8. Java Web Application Security – Part 1

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

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

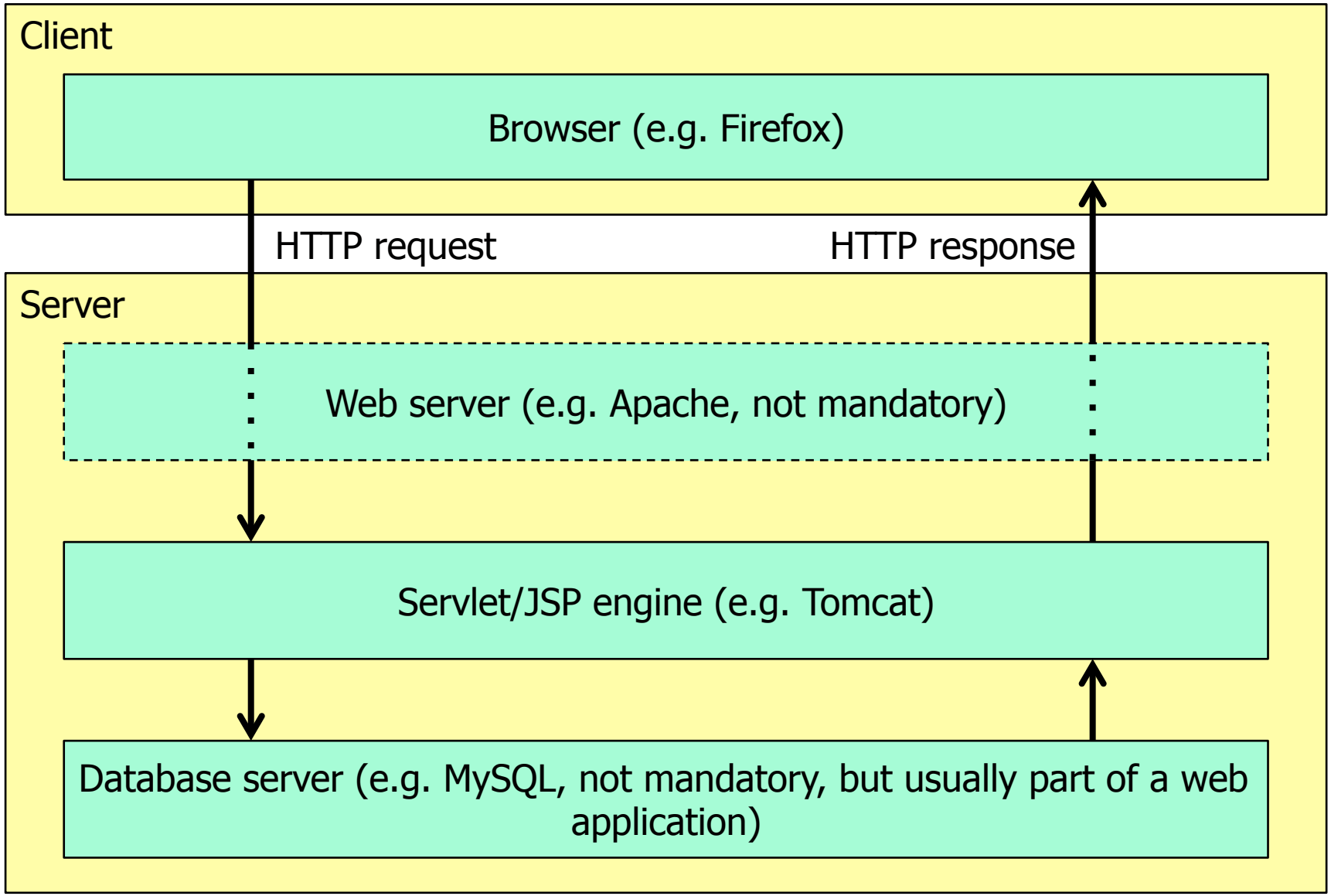
Java Web Applications

- 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

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

Components of a Servlet/JSP-based Web Application



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 conside-red 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 Marketplace Application

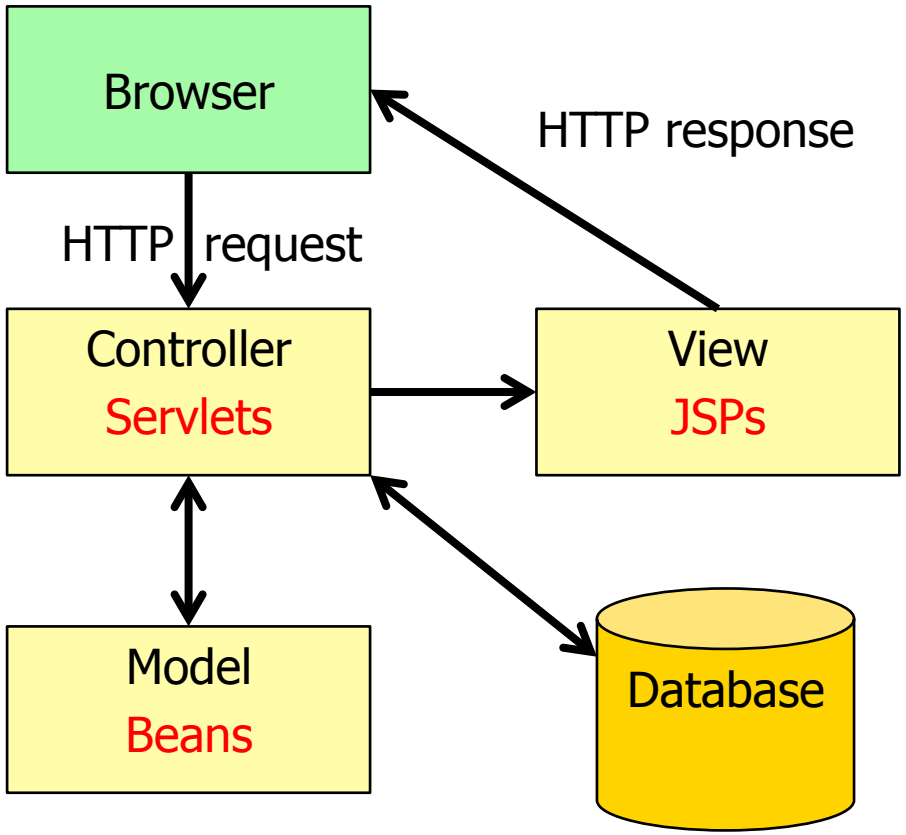
- To introduce the various security concepts, we use an example application: Marketplace

- 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

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

Marketplace – Walkthrough (1)

ubuntu.dev:8080/marketplace/

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)

ubuntu.dev:8080/marketplace/searchProducts?searchString=

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

- 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

Marketplace – Walkthrough (2)

ubuntu.dev:8080/marketplace/cart?productCode=0004

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

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

ubuntu.dev:8080/marketplace/checkout

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

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

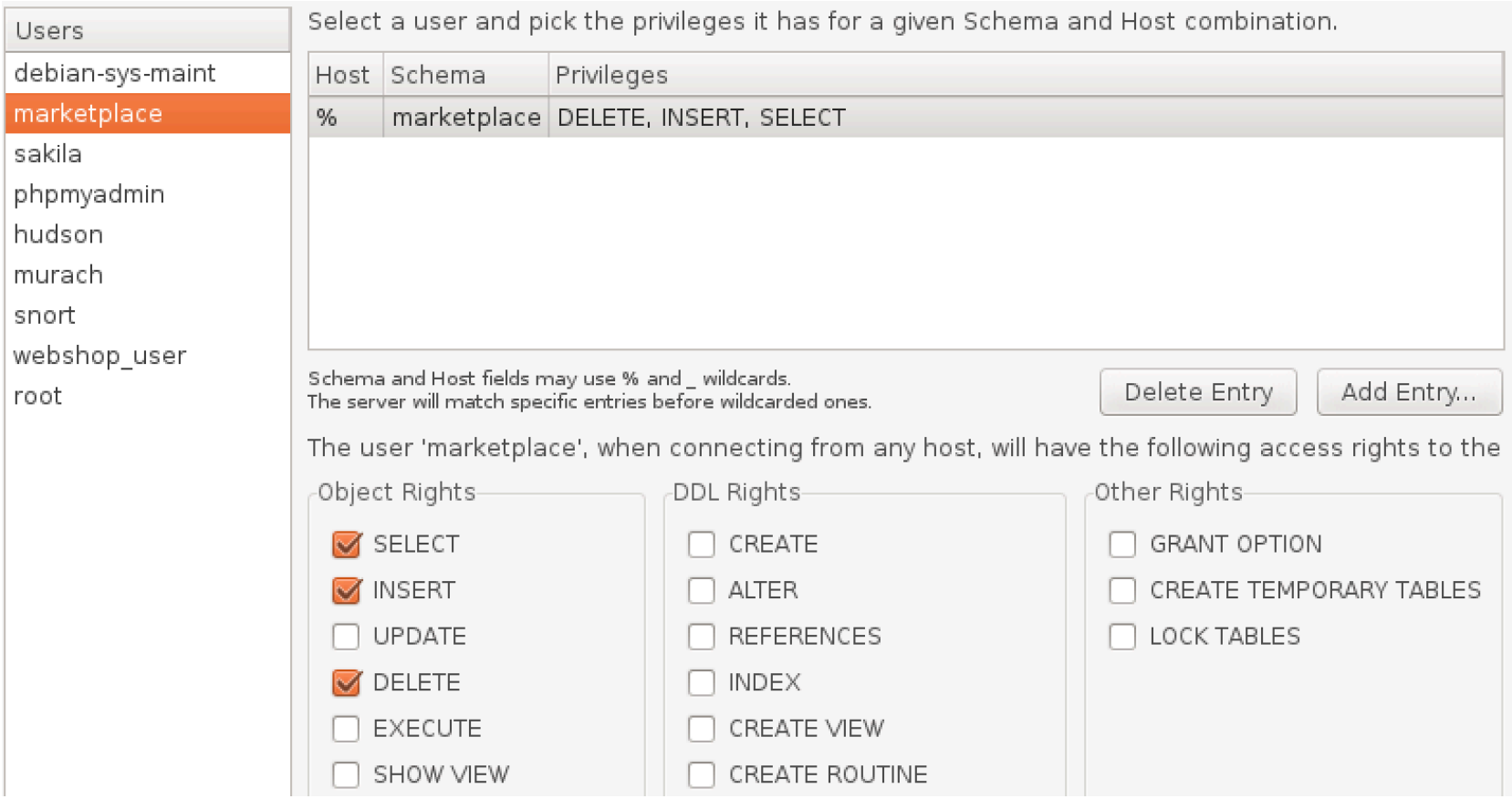
ProductID	ProductCode	ProductDescription	ProductPrice
1	0001	DVD Life of Brian - some scratches but still works	5.95
2	0002	Ferrari F50 - red, 43000 km, no accidents	250000.00
3	0003	Commodore C64 - rare, still the best computer ever built	444.95
4	0004	Printed Software-Security script - brand new	10.95

- 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)



Marketplace – Project Organization

- marketplace_v1
  - JAX-WS Web Services
  - Deployment Descriptor: marketplace_v1
  - Java Resources
    - src
      - beans
      - data
      - servlets
    - Libraries
  - JavaScript Resources
  - build
  - WebContent
    - META-INF
      - context.xml
      - MANIFEST.MF
    - WEB-INF
      - lib
      - web.xml
    - cart.jsp
    - checkout.jsp
    - index.jsp
    - products.jsp

Java classes in packages beans, data (DB access) and servlets

`context.xml` file for application-specific Tomcat configuration (here: for connection pooling)

`web.xml` file: the deployment descriptor for application configuration

The JSPs used in the application

- Java Resources
  - src
    - beans
      - Cart.java
      - LineItem.java
      - Product.java
      - Purchase.java
    - data
      - ConnectionPool.java
      - DBUtil.java
      - ProductDB.java
      - PurchaseDB.java
    - servlets
      - CartServlet.java
      - CheckoutServlet.java
      - ListProductsServlet.java
      - PurchaseServlet.java

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)

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

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)

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

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

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

Marketplace – Code Snippets – products.jsp

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

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

<c:choose>
  <c:when test="${fn:length(products) == 0}">
    <p>No products match your search</p>
  </c:when>
  <c:otherwise>
    <table cellpadding="5" border=1>
      <tr valign="bottom">
        <td align="left"><b>Description</b></td>
        <td align="left"><b>Price</b></td>
        <td align="left"></td>
      </tr>
      <c:forEach var="item" items="${products}">
        <tr valign="top">
          <td>${item.description}</td>
          <td>${item.priceCurrencyFormat}</td>
          <td><a href="<c:url value='/cart?productCode
            =${item.code}' />">Add to Cart</a></td>
        </tr>
      </c:forEach>
    </table>
  </c:otherwise>
</c:choose>
```

Access the request parameter with name `searchString`

Access the `products` attribute and check whether any products were read from the DB

Construct the HTML table, first the header row...

...then the content by accessing the request attribute `products`, which contains an `ArrayList` of `Product` beans

`item.x` is translated to `item.getX()`, which accesses the correct method in the product bean

Marketplace – Code Snippets – CartServlet.java (Session Handling Example)

```
protected void processRequest(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {

  // If a product was specified, add it to the cart
  String productCode = request.getParameter("productCode");
  if (productCode != null) {

    // If the session does not contain a cart, create it
    HttpSession session = request.getSession();
    Cart cart = (Cart) session.getAttribute("cart");
    if (cart == null) {
      cart = new Cart();
    }
    Product product = ProductDB.getProduct(productCode);
    LineItem lineItem = new LineItem();
    lineItem.setProduct(product);
    cart.addItem(lineItem);
    session.setAttribute("cart", cart);
  }

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

Get the session object

Access the `Cart` object (bean) stored in the session

Get product from DB

Add the product to a `LineItem` object and add it to the cart (`LineItem` is a bean that contains a `Product` and further methods)

Set the updated cart as an attribute of the session

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

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

```
<?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>
```

`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/...

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

Marketplace – Exercise

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

Standard Error Handling

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:

ubuntu.dev:8080/marketplace/cart?productCode=0005

HTTP Status 500 -

---

type Exception report

message

description The server encountered an internal error that prevented it from fulfilling this request.

exception

```
java.lang.NullPointerException
        beans.Cart.addItem(Cart.java:26)
        servlets.CartServlet.processRequest(CartServlet.java:35)
        servlets.CartServlet.doGet(CartServlet.java:53)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:621)
        javax.servlet.http.HttpServlet.service(HttpServlet.java:728)
```

note The full stack trace of the root cause is available in the Apache Tomcat/7.0.34 logs.

Apache Tomcat/7.0.34

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:

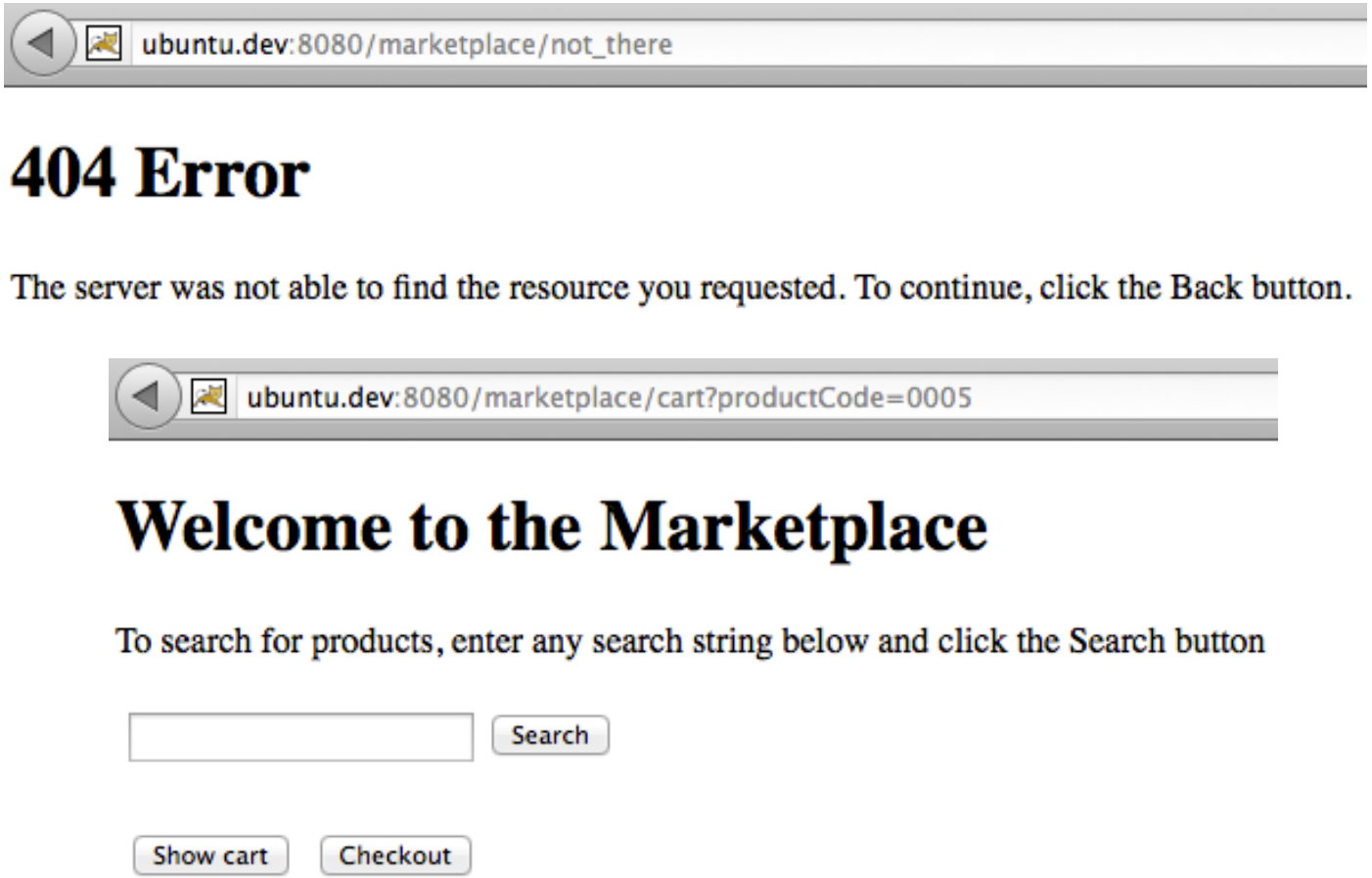
```
<error-page>
  <error-code>404</error-code>
  <location>/error_404.jsp</location>
</error-page>

<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!

Marketplace – Standard Error Handling (2)

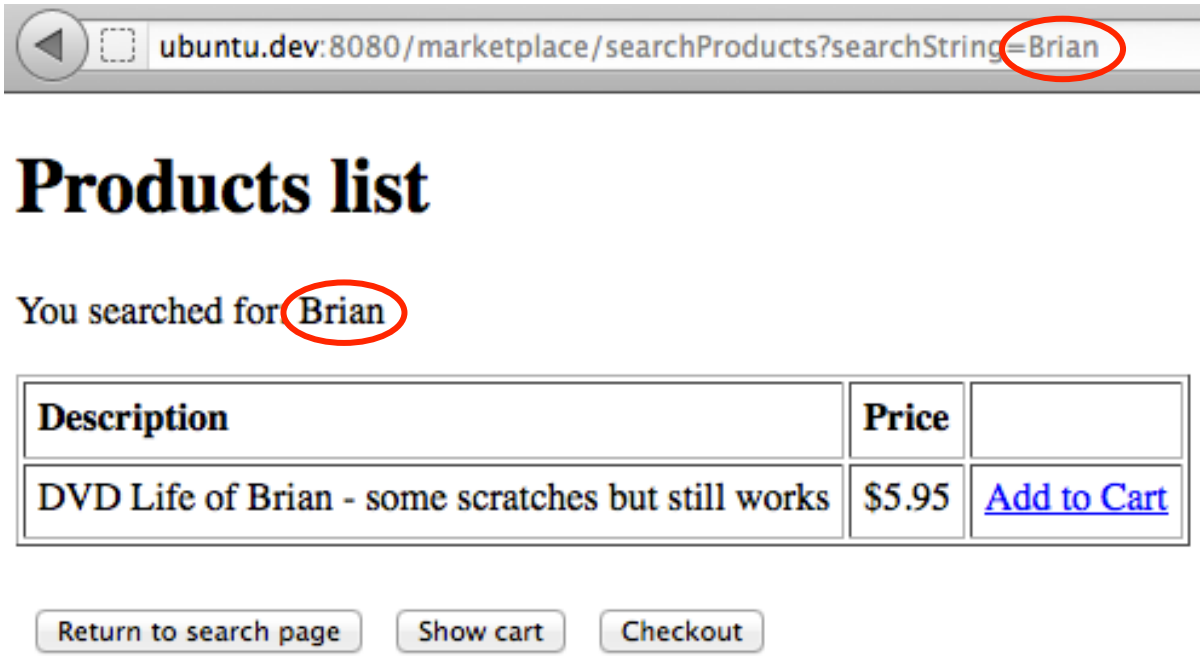
- Effect on Marketplace application:



Data Sanitation

Reflected User Data

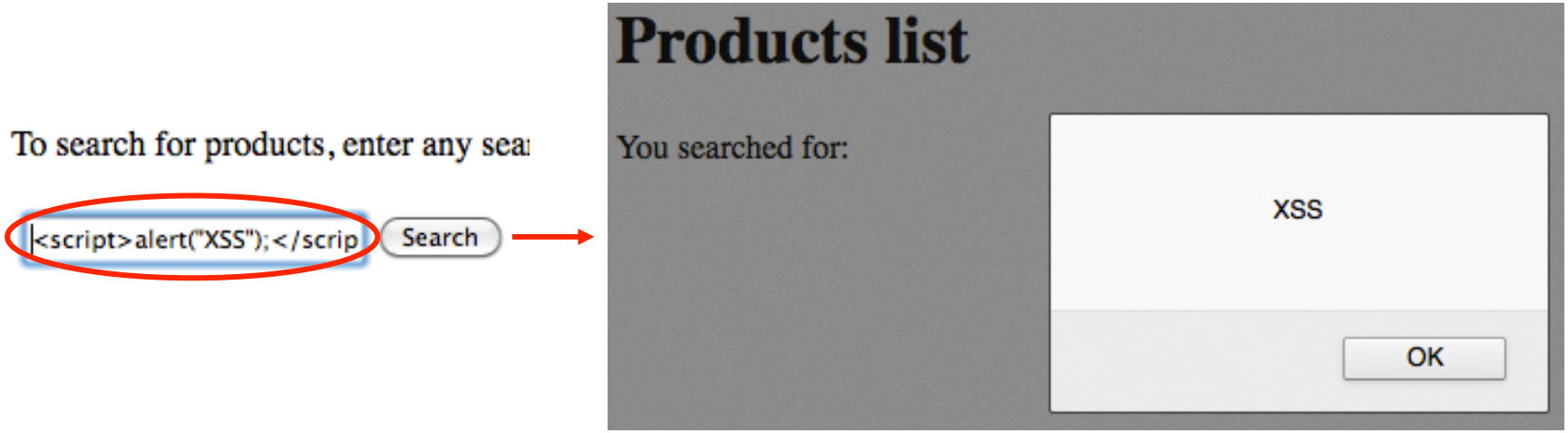
- The Marketplace application reflects the inserted search string



- 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:

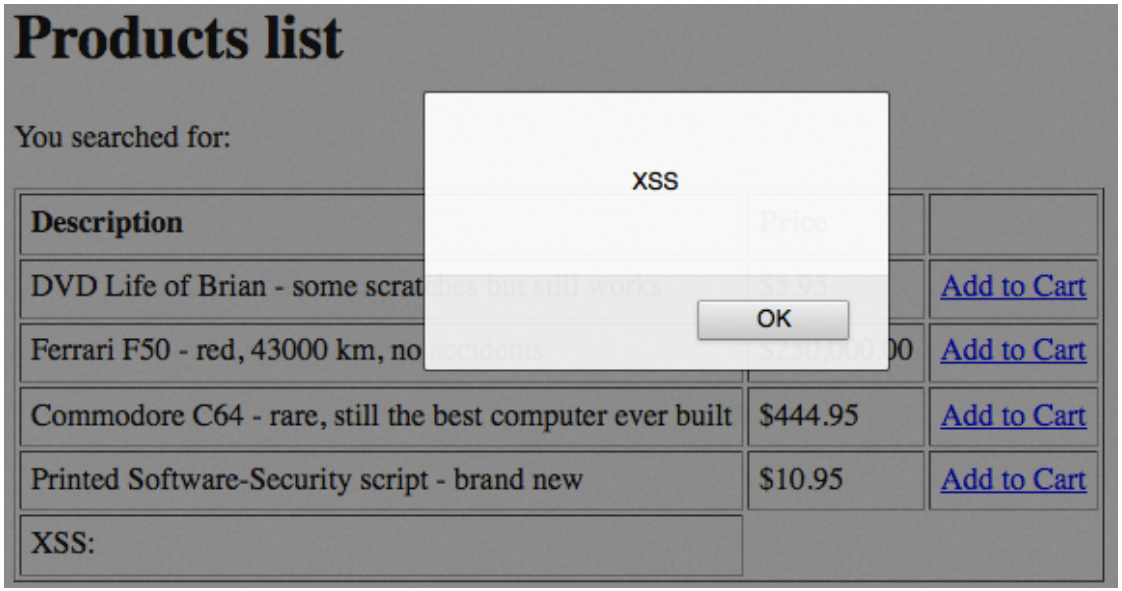
```
<h1>Products list</h1>

<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

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

Secure Database Access

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 '%'`

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

SQL Injection in the Marketplace Application (3)

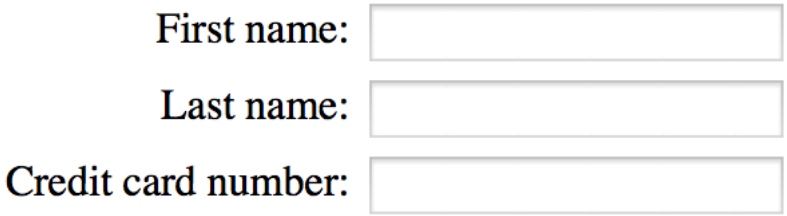
- And from the table UserPass, get the column names:
  - `DVD%' UNION SELECT 1,2,COLUMN_NAME,4 FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME = 'UserPass'--`

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

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

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

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:

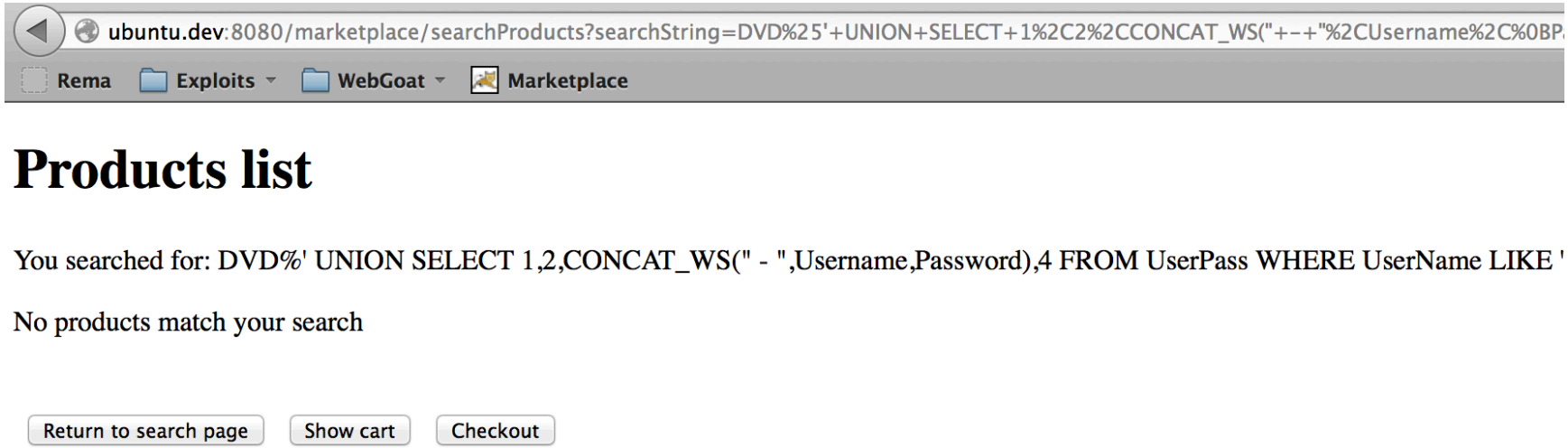
```
public static int insert(Purchase purchase) {
  ConnectionPool pool = ConnectionPool.getInstance();
  Connection connection = pool.getConnection();
  PreparedStatement ps = null;

  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)

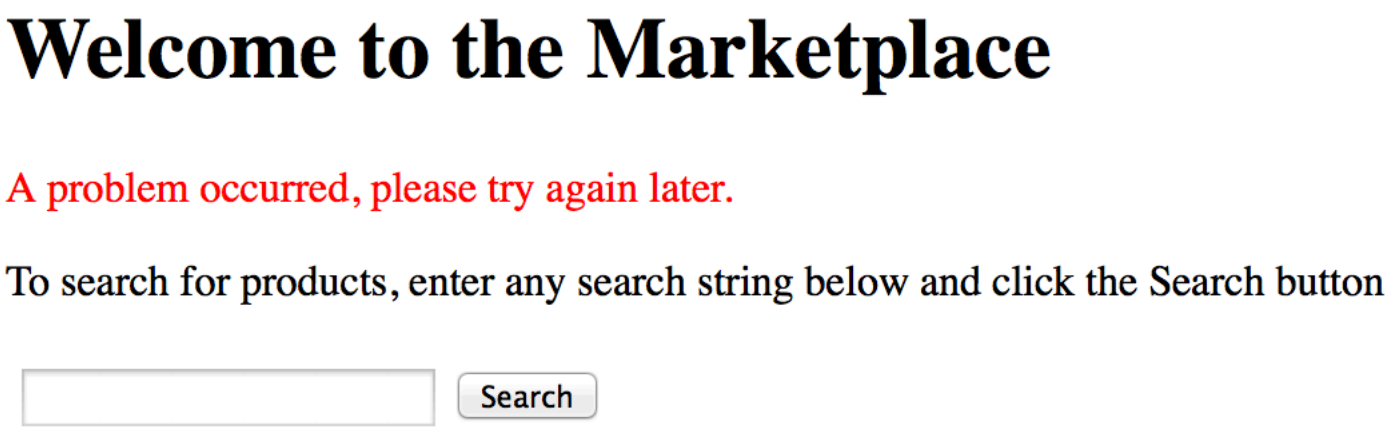
- Trying the SELECT SQL injection attack again does no longer work:



- 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)