - Supported by Tomcat 6 (and later)
- Java EE 6 includes Servlet 3.0 and JSP 2.2
  - Supported by Tomcat 7 (and later)
- Java EE 7 includes Servlet 3.1 and JSP 2.3
  - Supported by Tomcat 8
- Security-wise, only little has changed in recent versions
  - Most of what we will discuss here is supported (at least) since Java EE 5
  - We will point it out when using security features that are only supported since Java EE 6 or 7

Java Web Application Security (1)

In this chapter, we will look at several security-relevant details that you must consider when developing web applications:

- Suppressing detailed error messages
- Data sanitation
- Secure database access
- Authentication
- Access Control
- Secure communication
- Session handling
- Input validation
- Cross-Site Request Forgery prevention

Java Web Application Security (2)

Understanding the concepts and practices in this chapter will help you in various ways:

- Develop secure Java web applications
- Get a better understanding of security aspects that must be considered during web application development in general (also beyond Java)
- Get the basis to assess security features that are offered by web application development frameworks and third party libraries
  - Because you'll know what such a feature should offer to be secure
- Get a better understanding what can go wrong when neglecting security
  - Helps when security testing web applications, as knowing "what developers may have forgotten" helps to identify attack vectors
  - Helps during threat modeling, as you know realistic threats against poorly protected web applications

Java Web Application Security (3)

To build a secure web application, we will combine several different concepts and technologies

- Several of them are provided by Java EE and Java SE itself
  - E.g. authentication, access control, data sanitation, secure DB access
- Some are provided by third party libraries
  - E.g. Input validation
- And in some cases, we implement an own approach
  - E.g. CSRF prevention
- But don't forget: "general aspects" of secure/robust programming are still important as well to get a secure web application
  - This involves handling of "unexpected situations"
    - E.g.: If an attacker removes a parameter from the request, the web application should not crash with a NullPointerException
  - But we won't focus on that in this chapter – although the example application used in this chapter should be quite robust

Declarative and Programmatic Security

- Declarative Security
  - The application's security model is described in a form external to the actual program code
  - With Java EE applications, this is done wither in the deployment descriptor (web.xml) or with annotations (since Java EE 6)
  - Enforced during runtime by the servlet/JSP engine
  - Can easily be configured, but allows only relatively coarse-grained security and is limited to what it offers
- Programmatic Security
  - Security aspects are integrated in the program code
  - More complex, more error-prone, but also more flexibility
- Best practice
  - If possible, use declarative security whenever possible and supplement it with programmatic security when needed

The Marketplace Application

The basic Marketplace application is marketplace_v01.

The Marketplace Application

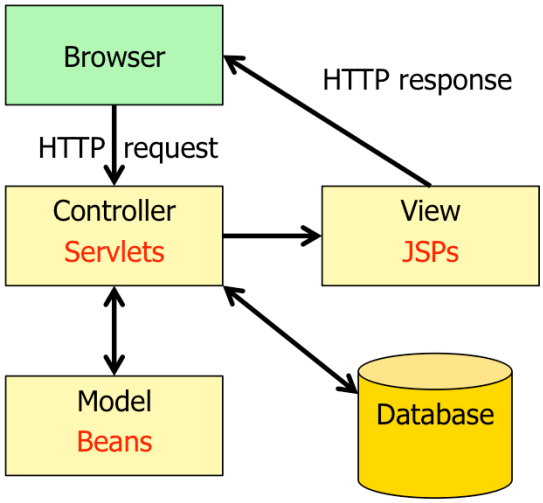
- To introduce the various security concepts, we use an example application: Marketplace
- It's a relatively simple servlet/JSP application, but it serves well to demonstrate many security problems and fitting solutions
- We will first explain the basic application to get an overview of its functionality
  - This also serves as a "refresher" of the servlet/JSP technology
  - But it's by no means an introduction to this technology – in fact, this chapter assumes you are familiar with servlet/JSP

Servlet/JSP

A very good book to get a solid introduction into servlet/JSP is the following: *Murach's Java Servlets and Jsp, ISBN-13: 978-1890774448*. In addition, the official Java EE documentation provides detailed information and tutorials, see *http://download.oracle.com/javaee/*.

Model-View-Controller (MVC) Pattern

- The application follows the MVC pattern as it is typically used with web applications based on servlets/JSP:

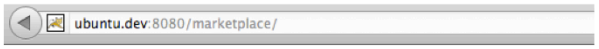


- A servlet gets requests from the browser
- The servlet controls the entire process to handle the request
  - Reads GET/POST parameters from the request (if any) and processes the request
  - If necessary, performs accesses to the database
  - If necessary, reads or writes data from/to the model, which is used to temporarily store data within the application
- Invokes a JSP, which sends the response to the browser

MVC

Not every single request must follow the pattern above. For instance, it may also be reasonable to directly request JSPs from the browser, especially when no real server-side processing happens. But for most requests, a web application follows the flow of actions depicted above.

Marketplace – Walkthrough (1)



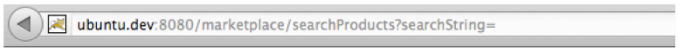
Welcome to the Marketplace

To search for products, enter any search string below and click the Search button

Search

Show cart Checkout

- The entry screen allows to search for products
  - Clicking "Search" lists all products matching the search criteria (if any)



- The resulting products list displays the search result
  - The entered search string (if any) is also displayed
  - Clicking "Add to Cart" inserts a product into the shopping cart

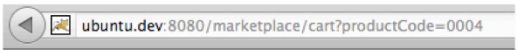
Products list

You searched for:

Description	Price	
DVD Life of Brian - some scratches but still works	$5.95	Add to Cart
Ferrari F50 - red, 43000 km, no accidents	$250,000.00	Add to Cart
Commodore C64 - rare, still the best computer ever built	$444.95	Add to Cart
Printed Software-Security script - brand new	$10.95	Add to Cart

Return to search page Show cart Checkout

Marketplace – Walkthrough (2)

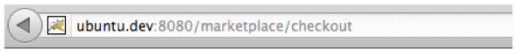


Your cart

Description	Price
DVD Life of Brian - some scratches but still works	$5.95
Printed Software-Security script - brand new	$10.95

Return to search page | Checkout

- The cart screen shows the products that have been put into the car
  - Clicking "Checkout" results in being forwarded to the checkout screen



Checkout

Please insert the following information to complete your purchase:

First name: Marc
Last name: Rennhard
Credit card number: 1234 5678 9012 3456

Purchase

Return to search page | Show cart

- The checkout page requires the user to enter the name and credit card information
  - Clicking "Purchase" completes the purchase

Marketplace – Data Model (1)

- The simple data model (the example uses the MySQL DBMS) consist of two tables (more tables will be introduced later)
- Table Product contains the products offered

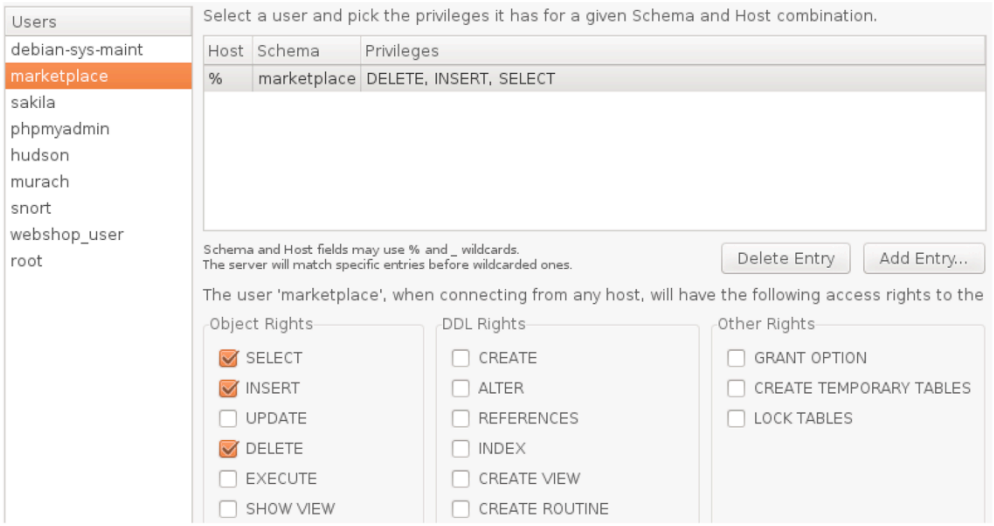
ProductID	ProductCode	ProductDescription	ProductPrice
1	0001	DVD Life of Brian - some scratches but still works	5.95
2	0002	Ferrari F50 - red, 43000 km, no accidents	250000.00
3	0003	Commodore C64 - rare, still the best computer ever built	444.95
4	0004	Printed Software-Security script - brand new	10.95

- Table Purchase contains an entry for each completed purchase

PurchaseID	FirstName	LastName	CCNumber	TotalPrice
1	Ferrari	Driver	1111 2222 3333 4444	250000.00
2	C64	Freak	1234 5678 9012 3456	444.95
3	Script	Lover	5555 6666 7777 8888	10.95
4	Marc	Rennhard	1234 5678 9012 3456	16.90

Marketplace – Data Model (2)

- The database schema is named "marketplace"
  - There's a user "marketplace" that has only the necessary rights needed in the application (principle of least privilege)

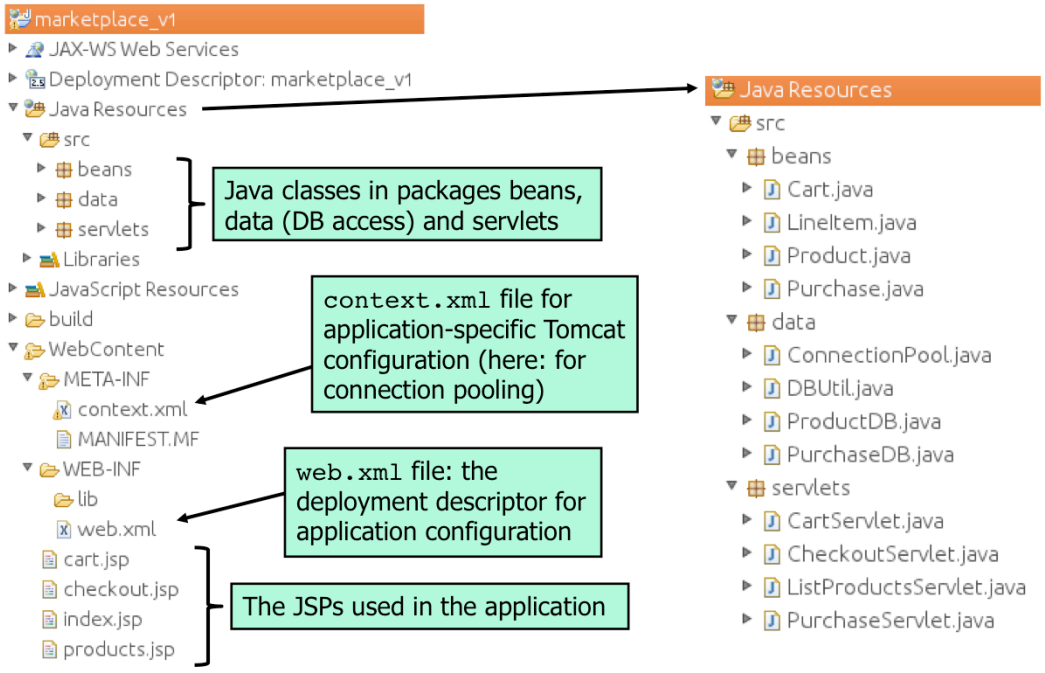


Minimal Rights

Using a database user with minimal rights can limit damage in case of an SQL injection, as the attacker cannot execute any commands (e.g. DROP, ALTER…) that have not been granted to the database user used by the application.

The DELETE right isn't needed yet, but it'll be needed later.

Marketplace – Project Organization



Marketplace – Code Snippets – index.jsp (1)

```
<html>
<head>
  <title>Marketplace</title>
</head>
<body>

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<h1>Welcome to the Marketplace</h1>

<p><font color="red">${message}</font></p>

<p>To search for products, enter any search string below and click the
Search button</p>

<form action="<c:url value='/searchProducts' />" method="get">
  <table cellspacing="5" border="0">
    <tr>
      <td><input type="text" name="searchString"></td>
      <td><input type="submit" value="Search"></td>
    </tr>
  </table>
</form>
```

We use the JSP standard tag library (JSTL), which significantly simplifies implementing JSPs

Print the content of the request attribute `message`, which was set previously by a servlet (`${...}` is the standard way of JSPs to access data)

Calls a servlet when the button is clicked (Note: we use the JSTL `url` tag for all links, which includes the session ID in the links in case the browser disables cookies)

JSTL

Today, JSP should use the JSP Standard Tag Library. It makes JSPs both easier to code and easier to read.

JSTL url tag

Cookies are used for session tracking. If the browser does not allow using cookies, Java EE uses a work-around and transmits the session ID as part of the URL (i.e. encodes the URL appropriately). This looks, e.g., as follows:

```
http://ubuntu.dev:8080/marketplace/
cart;jsessionid=C0DE93E1353A72C65B16203DCBF0BABE?productCode=0004
```

When using JSTL, simply use the url tag and URL encoding will always work correctly.

Marketplace – Code Snippets – index.jsp (2)

```
<br />

<table cellpadding="5">
  <tr valign="top">
    <td>
      <form action="<c:url value='/cart' />" method="get">
        <input type="submit" value="Show cart">
      </form>
    </td>
    <td>
      <form action="<c:url value='/checkout' />" method="get">
        <input type="submit" value="Checkout">
      </form>
    </td>
  </tr>
</table>

</body>
</html>
```

Effect of the JSTL `url` tag

- Usually, the session ID is sent by the browser in the HTTP request Cookie header
- If a user disables cookies, session tracking by the server is no longer possible and the application cannot be used
- Workaround: include the session ID in the links if cookies are disabled
  - Which is supported by Java EE by using the JSTL `url` tag

```
ubuntu.dev:8080/marketplace/searchProducts;jsessionid=679EFCAD86A7EF4F8C8FAB527456DE61?search
```

- Question: Do you think this is a good idea?

Marketplace – Code Snippets – ListProductsServlet.java (1)

```
package servlets;

public class ListProductsServlet extends HttpServlet {

  private static final long serialVersionUID = 1L;

  /**
   * Handles the HTTP GET method.
   * @param request servlet request
   * @param response servlet response
   */
  protected void doGet(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {
    processRequest(request, response);
  }

  /**
   * Handles the HTTP POST method.
   * @param request servlet request
   * @param response servlet response
   */
  protected void doPost(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {
    processRequest(request, response);
  }
```

For GET and POST request (the servlets are coded to allow both methods), call the `processRequest` method

Import Statements

Due to space restrictions, we left out the import statements above:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.ArrayList;
import beans.Product;
import data.*;
```

Marketplace – Code Snippets – ListProductsServlet.java (2)

```
  /**
   * Processes requests for both HTTP GET and POST methods.
   * @param request servlet request
   * @param response servlet response
   */
  protected void processRequest(HttpServletRequest request,
      HttpServletResponse response) throws ServletException, IOException {

    // Get the search string and get results from DB
    String searchString = request.getParameter("searchString");
    ArrayList<Product> products = ProductDB.searchProducts(searchString);

    // Store the products in the request
    request.setAttribute("products", products);

    // Forward to JSP
    String url = "/products.jsp";
    RequestDispatcher dispatcher =
        getServletContext().getRequestDispatcher(url);
    dispatcher.forward(request, response);
  }
}
```

Request attributes are available to all further components that handle this request (here: the JSP that is invoked next)

This is the standard way to forward the request / response handling to another component (a servlet or a JSP)

`ServletContext`
Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.
Source: Java Platform, Enterprise Edition 7 API Specification

Marketplace – Code Snippets – Bean Product.java (1)

- JavaBeans (often simply Beans) are used to store data within the application
  - Beans are simple Java classes that must follow a well-defined format
  - The main reason for storing data in beans is that the data can easily be accessed with `${...}` in JSPs

```
package beans;

public class Product implements Serializable {

  private static final long serialVersionUID = 1L;

  private String code;
  private String description;
  private double price;

  public Product() {
    code = "";
    description = "";
    price = 0;
  }
```

Implement the `Serializable` interface

Private instance variables that hold the content of the bean

A public standard constructor (without arguments) to initialize an "empty" bean

JavaBeans

JavaBeans must contain the following:
- A public standard constructor
- Public getter and setter methods for the instance variables
- Serializable

Import Statements

Due to space restrictions, we left out the import statements above:

```
import java.io.Serializable;
import java.text.NumberFormat;
```

Serializable

Implementing the Serializable interface is not mandatory, but common (best practice) as it easily allows storing beans e.g. in the database or in a file or send but across the network(by serializing it).

Marketplace – Code Snippets – Bean Product.java (2)

```
  public void setCode(String code) {
    this.code = code;
  }
  public String getCode() {
    return code;
  }

  public void setDescription(String description) {
    this.description = description;
  }
  public String getDescription() {
    return description;
  }

  public void setPrice(double price) {
    this.price = price;
  }
  public double getPrice() {
    return price;
  }

  public String getPriceCurrencyFormat() {
    NumberFormat currency = NumberFormat.getCurrencyInstance();
    return currency.format(price);
  }
}
```

Public getter and setter methods for all attributes

JavaBeans may also contain additional methods, here a convenience method to "pretty print" currency and price

Marketplace – Code Snippets – ProductDB.java (1)

```
package data;

public class ProductDB {

  public static ArrayList<Product> searchProducts(String searchString) {
    ConnectionPool pool = ConnectionPool.getInstance();
    Connection connection = pool.getConnection();
    Statement statement = null;
    ResultSet rs = null;
    ArrayList<Product> products = new ArrayList<Product>();

    String query = "SELECT * FROM Product WHERE ProductDescription LIKE '%"
                   + searchString + "%'";
    try {
      statement = connection.createStatement();
      rs = statement.executeQuery(query);
      Product product = null;
      while (rs.next()) {
        product = new Product();
        product.setCode(rs.getString("ProductCode"));
        product.setDescription(rs.getString("ProductDescription"));
        product.setPrice(rs.getDouble("ProductPrice"));
        products.add(product);
      }
```

Get a connection from the connection pool

Construct the DB query

Execute the query and store products in an `ArrayList` of `Product` beans

Import Statements

Due to space restrictions, we left out the import statements above:

```
import java.sql.*;
import java.util.ArrayList;
import beans.Product;
```

Connection Pooling

Opening a connection to the database is a time-consuming process that can degrade the performance of an application. It is therefore common practice to create a collection of connection objects and store them in another object, which is usually called a connection pool. The connections in the pool are then shared by all users of the web application. You usually don't "invent your own connection pool", but use one that is already available, as is the case with Tomcat.

Marketplace – Code Snippets – ProductDB.java (2)

```
    } catch (SQLException e) {
      e.printStackTrace();
      return null;
    } finally {
      DBUtil.closeResultSet(rs);
      DBUtil.closeStatement(statement);
      pool.freeConnection(connection);
    }
    return products;
  }

  public static Product getProduct(String productCode) {

    // Another method to get a single product
    ...
  }
}
```

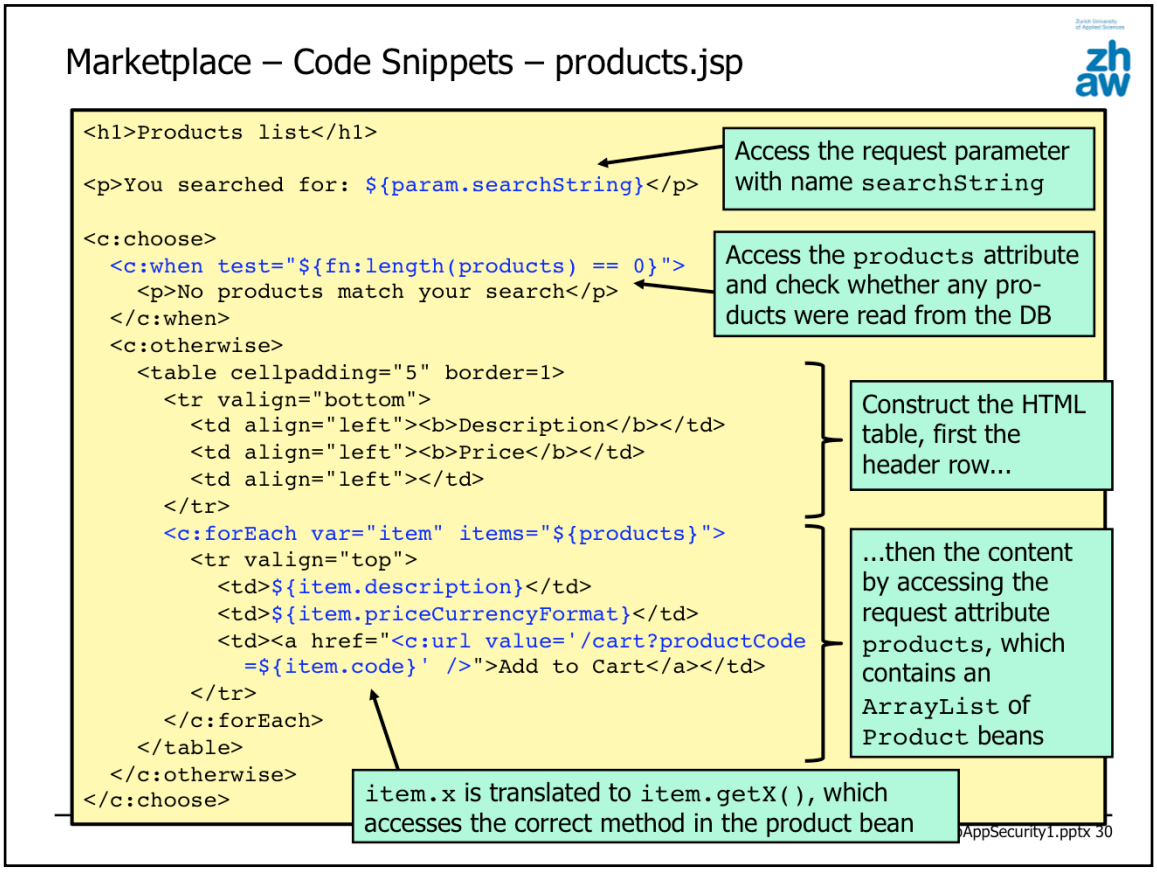
Return the connection to the pool

Marc Rennhard, 03.06.2014, SSI_JavaWebAppSecurity1.pptx 29

DBUtil

DBUtil is a utility class to close statements and result sets. The code is given below:

```
package data;

import java.sql.*;

public class DBUtil {

  public static void closeStatement(Statement s) {
    try {
      if (s != null)
        s.close();
    } catch(SQLException e) {
      e.printStackTrace();
    }
  }

  public static void closePreparedStatement(Statement ps) {
    try {
      if (ps != null)
        ps.close();
    } catch(SQLException e) {
      e.printStackTrace();
    }
  }

  public static void closeResultSet(ResultSet rs) {
    try {
      if (rs != null)
        rs.close();
    } catch(SQLException e) {
      e.printStackTrace();
    }
  }
}
```



Taglibs

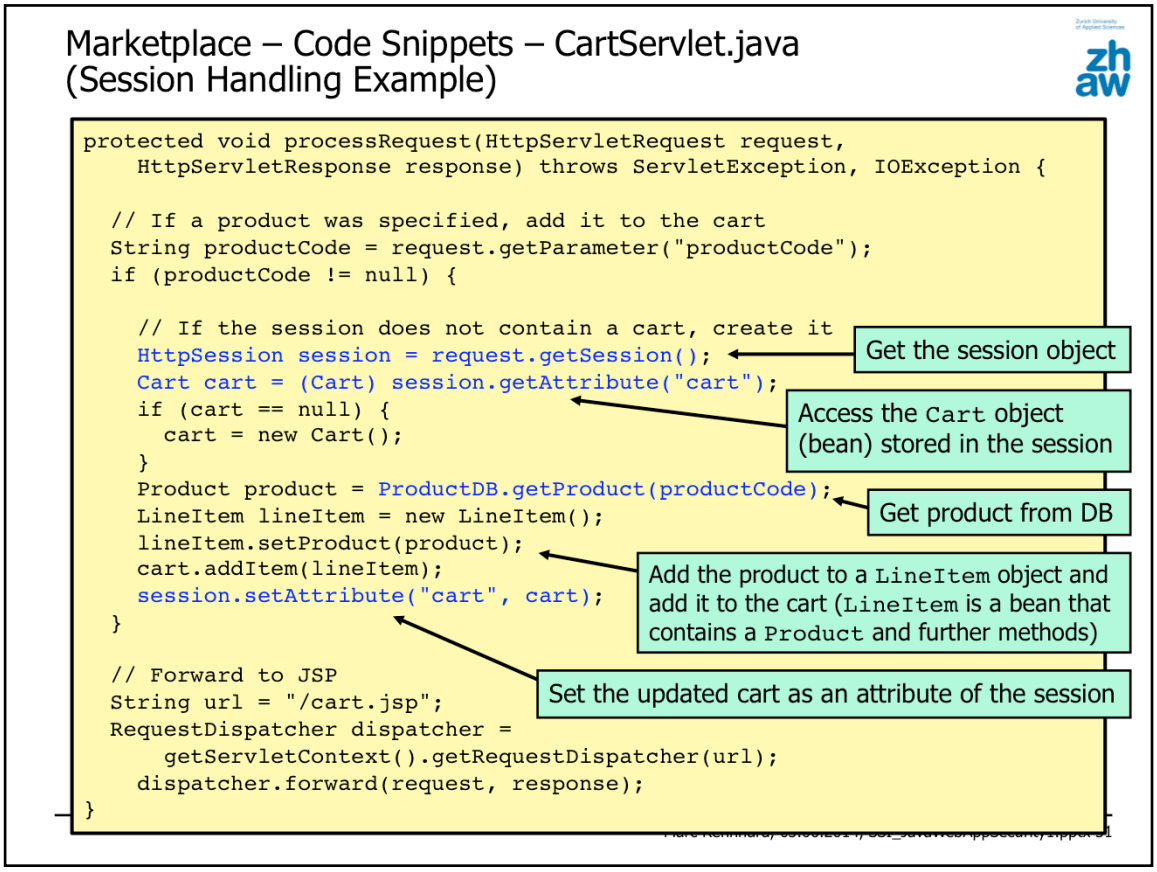
The following two JSTL taglibs must be imported in the JSP above:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
```

The <c:choose> works like a Java switch statement in that it lets you choose between a number of alternatives. Where the switch statement has case statements, the <c:choose> tag has <c:when> tags. A a switch statement has default clause to specify a default action and similar way <c:choose> has <c:otherwise> as default clause.

Collections

To access the length and the items in an attribute (products above), the object must be a subclass of the Collection Java class. This is why we used an ArrayList to store the products.



CartServlet

The Servlet is used to add a product and to display the cart contents (adding a product means there's a `productCode` parameter), that's the reason for the check at the beginning of the method.

Marketplace – Code Snippets – web.xml (1)

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">

  <servlet>
    <servlet-name>ListProductsServlet</servlet-name>
    <servlet-class>servlets.ListProductsServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>ListProductsServlet</servlet-name>
    <url-pattern>/searchProducts</url-pattern>
  </servlet-mapping>

  <servlet>
    <servlet-name>CartServlet</servlet-name>
    <servlet-class>servlets.CartServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>CartServlet</servlet-name>
    <url-pattern>/cart</url-pattern>
  </servlet-mapping>
```

Defines a servlet (name and class)

Defines the mapping of a URL to a servlet

Attributes of `web-app` Tag

You don't really have to understand the attributes of the web-app tag as it is usually created correctly by your IDE (e.g. Eclipse). In this example, you can see that the servlet version is 3.0, which is the version supported by Java EE 6.

Marketplace – Code Snippets – web.xml (2)

```
  <servlet>
    <servlet-name>CheckoutServlet</servlet-name>
    <servlet-class>servlets.CheckoutServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>CheckoutServlet</servlet-name>
    <url-pattern>/checkout</url-pattern>
  </servlet-mapping>

  <servlet>
    <servlet-name>PurchaseServlet</servlet-name>
    <servlet-class>servlets.PurchaseServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>PurchaseServlet</servlet-name>
    <url-pattern>/purchase</url-pattern>
  </servlet-mapping>

  <session-config>
    <session-timeout>10</session-timeout>
  </session-config>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

The session timeout in minutes (after 10 minutes inactivity, the server terminates the session)

The default file to serve when the request does not contain a specific resource

Marketplace – Code Snippets – context.xml

- `context.xml` is used to for application-specific tomcat configuration

`path` attribute specifies for which part of the application the configuration should be used
- Here: for the entire application
- We therefore use `/marketplace`, as all resources of the application are reached via http://host/marketplace/...

```
<?xml version="1.0" encoding="UTF-8"?>

<Context path="/marketplace">

  <Resource name="jdbc/marketplace" auth="Container"
    maxActive="100" maxIdle="30" maxWait="10000"
    username="marketplace" password="marketplace"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/marketplace?autoReconnect=true"
    logAbandoned="true" removeAbandoned="true"
    removeAbandonedTimeout="60" type="javax.sql.DataSource" />

</Context>
```

Currently, `context.xml` is only used to configure the connection pool, which is provided by Tomcat (more will follow later)

Connection Pool Configurations

For details, refer to *http://tomcat.apache.org/tomcat-7.0-doc/config/context.html*.

Connection Pool class

In the ConnectionPool class used in the Marketplace application, this context is used as follows:

```
package data;
import java.sql.*;
import javax.sql.DataSource;
import javax.naming.InitialContext;

public class ConnectionPool {
  private static ConnectionPool pool = null;
  private static DataSource dataSource = null;

  private ConnectionPool() {
    try {
      InitialContext ic = new InitialContext();
      dataSource = (DataSource) ic.lookup("java:/comp/env/jdbc/marketplace");
    } catch(Exception e) {
      e.printStackTrace();
    }
  }

  public static ConnectionPool getInstance() {
    if (pool == null) {
      pool = new ConnectionPool();
    }
    return pool;
  }

  ... (see next slide)
```

Marketplace – Exercise

Just by looking at what we have discussed so far about the Marketplace application, can you spot some security issues?

Connection Pool class (continued)

```
  ...

  public Connection getConnection() {
    try {
      return dataSource.getConnection();
    } catch (SQLException sqle) {
      sqle.printStackTrace();
      return null;
    }
  }

  public void freeConnection(Connection c) {
    try {
      c.close();
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
  }
}
```

Standard Error Handling

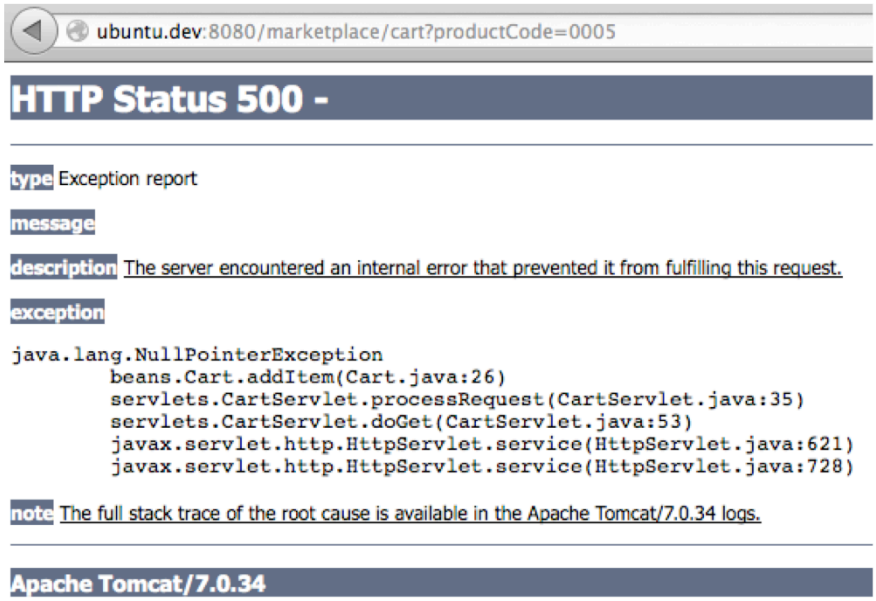
The extensions of this section are integrated in marketplace_v02.

Standard Error Handling (1)

- Per default, an error in a servlet/JSP-based web applications results in sending a container-specific message to the browser
- If an unhandled exception is thrown, this usually results in including the stack trace in the error message
  - This is convenient during development and testing
  - But should not be done in a productive system
- Example 1: accessing a non-existing resource:

Standard Error Handling (2)

- Example 2: unhandled exception:



Exception

Of course, the information disclosed in this example does not give away many details and much more critical information is imaginable (e.g. SQL error messages giving away information about the database structure. Nevertheless, one should avoid any information leakage through error message.

The exception above happens because when adding a product to the cart, it is not checked whether a product was actually received from the database. The line

```
String code = item.getProduct().getCode();
```

in `Cart.java` then causes the exception because `getProduct()` returns null. This is not directly a security problem, but a robustness problem.

Standard Error Handling (3)

- Java EE makes it very simple to enforce a standard error handling that is used throughout the application
  - This is configured in the deployment descriptor (web.xml)
- Handling of specific HTTP error codes:

```
<error-page>
  <error-code>404</error-code>
  <location>/error_404.jsp</location>
</error-page>
```

The HTTP error code

The page to display

- Handling of unhandled exceptions:

```
<error-page>
  <exception-type>java.lang.Throwable</exception-type>
  <location>/error_java.jsp</location>
</error-page>
```

The exception to handle (here: any exception)

The page to display

Marketplace – Standard Error Handling (1)

- We add the following to web.xml:

```
<error-page>
  <error-code>404</error-code>
  <location>/error_404.jsp</location>
</error-page>

<error-page>
  <exception-type>java.lang.Throwable</exception-type>
  <location>/index.jsp</location>
</error-page>
```

- This means that:
  - The page /error_404.jsp is displayed when an HTTP 404 error occurs
  - The user is simply redirected to the start page when an exception occurs

- Best practice: Add standard error handling early during development but comment the entries in web.xml and uncomment them in production!

Unhandled Exception and HTTP error 500

An unhandled exception is always sent to the browser in the form of an HTTP 500 message. Therefore, one could also use the following for the second <error-page> tag:

```
<error-page>
  <error-code>500</error-code>
  <location>/index.jsp</location>
</error-page>
```

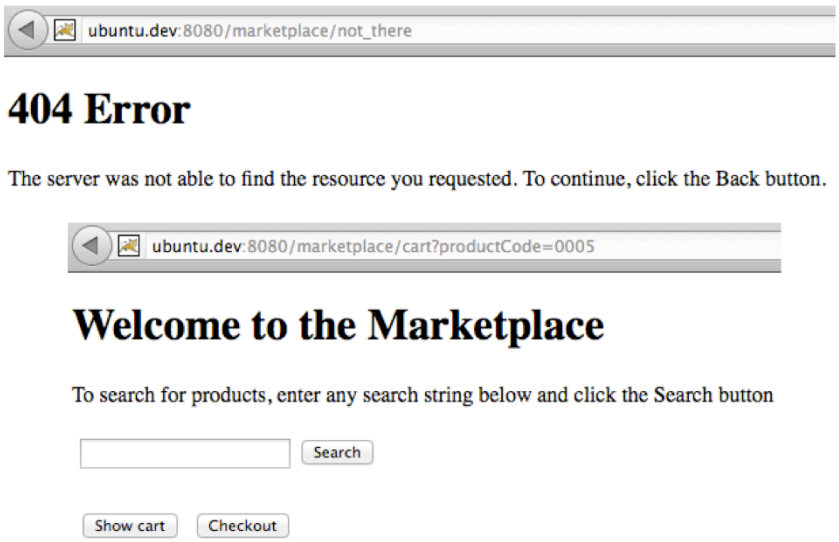
error_404.jsp

This page is very simple:

```
<!DOCTYPE html>
<html>
<head>
  <title>Marketplace</title>
</head>
<body>

<h1>404 Error</h1>

<p>The server was not able to find the resource you requested. To
continue, click the Back button.</p>

</body>
</html>
```

Marketplace – Standard Error Handling (2)

- Effect on Marketplace application:

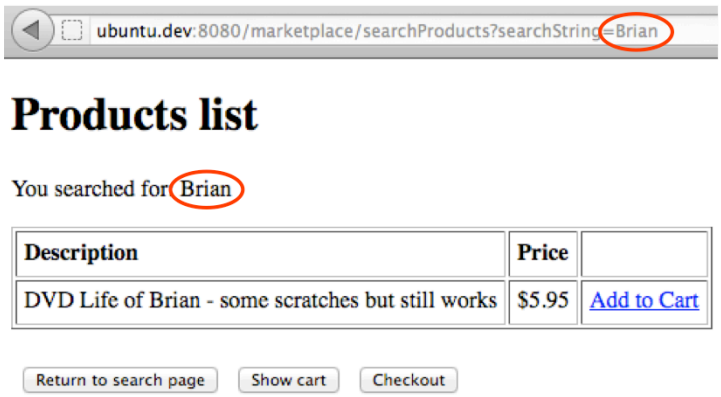


Data Sanitation

The extensions of this section are integrated in marketplace_v02.

Reflected User Data

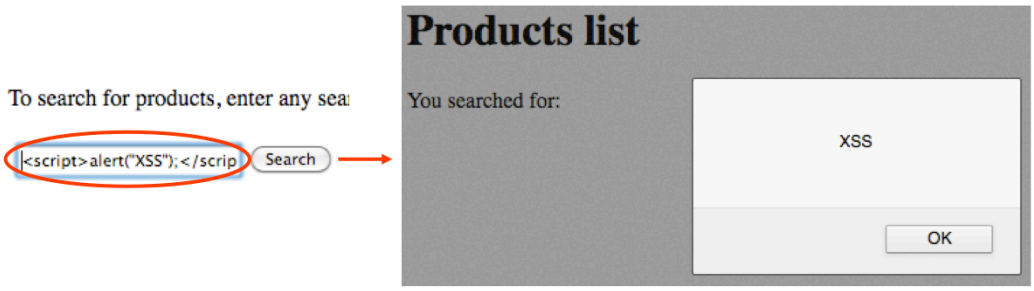
- The Marketplace application reflects the inserted search string



- Such data reflection always bears the risk of a reflected XSS vulnerability, in particular if control characters are not properly sanitized (encoded)

Reflected Cross-Site Scripting (XSS)

- Proof-of-concept of an existing XSS vulnerability



- Inspecting the HTML code confirms that no encoding / sanitation takes place:

```
<h1>Products list</h1>

<p>You searched for:<script>alert("XSS");</script></p>


    <p>No products match your search</p>
```

Input Validation and Data Sanitation

- One can argue if this is an input validation or data sanitation problem
  - It can be fixed by performing either of the two (or both – remember defense in depth?)
  - Question: What do you think – should such situations be fixed with input validation or data sanitation or both?

- With JSTL, this is extremely simple:
  - Simply use the `out` tag in the JSP file
  - This automatically handles several control characters such as <, >, " etc.

Marketplace – Data Sanitation using the JSTL out tag (1)

- Before using the out tag (vulnerable):

```
<p>You searched for: ${param.searchString}</p>
```

- Using the out tag:

```
<p>You searched for: <c:out value="${param.searchString}" /></p>
```

- Effect on application behavior:

- HTML code with encoding / data sanitation of critical characters:

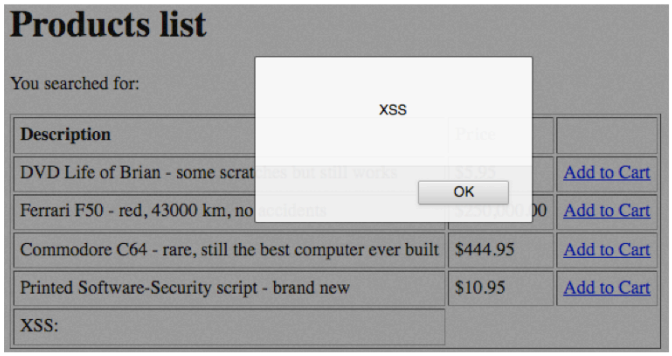
```
<h1>Products list</h1>

<p>You searched for: &lt;script&gt;alert(&#034;XSS&#034;);&lt;/script&gt;</p>
```

More Data Sanitation? (1)

- We have now fixed the case of reflected XSS via the search form
- Question: Are there other places in the Marketplace application where we should perform data sanitation? If yes, where and why?

More Data Sanitation? (2)

- E.g.: A malicious product manager / database administrator inserting a JavaScript in a product description:
  - `INSERT INTO Product VALUES ('5', '0005', 'XSS: <script>alert("XSS");</script>', '1.95')`
- Search results without using the out tag:



- → Use the out tag consistently, don't think too much about where the data may come from and who may insert malicious data

Malicious Product Manager

It may also be that product managers get an own tool to manage the products, and security (e.g. input validation) is often neglected in such internal administrator tools. All this justifies that it is very important to sanitize all data sent to the user.

Marketplace – Data Sanitation using the JSTL out tag (2)

- Correct data sanitation in products.jsp:

```
<c:forEach var="item" items="${products}">
  <tr valign="top">
    <td><c:out value="${item.description}" /></td>
    <td><c:out value="${item.priceCurrencyFormat}" /></td>
    <td><c:url var="url" value='/cart?productCode=${item.code}' />
        <a href="<c:out value="${url}" />">Add to Cart</a></td>
    ...
```

- Search results:

Products list

You searched for:

Description	Price	
DVD Life of Brian - some scratches but still works	$5.95	Add to Cart
Ferrari F50 - red, 43000 km, no accidents	$250,000.00	Add to Cart
Commodore C64 - rare, still the best computer ever built	$444.95	Add to Cart
Printed Software-Security script - brand new	$10.95	Add to Cart
XSS: <script>alert("XSS");</script>	$1.95	Add to Cart

Combining c:url and c:out

Make sure to combine c:out and c:url when displaying URLs. As the tags cannot be nested, they must be used as in the example above: Write the URL into a variable and use the variable in the c:out tag. If the c:out tag would not be used here, an XSS attack would be possible using the following for a product code: "><script>alert("XSS");</script>

Secure Database Access

The extensions of this section are integrated in marketplace_v02.

SQL Queries based on String Concatenation (1)

- Currently, the Marketplace application uses string concatenation to build SQL queries
  - This is very critical, especially if the user-supplied data is included in the string concatenation without proper input validation
- To verify an SQL injection vulnerability, we use the search function to access all data in the UserPass table
  - We will use this table later, which is why we haven't introduced it yet in the context of the basic Marketplace application
- UserPass table:

Username	Password
john	wildwest
alice	rabbit
robin	arrow
donald	daisy
luke	jollyjumper
bob	patrick

SQL Queries based on String Concatenation (2)

- In ProductDB.java, the query is built as follows:
  - `String query = "SELECT * FROM Product WHERE ProductDescription LIKE '%" + searchString + "%'";`
- To also read the contents of the UserPass table, we have to inject the following:
  - `DVD%' UNION SELECT 1,2,CONCAT_WS(" - ",Username, Password),4 FROM UserPass WHERE UserName LIKE '`
    - As only one string column of the predefined SELECT statement is displayed, we use the CONCAT_WS MySQL function to concatenate the columns for Username and Password
- Resulting query (in the Username case):
  - `String query = "SELECT * FROM Product WHERE ProductDescription LIKE '%DVD%' UNION SELECT 1,2,CONCAT_WS(" - ",Username,Password),4 FROM UserPass WHERE Username LIKE '%'`

UNION Statements

Remember: The SELECT statements combined in a UNION statement must both return the same number of columns and the data types must match.

SQL Injection in the Marketplace Application (1)

- Submitting the query as search string results in the following:

You searched for: DVD%' UNION SELECT 1,2,CONCAT_WS(" - ",Username,Password),4 FROM UserPass

Description	Price	
DVD Life of Brian - some scratches but still works	$5.95	Add to Cart
alice - rabbit	$4.00	Add to Cart
bob - patrick	$4.00	Add to Cart
donald - daisy	$4.00	Add to Cart
john - wildwest	$4.00	Add to Cart
luke - jollyjumper	$4.00	Add to Cart
robin - arrow	$4.00	Add to Cart

- Injection is also possible (somewhat easier) with SQL comments:
  - `DVD%' UNION SELECT 1,2,CONCAT_WS(" - ",Username, Password),4 FROM UserPass--`

With MySQL, a space character must follow the comment mark!

SQL Injection in the Marketplace Application (2)

- What if the attacker does not know the database schema?
  - Try to access system tables – also with SQL injection
  - With MySQL: INFORMATION_SCHEMA.TABLES and .COLUMNS
- Get all tables the DB user marketplace is allowed to access:
  - `DVD%' UNION SELECT 1,2,TABLE_NAME,4 FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_TYPE = 'BASE TABLE'--`

You searched for: DVD%' UNION SELECT 1,2,TABLE_NAME,4 FROM I

Description	Price	
DVD Life of Brian - some scratches but still works	$5.95	Add to Cart
Product	$4.00	Add to Cart
Purchase	$4.00	Add to Cart
User	$4.00	Add to Cart
UserPass	$4.00	Add to Cart
UserRole	$4.00	Add to Cart

MySQL and INFORMATION_SCHEMA.TABLES and .COLUMNS

Each MySQL user has the right to access these tables, but can see only the rows in the tables that correspond to objects for which the user has the proper access privileges (e.g. SELECT). As a result, it is usually possible to get schema information with SQL injection when MySQL is used – provided an SQL injection vulnerability exists.

See *http://dev.mysql.com/doc/refman/5.5/en/information-schema.html*

SQL Injection in the Marketplace Application (3)

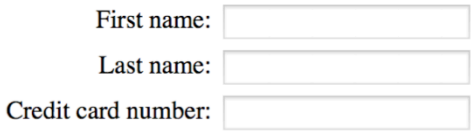
- And from the table UserPass, get the column names:
  - `DVD%' UNION SELECT 1,2,COLUMN_NAME,4 FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME = 'UserPass'--`

You searched for: DVD%' UNION SELECT 1,2,COLUMN_NAME,4 FROM

Description	Price	
DVD Life of Brian - some scratches but still works	$5.95	Add to Cart
Username	$4.00	Add to Cart
Password	$4.00	Add to Cart
Digest1	$4.00	Add to Cart
Salt	$4.00	Add to Cart
Digest2	$4.00	Add to Cart

SQL Injection on INSERT queries (1)

- We can also exploit the INSERT query that inserts a purchase
- In PurchaseDB.java, the query is built as follows:
  - `String query = "INSERT INTO Purchase (FirstName, LastName, CCNumber, TotalPrice) VALUES ('" + purchase.getFirstName() + "', '" + purchase.getLastName() + "', '" + purchase.getCcNumber() + "', " + purchase.getTotalPrice() + ")";`
- In the application, the first three values are provided, the fourth is computed internally



- How can this be exploited in an SQL injection attack?
  - Append arbitrary statements is not possible as the application uses `executeQuery()` and not `executeBatch()`
  - Option 1: Choose the value for the 4th column of the inserted row
  - Option 2: Insert additional rows in the table

SQL Injection on INSERT queries (2)

- Option 1: Choose the value for the 4th column of the inserted row by inserting the following in the credit card number field:
  - `1111 2222 3333 4444', -1000)--`
- Resulting query:
  - `INSERT INTO Purchase (FirstName, LastName, CCNumber, TotalPrice) VALUES ('Mickey', 'Mouse', '1111 2222 3333 4444', -1000)-- ', 5.95)`
- Result in DB:

PurchaseID	FirstName	LastName	CCNumber	TotalPrice
4	Mickey	Mouse	1111 2222 3333 4444	-1000.00

Negative Price

Inserting a negative total price may allow the attacker to receive money via a credit card chargeback.

SQL Injection on INSERT queries (3)

- Option 2: Insert additional rows in the table by inserting the following in the credit card number field:
  - `1111 2222 3333 4444', -1000), ('Donald', 'Duck', '5555 6666 7777 8888', 2000)--`
- Resulting query:
  - `INSERT INTO Purchase (FirstName, LastName, CCNumber, TotalPrice) VALUES ('Mickey', 'Mouse', '1111 2222 3333 4444', -1000) , ('Donald', 'Duck', '5555 6666 7777 8888', 2000)--`
- Result in DB:

PurchaseID	FirstName	LastName	CCNumber	TotalPrice
5	Mickey	Mouse	1111 2222 3333 4444	-1000.00
6	Donald	Duck	5555 6666 7777 8888	2000.00

Additional Rows

Inserting additional rows may allow the user to charge money to other users – maybe the attacker even can receive the money himself if he is offering articles in the marketplace application.

Prepared Statements

- Again, one can argue that proper input validation should prevent this
  - But what if user should be allowed to search for any strings?
- The fundamentally right approach to get protection from SQL injection in general is therefore to use Prepared Statements
- What are prepared statements?
  - Prepared statements are SQL statements that are sent to the DBMS before they are actually "used and executed"
  - When receiving a prepared statement, it is checked for syntactical correctness and precompiled by the DBMS
  - They can contain parameters that are specified later to actually execute the statement, type checking and escaping of control characters is enforced
  - Specified parameters can never change the semantics of the prepared statement
  - Side note: Prepared statements – if executed repeatedly – improve performance as syntax checking and compilation must be done only once

Do Object Relational Mapping (ORM) Frameworks such as Hibernate protect from SQL Injection?

In general, ORM frameworks provide protection as they usually use prepared statements themselves. Nevertheless, make sure to check this for a particular framework you are using. For Hibernate, you can, e.g., get here more information: *http://www.owasp.org/index.php/Hibernate-Guidelines*

Support for Prepared Statements

Most powerful relational DBMS support prepared statements. Besides SELECT (as used in the following example), prepared statements can also be used with UPDATE, INSERT and DELETE and sometimes for further (e.g. DROP, ALTER etc.) statements. More information about prepared statements in Java can be found here: *http://download.oracle.com/javase/tutorial/jdbc/basics/prepared.html*

Marketplace – Prepared Statements (1)

- Modify `searchProducts` method in ProductDB.java:

```
public static ArrayList<Product> searchProducts(String searchString) {

  ConnectionPool pool = ConnectionPool.getInstance();
  Connection connection = pool.getConnection();
  PreparedStatement ps = null;
  ResultSet rs = null;
  ArrayList<Product> products = new ArrayList<Product>();

  // Create the query string using ? to identify parameters
  String query = "SELECT * FROM Product
                  WHERE ProductDescription LIKE ?";
  try {
    ps = connection.prepareStatement(query);

    ps.setString(1, "%" + searchString + "%");

    rs = ps.executeQuery();
    Product product = null;
```

Use `PreparedStatement` instead of `Statement`

The prepared SQL statement

Send the prepared statement to the DBMS

Set the first parameter (a string) to the specified search string

Marketplace – Prepared Statements (2)

```
    while (rs.next()) {
      product = new Product();
      product.setCode(rs.getString("ProductCode"));
      product.setDescription(rs.getString("ProductDescription"));
      product.setPrice(rs.getDouble("ProductPrice"));
      products.add(product);
    }
  } catch (SQLException e) {
    e.printStackTrace();
    return null;
  } finally {
    DBUtil.closeResultSet(rs);
    DBUtil.closePreparedStatement(ps);
    pool.freeConnection(connection);
  }
  return products;
}
```

Marketplace – Prepared Statements (3)

- Modify `insert` method in PurchaseDB.java:

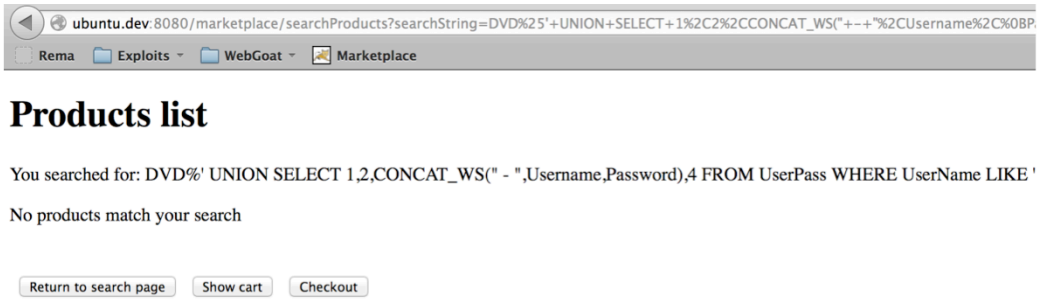
```
public static int insert(Purchase purchase) {
  ConnectionPool pool = ConnectionPool.getInstance();
  Connection connection = pool.getConnection();
  PreparedStatement ps = null;

  String query = "INSERT INTO Purchase (FirstName, LastName, CCNumber,
                 TotalPrice) " + "VALUES (?, ?, ?, ?)";
  try {
    ps = connection.prepareStatement(query);
    ps.setString(1, purchase.getFirstName());
    ps.setString(2, purchase.getLastName());
    ps.setString(3, purchase.getCcNumber());
    ps.setDouble(4, purchase.getTotalPrice());
    return ps.executeUpdate();
  }
  ...
}
```

For modifying queries, use `executeUpdate()` instead of `executeQuery()`

- Conclusion: using prepared statements requires very little adaptation – and is by no means more difficult – than using "normal" statements

Marketplace – Prepared Statements (4)

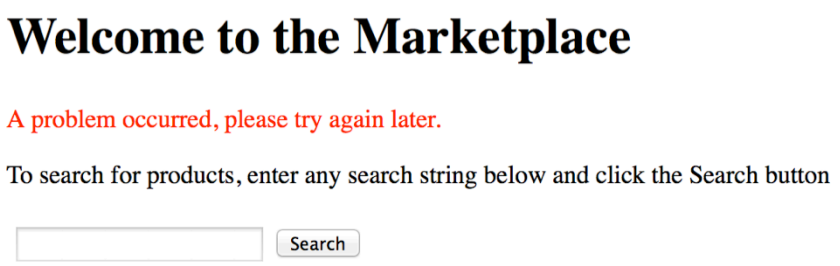
- Trying the SELECT SQL injection attack again does no longer work:



- The result set is empty because there is no product matching the search string (which is the injected SQL statement)
  - In fact, only a single SELECT statement – and not two SELECTs combined with a UNION statement – were executed

Marketplace – Prepared Statements (5)

- Trying the INSERT SQL injection attack again does no longer work (using `1111 2222 3333 4444', -1000)-- `):



- The INSERT query throws an exception (which is caught) because the value used for the CCNumber column is too large
  - In fact, the entire string `1111 2222 3333 4444', -1000)--` is used for the value, which is too long for VARCHAR(20)