Security Lab – Java Web Application Security

VMware

- You will work with the **Ubuntu image**, which you should start in networking-mode **Nat**.

1 Introduction

In this lab, you are extending an existing Java web application with a focus on using the security features offered by Java SE und Java EE correctly. The goal of the lab is that you get more experienced with using these security features so that you can use them correctly and consistently in your future applications.

The Marketplace application from the lecture is used as the basis. We are using the final version, which already includes all extension from the lecture (including annotations, prepared statements, input validation with ESAPI, and CSRF protection). In the following, it is assumed that you are familiar quite well with this final state of the application.

2 Basis for this Lab

On the Ubuntu image, there exists an Eclipse project *MarketplaceLab*, which is based on the final version of the Marketplace application from the lecture.

- To start Eclipse, open a terminal, change to */home/user/eclipse*, and enter *./eclipse*.
- The project directory is */home/user/workspace/MarketplaceLab*.

3 Deploying and Using the Application

On the image, Tomcat 7 is installed (*/home/user/tomcat7*). To deploy the application, do the following:

- Right-click the project in the *Project Explorer* (left window) in Eclipse, select *Export...*, and then select *Web → WAR file* in the list.
- Enter the following in the field *Destination* in the window that has just been opened: */home/user/tomcat7/webapps/marketplace.war*

To start or stop Tomcat, enter the following in a terminal in directory */home/user/tomcat7/bin* (as *user*):

- `./startup.sh`
- `./shutdown.sh`

When deploying a new WAR file, Tomcat usually does not have to be restarted. Using the a browser, you can reach the application as follows:

- `http://localhost:8080/marketplace/`

If you encounter any problems, the Tomcat logs under */home/user/tomcat7/logs/* may help.

When accessing the secure area (HTTPS), you'll likely get a certificate warning. It's easiest to accept this certificate permanently to get rid of future warnings.

Die database contains the test data you know from the lecture. If you have to reset them for any reasons to the initial state, you can do this as follows:

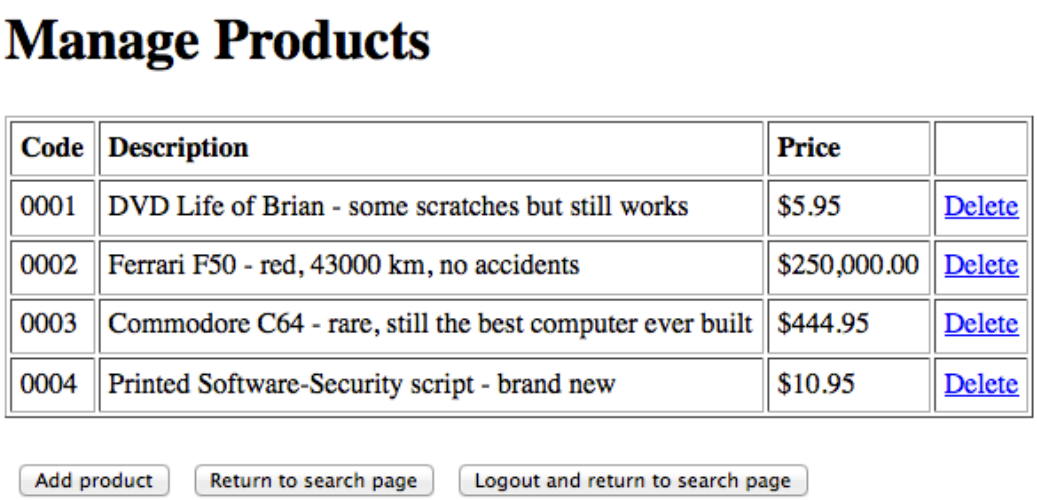
- Open the *MySQL Workbench* (Menu *Applications → Programming*).
- Double-click on the left to *Local MySQL Server* and enter *root* as password.

- Choose *Open SQL Script...* in the menu *File* and select the file */securitylab//j2eesecurity_marketplace/Marketplace.sql*.
- Click the *Execute* icon.

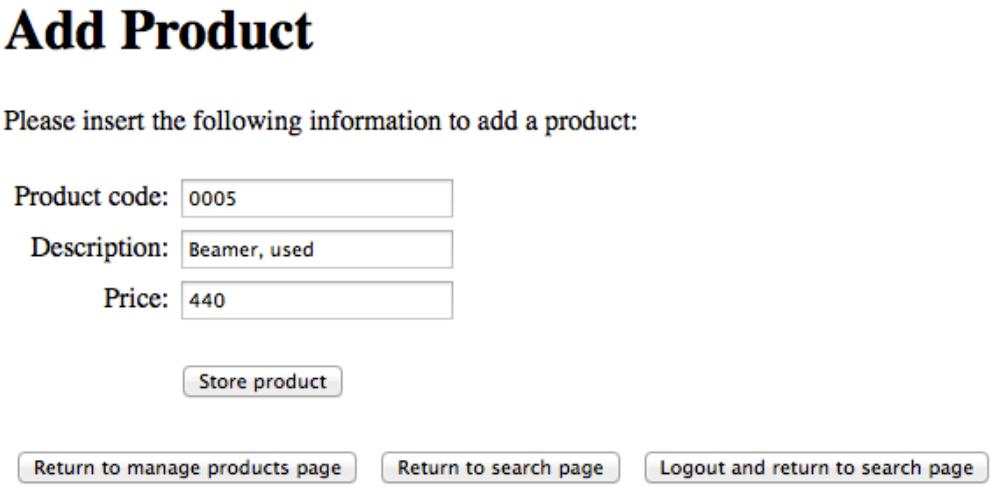
4 Task: Product Administration

Your task is to extend the application with a simple product administration. It is specified as follows:

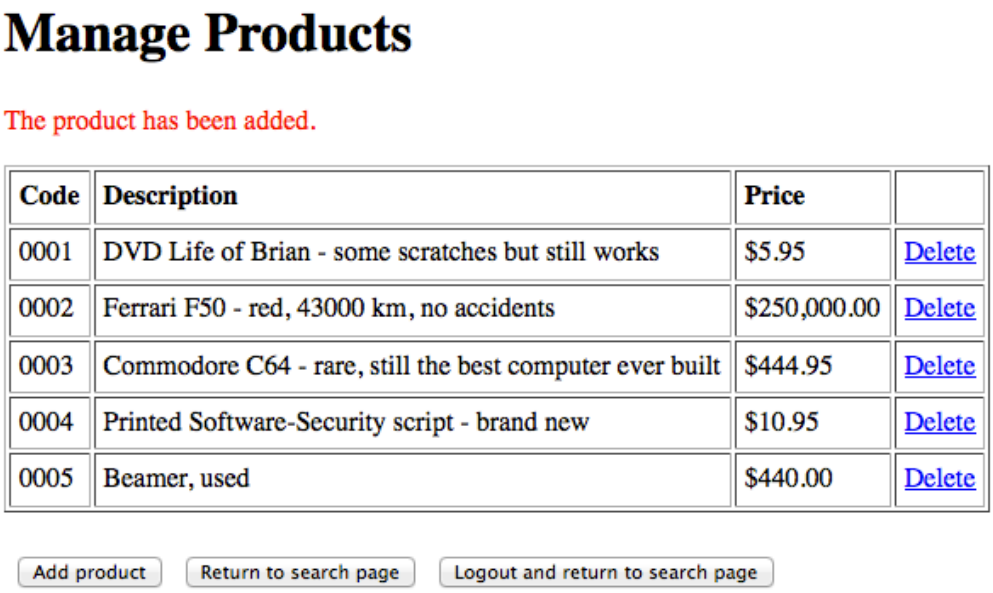
- Only users with role *productmanager* can administrate products, other users don't get access to the product administration.
- There are two users with this role:
  - *donald* with password *daisy*
  - *luke* with password *jollyjumper*
- The product administration is part of the admin area, which is already used by users with roles *sales* or *marketing* to view and complete purchases. As a result, the existing login form will also be used to access the product administration.
- If a user with role *productmanager* logs in, he should be forwarded to the following page, where all existing products are listed:



- Clicking *Delete* deletes the product (without requesting confirmation).
- Clicking *Add product* shows the following page:



- Entering a product and clicking *Store product* adds the product and forwards the user again to the *Manage Products* page:



A lot of the new functionality is already provided, so you can really focus on the security aspects. Compared to the final version from the lecture, the following servlets and JSPs have been added:

- Servlets: *ManageProductsServlet*, *DeleteProductServlet*, *AddProductServlet*, and *Store-ProductServlet*, which are all located in package *Servlets*.
- JSPs: *manageproducts.jsp* and *addproduct.jsp*, which are located in *WEB-INF/pages/admin/*.

The necessary entries in the deployment descriptor *web.xml* have also already been added.

Not that the new components always use GET requests, which means the parameters are visible in the address bar in the browser. This does not conform to best practice but makes testing the application easier. In addition, Tomcat error messages are not suppressed, which is also typical during development.

In the following, you'll be guided step-by-step through the extensions you have to make. Of course you can implement the extensions in another order, but following the steps will help you to approach the solution in a reasonable order and to carry out all necessary steps. Important: Do only proceed to the next step once you are sure "everything so far works correctly".

4.1 Get familiar with the new components

First, get familiar with the new components. You should understand the interactions between the new servlets and JSPs. Also, inspect web.xml to see the mapping of URLs to servlets.

4.2 Adding the new role *productmanager*

Add the new role *productmanager* to web.xml at a suitable location. The role is now defined and can be used in the application.

If you access now the *Admin area* and log in as user *donald*, you are forwarded to the *Purchases*, which certainly does not correspond to the specifications. You therefore have to adapt the code such the that new role is used correctly. You can do this as follows:

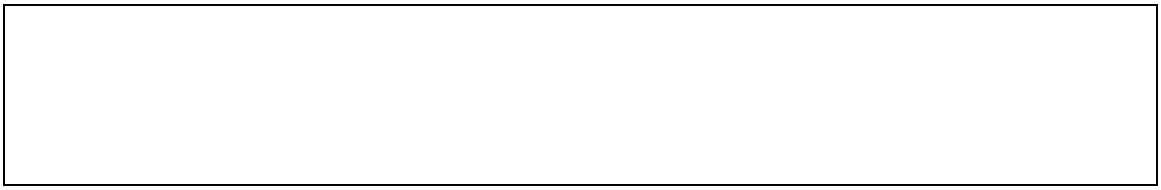
- Adapt *LoginServlet* such that the new role is used correctly. To do this, you have to adapt the method *processRequest* such that a user with role *productmanager* is forwarded to */admin/manageproducts* (which is mapped to *ManageProductsServlet* in web.xml).

- Note that the method consist of „two parts“. The „upper part“ (if) handles the case if the user clicked on the button *Admin area*. The „lower part “ (else) handles the case if the servlet is called from the login form. You must adapt both parts correctly.
- You should also handle the case if a user that has another role than *sales*, *marketing* or *product-manager* logs in. To test this, there exists a user *bob* with password *patrick*, which has another role. If such a user logs in correctly, he should not get access to the admin area but should be logged off right away (forward to */admin/logout*) and then be forwarded to */index.jsp* (this second forward is already implemented correctly in *LogoutServlet*).
- You must also adapt the security annotations in *LogoutServlet* to allow users with the new role *productmanager* to log out.

Test the application and especially consider the following cases:

- Does a user with role *productmanager* indeed get to the *Manage Products* page?
- Do users with the „old “ roles *sales* (user *alice*, password *rabbit*) and *marketing* (user *robin*, password *arrow*) still get to the correct pages?
- Can users log out and are they truly logged out?
- Are users (*bob*) with other roles prevented from reaching the admin area and are they correctly logged out right after logging in?

One final question: If you have done everything as described above, *bob* is logged out right after logging in, although his role does not have the permission to access the servlet *LogoutServlet*. Why does it work nevertheless?



4.3 Security annotations of the new servlets

During the previous step, you made sure that authenticated users are forwarded to the right place in the application. However, it is currently possible for any user (authenticated or not) to access the product administration by using forced browsing. Test this by entering the following URL in the browser (not as a user with role *productmanager*):

- `http://localhost:8080/marketplace/admin/manageproducts`

To prevent this, you must define in each servlet who is allowed to access it. Add the correct security annotations to all new servlets according to the following rules:

- Only GET and POST methods are allowed and only for role *productmanager*.
- Access is only allowed via TLS.

Test whether access control now works correctly (using forced browsing). The application should behave as follows:

- Entering `http://localhost:8080/marketplace/admin/manageproducts` always results in a redirect to `https://localhost:8443/marketplace/admin/manageproducts`.
- Accessing `https://localhost:8443/marketplace/admin/manageproducts` is only possible for an authenticated user with role *productmanager*.

Since the other three new servlets have received the same security annotation, they should behave in the same way. For completeness, you should test them as well. You can get the URLs to use from the servlet mappings in web.xml.

4.4 Secure database access

Two new methods *insertProduct* and *deleteProduct* must be implemented to access the database. The methods are already used correctly by the servlets *StoreProductServlet* and *DeleteProductServlet*, but they have not yet been implemented in *data/ProductDB.java* (the methods exist but the method bodies are empty). To prevent SQL injection attacks, make sure to use prepared statements. You should also free the resources (the connection) before leaving the methods. The methods are specified as follows:

- *insertProduct(Product product)* receives a Product bean (see *beans/Product.java*), in which code, description, and price of a product are specified. The product should be included into the database. The primary key (Column ProductID in table Product) is automatically generated by the DBMS. You don't have to check whether a product with the same product code already exists (although you can do this if you like). The return value of the method corresponds too the number of inserted rows (1 or – in the case of a problem, 0).
- *deleteProduct(String productCode)* receives a product code and deletes the product with this code (Column ProductCode). The method returns the number of deleted rows (typically 1 or – in the case of a problem, 0).

To get details about the database structure and the tables, use the *MySQL Workbench* (home button on the top left, enter *mysql* and click on the program icon, then a double-click on *Local MySQL Server* and use *root* as password. The database scheme is named *marketplace*.).

Test if you can store products and delete them again. Currently, some entries will results in exceptions (e.g. using an arbitrary string as the price), but this will be fixed later.

4.5 Data sanitation

Enter a product with description `<script>alert("XSS");</script>`. When returning to the *Manage Products* page, the script will be executed. This means the application currently contains a stored XSS vulnerability – „stored" because the script is inserted into the database. Using this vulnerability, a malicious product manager can attack other product managers (and e.g. present them a fake login screen to steal their credentials). If the vulnerability also exists in other places in the application, further user groups can be attacked as well.

One can argue that the script should never find its way into the database in the first place and this should be checked when entering a product. That's true and we will fix this below. However, you never know what additional methods may exist (or will exist in the future) to enter products (e.g. directly in the database, via another tool etc.). Or it may even be that scripts are in fact allowed in product descriptions. Therefore, it is important that critical characters are always HTML-encoded before inserting them into a generated web page to prevent possibly available scripts from being executed in the browser.

As you have learned in the lecture, JSTL offers a simple way to perform data sanitation. Apply this correctly to *manageproducts.jsp* (to all displayed data, including the *Delete* link, behind which there's also data from the database). Test whether the script is only displayed and no longer executed. Leave the JavaScript in the product table because after implementing the input validation, it will no longer be possible to enter such a script.

As soon as this works, inspect the HTML code displayed in the browser. Which critical characters were replaced and how?

4.6 Input validation

Next, you should implement a proper input validation. This should guarantee that only products that follow a certain standard can be entered. To perform input validation, we used OWASP ESAPI in the lecture, which you'll use here as well.

The three fields of a product are defined as follows:

- *Product Code*: Contains exactly 4 characters consisting of any combination of letters (lower- and upper-case) and digits.
- *Description*: Contains at least 10 and at most 100 characters. Allowed characters are letters (lower- and upper-case), digits, the space character, and the special characters comma (,), quote (‘) and dash (-).
- *Price*: A decimal number in the range 0 – 999'999.00. This means that there may also be products that are for free. In addition, at most two decimal places can be entered and a dot (.) must be used as decimal mark. In front of the decimal mark, there must be at least one digit, but using the decimal places is optional. The following entries are all correct: 100 ; 100. ; 100.9 ; 100.95. The following entries are invalid: 3.567 ; .95 ; 1,50.

Using ESAPI is explained in the lecture slides. Proceed as follows:

- In *util/Validation.java* there exist method headers for the necessary validation methods. Implement them according to the already available method *validatePersonName*.
- For each of the three methods, a regex must be configured in */home/user/.esapi/ESAPI.properties*, ideally right at the end where the *Marketplace rules* are found. Of course, they have to fulfill the requirements above. In addition, write them into the following box:

  Product Code:

  Description:

  Price[1]:

- Restart tomcat to make sure the new configurations are used.
- Implement the input validation in *StoreProductServlet* and insert the product into the database only if all rules are fulfilled. If this is not the case, the user should be redirected again to the *Add Products* page and a suitable error message should be displayed. You can use the already existing validation code in *PurchaseServlet.java* if you need "some inspiration".

Test if the input validation works correctly. Test all three fields and proceed as follows:

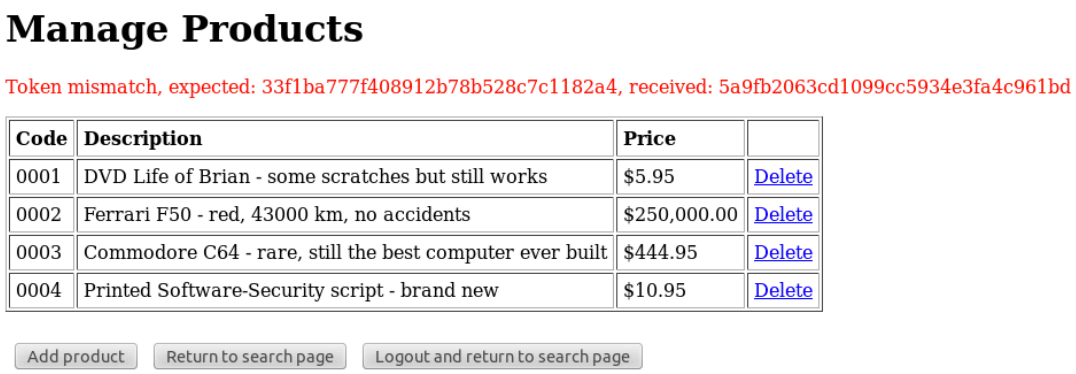
[1] A literal dot (.) in a regex must be escaped with \, so \. is usually written in a regex. The regex specified here is used in the program as a Java string. In Java however, the backslash character also has a special meaning to escape some special characters. In particular, a \ must be written as \\ in a Java string to be interpreted as a literal backslash. Therefore, \\. must be used in the regex to make sure it is actually interpreted as a literal dot (.).

- Are the characters that should be allowed really accepted?
- Are the borderline cases working correctly? E.g. when entering 9 or 101 characters in the description? Or when using a price of 1'000'000?
- Are inputs with disallowed characters correctly declined?

4.7 CSRF protection

Finally, you should protect the actions „Delete Product" und „Store Product" from CSRF attacks. Use the approach with the CSRF token we have introduced in the lecture and which is already used to protect a part of the application („Complete Purchase").

If the received token is not equal to the one stored in the session, the application should behave in the same way as it is already the case with „Complete Purchase": The action is not performed and both the expected and the received token should be printed, according to the following screenshot:



Test whether CSRF attacks are indeed detected by deleting or modifying the sent CSRF token.

4.8 Final tests

If you have reached this point and have implemented and tested everything correctly, you should be done. However, it is important to make one final, complete test because so far, you have focused on individual problems and it may be that you accidentally "unfixed" something you have solved before. It is therefore reasonable that you repeat the tests of all individual tasks you have solved, although some tests – e.g. entering a JavaScript – hopefully won't be possible any more. During this final test, think about functionality *and* security, i.e.:

- Functionality: Does everything work that should work? Are users allowed to access the functions they should be able to access according to their role? The idea is to make sure that you haven't introduces too many restrictions by introducing all theses security features.
- Security: Is everything prevented that should be prevented? Are the users really prevented from accessing the functions of other roles, also with forced browsing? Are the sensitive areas really only reachable via HTTPS? Are input validation and CSRF protection working correctly?

In addition, perform automatic security tests using Fortify SCA and Arachni, in exactly the same way as you have done in a previous lab. Compare the results with the ones observed during that lab when testing Marketplace V10. Does any of the tools report additional vulnerabilities (that are no false positives) in the range *Medium – Critical* you should get rid of? If yes, correct your code.

Lab Points

For **4 Lab Points**, you must demonstrate to the instructor that your Marketplace application runs according to the specifications:

- You get 1 point if the users get access only to the functions they are allowed to access according to their role. In addition, access is only possible via HTTPS and even forced browsing does not allow illegitimate access (tasks 4.2 and 4.3).
- You get a 2nd point if database access works correctly and uses prepared statements and if data sanitation also works correctly (tasks 4.4 and 4.5).
- You get the 3rd point if input validation and CSRF protection work correctly (tasks 4.6 and 4.7).
- You get the 4th point if the outputs of Fortify SCA and Arachni don't show additional vulnerabilities compared to Marketplace V10 (task 4.8).