**EE6052/ED5022/CE4208/EE4023**

Web Based Application Design

Project 3

Spring 2014

Note1: We have used Java EE7, sorry for that.

Note2: If the Message Drive Beans can not work well, please set it as the JSMSetting.pdf.

Note3: “AntiSamy” should be installed as the “Ex4.4” of lab4.pdf.

# 1 Application Functionality/Overview

When the application is initially run, the user is presented with a login screen welcoming them to the application and requesting login details from the user. It is also possible to register a new user from this page by entering new details in the User Name and Password fields then clicking the Register button. This will create a new user account which will be added to the "USERS" database and stored for future use. The required accounts for the project as described in the project brief have been initially included on the database and any subsequent account will be added as they are created. The admin account supplied in the project brief is to be the only admin account for the application, every subsequent account that is added is treated as a customer account only. As such, they will not be granted the same level of access that the admin account has been granted.

When a user logs in, they are presented with a list of products, each of which has an ID number, Name, Price, Quantity and Comments attribute. These values are all stored in the "PRODUCTS" database, which only the administrator may access and modify. Above the product list are two search bars, one to search by name and the other to search by ID. The user may use either the Name or the product ID to search for their required product.

If an incorrect value has been entered in either of the search fields the user is presented with an error page notifying them that the result has come back as invalid. It is then possible to return to the product search page, allowing the user to try again.

When a correct value has been entered, the user is presented with the corresponding database entry allowing them to add/remove a quantity of the product to their shopping cart. It is also possible to enter a comment on the product, which will in turn be added to the database.

If a valid quantity of the item is entered and added to the shopping cart, the user may click to view their shopping cart and all of its contents. This list will then be presented to the user. It is then possible to checkout of the shop, or return to add more products to the shopping cart.

When the user checks out, a summary of their purchases including the price, name and quantity of the items is presented to the user and the quantity contained in the database is reduced by the corresponding amount. This is the only way in which the user may alter the database (by purchasing an item and reducing the available quantity). The administrator is the only account allowed to update the database in any other manner.

Finally, the user may return to search and buy other products in the same manner. They may then also logout and login as a different account. The security of the application against the outlined OWASP vulnerabilities is described as the report continues.

Every time a customer confirms an order or cancels an order a corresponding entry is added to the log. Every time an administrator adds or removes a product a corresponding entry is added to the log. We store the log to “c:/12201766DataBaseLog.txt” using the “messageDriveBeans”.

Application Screenshots

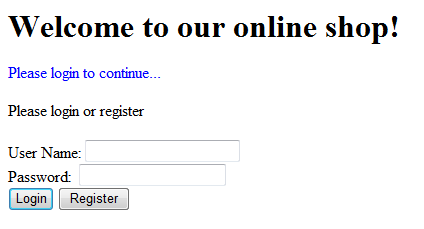


Figure 1 - Initial Login Page

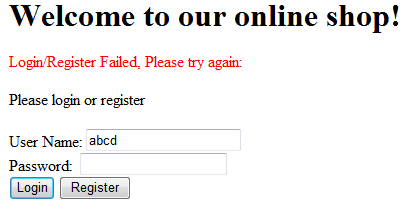


Figure 2 - Invalid Login Used

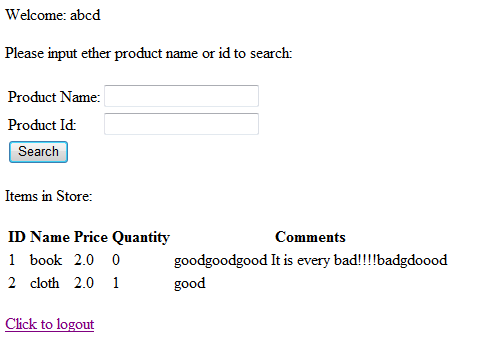


Figure 3 - Shopping Page with Item Search and Product List



Figure 4 - Add to Cart/Add Comment etc. Page

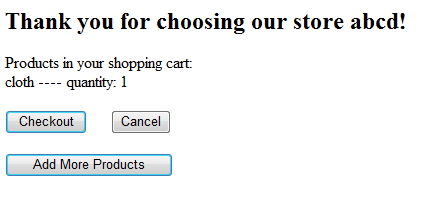


Figure 5 - Shopping Cart Page

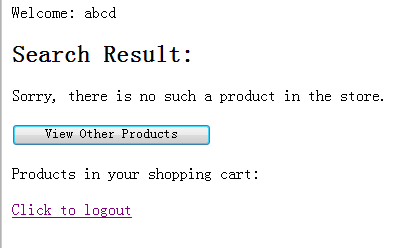


Figure 6 - Error Searching for Product

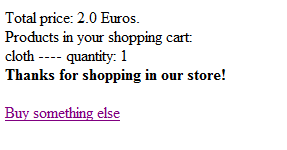


Figure 7 - Checkout Screen

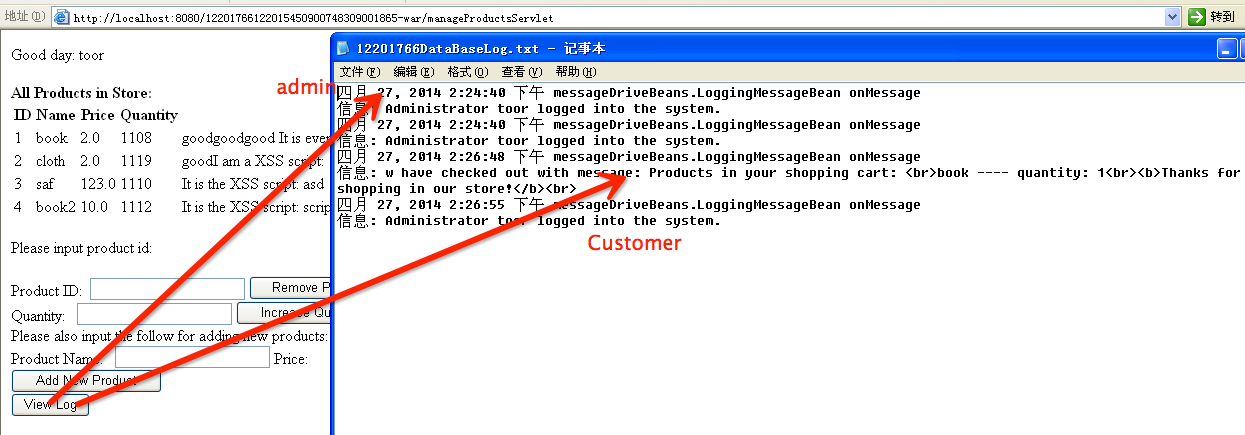


Figure 8 - Logging Screen

# 2 Protection from OWASP Vulnerabilities

## 2.1 Injection:

### Implement‬：

We have made our application resilient to Injection attacks by using prepared statements for database calls. This means that we do not use dynamically created database queries in the code. This ensures that an attacker cannot change the intended actions of a query. Our prepared statements have only been created to provide only the necessary functions for our application. The statements allow the user to perform only a set number of functions which have pre-defined behaviour that cannot be altered by an attacker. An example of some of these queries can be seen as follows:

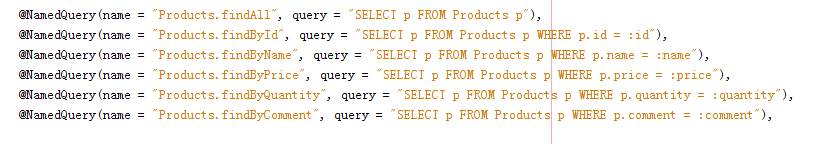
****

Figure 8 - Pre-defined Queries

We also use the Filter to filer all the input variables. The filter will detect all the variables of the input and check whether it have the commands. If it is dangerous, we will make it “white list” with the “AntiSamy”. We will introduce the “AntiSamy” later. The following is the filter:

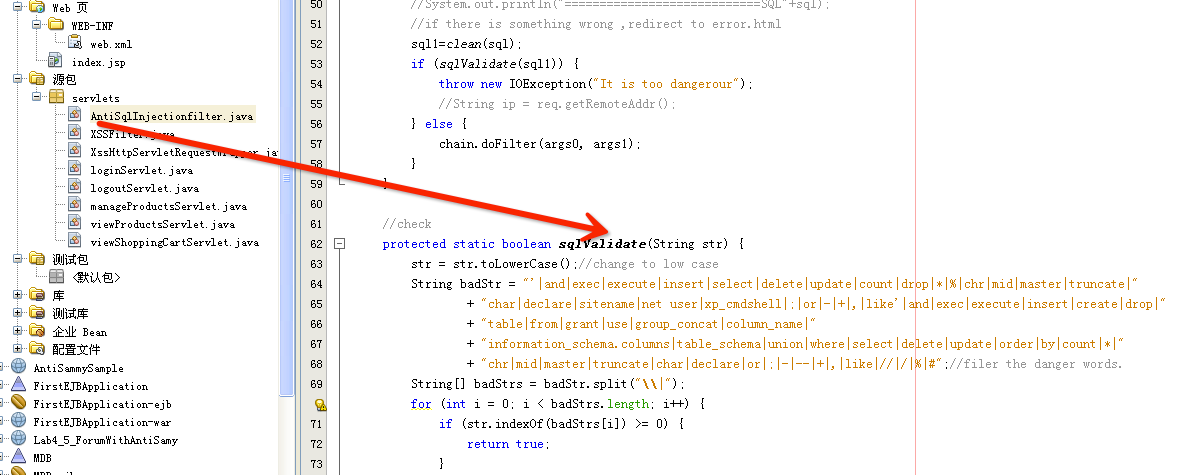


Figure 8 - filter

### Test1：

Scenario #1: The application uses untrusted data in the construction of the following vulnerable SQL call:

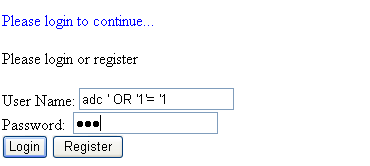


Figure 9 tests the injection.

The parameter “ abc ’ OR ‘1’ = ‘1” will be read in as the user name as a whole for query in the Users table, because there is no such a user named: “anything’ OR ‘1’ = ‘1”. So the result is Login failed.



Figure 10 tests the injection.

## 2.2 Cross Site Scripting (XSS):

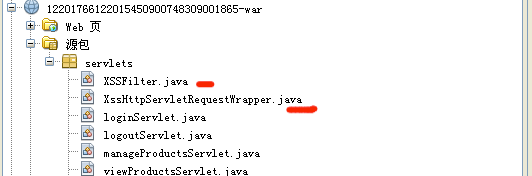
### Implement‬：

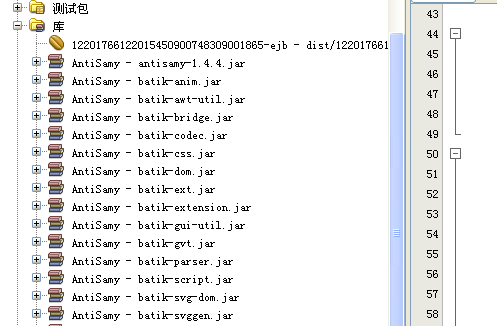
Our application is secure against cross-site scripting because we do not use much HTML throughout the project. All pages are handled through servlets. The XSS attack scenarios all involve the modification of values used in the HTML document. These are to be prevented by not allowing untrusted data to be used in the HTML documents unless a strategy has been put in place to deal with it in an appropriate manner. Since we do not use any real HTML in out project this threat has been nullified. The only HTML in this project is used to redirect to the login servlet, and from this point on all functions and redirects are performed using servlets not HTML requests. Therefore none of the user information is used in HTML documents where it may be redirected to an attackers own site. We do not use links to any other webpages in this project so the user cannot be redirected to an intruder’s site.

****

Figure 11 - HTML Document Only Used to Launch Initial Servlet

What’s more, we use the following Filter to filter the dangerous character. The filter detects all the “getParameter” methods. It filters out XSS attacks and returns a clean string. And we use the “AntiSamy” to clean the input words in the filter. Note : we set the libs as showing in the lab4.





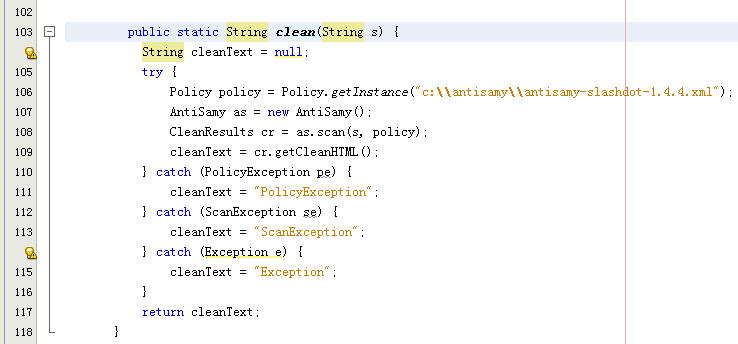


Figure 12 – The Filter for the XSS

### Test:

We write the “I**t is the XSS script: <script> alert("XSS attack!!!") </script>**” in the comments. And the web application can clean the words.

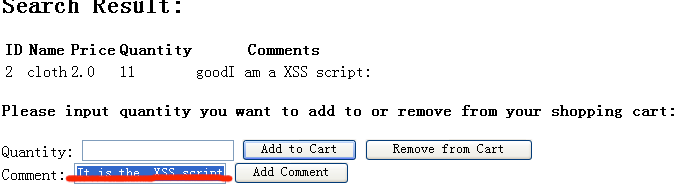


Figure 13 – The test for XSS

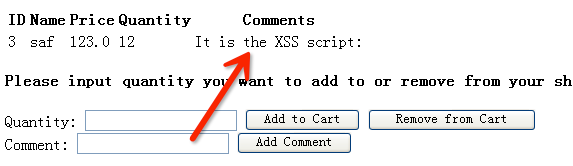


Figure 14 – The result for XSS

## 2.3 Insecure Direct Object References:

### Implement‬：

Attacks based around this mainly involve an attacker changing or modifying the URL to redirect to another resource that they may not be authorised to access. In defence against this, the application does not use direct object references in the URL or within the code itself. It is not possible for an attacker to change "http://www.ourapplication.com/getfile.cfm?filename =somefile.txt" to "http://www.ourapplication.com/getfile.cfm?filename =passwords.txt" for example.

As well as this, there is access control on the database. Only the administrator may alter the database (without purchasing). Every other user (both already registered and newly registered) is treated as just users and is not given administrator access to the application. It is not possible to create a new admin account and as such, access control has been implemented on functions and features (especially around the databases) in the application. This prevents unauthorised parties from accessing data and functions.

What’s more, we check all the servlets when they’re being calling .It is to make sure whether their have rights to access the appropriate web page. We use the cookies and sessions to check it.

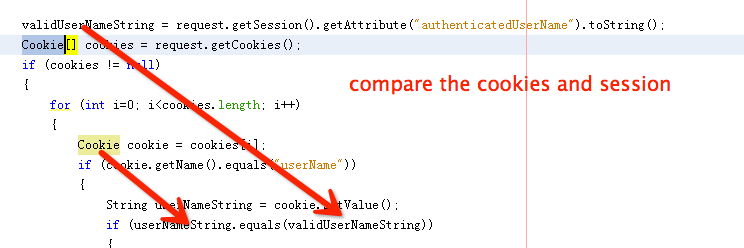


Figure 15 – The cookies and session

### Test‬：

If the customers login in the website and call the “manageProductsServlet” which is used for the administrator. Our web application will check the customer whether have the right to visit the servlet.

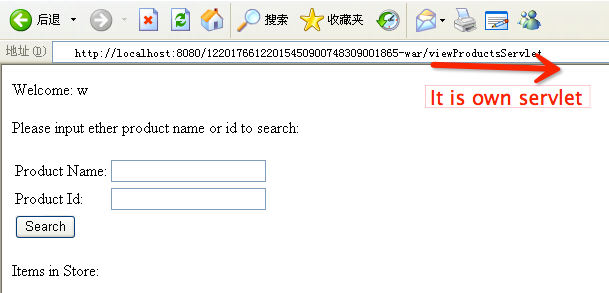


Figure 15 – The test

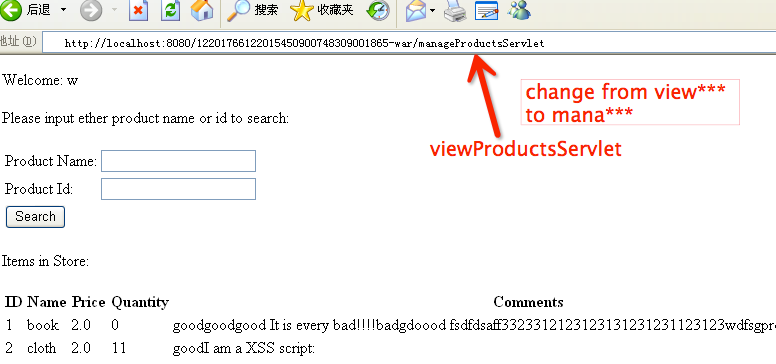
****

Figure 16 – The test

****

Figure 17 – The result

## 2.4 Failure to Restrict URL Access:

### Implement‬：

To implement a countermeasure against this it is recommended by OWASP to make user accounts and authorisation role based. In our application, there is only one administrator and every other user (new or existing) is treated and a customer only. This ensures that only one admin is allowed access to alter the database and perform certain functions. There is no "everyone" or "guest" account available; a valid log in is required to use the application. This couples with access control as only the admin may perform admin-authorised functions and every other user is restricted to a certain number of functions. Because of these implemented restrictions only an authorised party may access certain URLs, thus improving the security of the application.

The technology we used in “Insecure Direct Object References" can also defend the attack. Because we check all the servlets in our web application, check whether the cookies and session is same or not. This can prevent the “Failure to Restrict URL Access”.

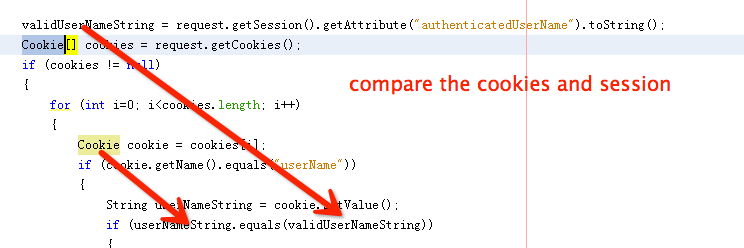


Figure 18 – The Check for cookies and session

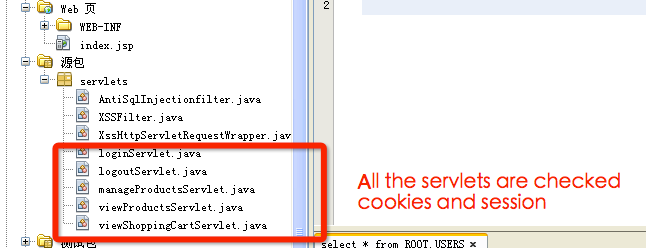


Figure 19 – All the servlets

**Test:**

The user steals the following web address, but he doesn’t have the account and password.”http://localhost:8080/12201766122015450900748309001865-war/viewProductsServlet” .He can’t open the page, and he will go to the login page.

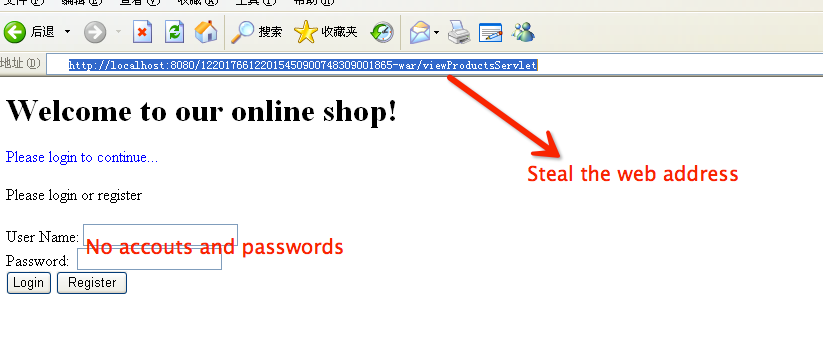


Figure 20 – steal the address



Figure 21 – Login page