****

**Web Based Application Design**

Due Date: Friday the 22sd of April

Student: Hugh Mullins, Student ID: 16106989

Student: Sunday Jimoh, Student ID: 16090462

Student: Conor McDonnel, Student ID: 16120604

Student: Aaron Dias, Student ID: 16068963

Student: Samaa Alsafwani, Student ID: 16087941

Course: Information and Network Security MENG

Department: Electronic & Computer Engineering

Table of Contents

Table of Contents 2

List of Figures 3

1 OWASP Top 10 Defences - Techniques 4

1.1 Injection 4

1.2 Session Management 4

1.3 Cross-Site Scripting (XSS) 4

1.4 Missing Function Level Access Control 4

1.5 Cross-Site Request Forgery (CSRF) 4

1.6 Failure to Restrict URL Access 5

1.7 Insecure Direct Object References 5

2 Testing 6

2.1 Introduction 6

2.2 Conclusion 6

2.2.1 Issues Encountered 6

2.2.2 HSV saturation 6

2.2.3 Data input to Euclidean algorithm 6

List of Figures

[Figure 1. Code added to stop URL rewriting 7](#_Toc479539748)

[Figure 2. Before and After code Snippet 7](#_Toc479539749)

[Figure 3. The test for XSS 7](#_Toc479539750)

[Figure 4. The result for XSS test 8](#_Toc479539751)

[Figure 5. Admin Access regular user 8](#_Toc479539752)

[Figure 6. Access Denied Page 9](#_Toc479539753)

[Figure 7. Admin Access regular user admin user 9](#_Toc479539754)

[Figure 8. Admin Panel Page 10](#_Toc479539755)

[Figure 9. Unique hidden token code 10](#_Toc479539756)

[Figure 10. Unedited Action 11](#_Toc479539757)

[Figure 11. Edited Action 11](#_Toc479539758)

[Figure 12. Returns Error Page 12](#_Toc479539759)

[Figure 13. Access Denied Page 12](#_Toc479539760)

[Figure 14. Admin Panel Page 13](#_Toc479539761)

[Figure 15. Access Denied Page 13](#_Toc479539762)

[Figure 16. Example URLs 13](#_Toc479539763)

# OWASP Top 10 Defences - Techniques

## Injection

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.

## Session Management

Running OWASP ZAP against the application showed that our application was vulnerable to URL rewriting. Adding a session timeout and setting the tracking mode to cookie in the “web.xml” file stopped ZAP from detecting this by forcing the session to be tracked in a cookie and with a timeout value.

## Cross-Site Scripting (XSS)

We used OWASP anti-samy to clean input from the user views in the login as well as the search fields. Any updates to the user profile that are added are also cleaned by anti-samy before being stored in the database.

When a user enters a username the name is filtered for reflexive XSS attacks from the name being displayed in an error.

## Missing Function Level Access Control

Each time a controller admin function is called via a URL pattern, the user role is checked before executing these functions. If they are not authorized the controllers redirect them automatically.

## Cross-Site Request Forgery (CSRF)

A CSRF attacks are based on a user being logged in. An attacker will send forged HTTP requests to another site with that users session information and cookies as well as other authenticated and sensitive information.

The OWASP organization recommends a number of things to include in your web application to prevent and mitigate against this vulnerability. The most common and optimal option is to add a hidden HTML field that is sent in the HTTP request and then validating this on the server side. If the token matches the one stored on the server, the request can be carried out. If the token does not match or is missing then the request is not carried out. A similar option for this is for the token to be displayed in the URL but this is riskier since the URL and token are exposed to the attacker. Using the token method we were able to secure our application against cross-site request forgery.

## Failure to Restrict URL Access

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 an administrator user type and a customer user type. 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.

## Insecure Direct Object References

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.

# Testing

## Injection

We setup a ZAP proxy and captured the initial post request for login to the application and then used this with SQLmap to attempt to attack the server with injection attempts.

***sqlmap -r post.txt http://localhost:8080/GROUP16ShopApp-war/faces/login.xhtml***

*[15:19:07] [WARNING] POST parameter 'login' is not injectable*

*[15:19:56] [WARNING] POST parameter 'login:username' is not injectable*

*[15:20:41] [WARNING] POST parameter 'login:password' is not injectable*

*[15:21:26] [WARNING] POST parameter 'login:enter' is not injectable*

*[15:22:03] [WARNING] POST parameter 'javax.faces.ViewState' is not injectable*

*[15:23:03] [CRITICAL] all tested parameters appear to be not injectable.*

From SearchPage page, captured a search post header and again ran SQLmap against the application.

[16:00:03] [WARNING] POST parameter 'productName' is not injectable

[16:01:12] [WARNING] POST parameter 'prodcutId' is not injectable

[16:02:03] [CRITICAL] all tested parameters appear to be not injectable.

From srchCstmrPage page, captured a search post header and again ran SQLmap against the application.

[16:00:03] [WARNING] POST parameter 'username' is not injectable

[16:01:12] [WARNING] POST parameter 'uid' is not injectable

[16:02:03] [CRITICAL] all tested parameters appear to be not injectable.

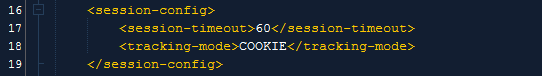
## Session Management

Below are 2 consecutive session ids captured via ZAP, they do not appear to be predicable.

JSESSIONID=344ce6e1ae97dfb5217b6a43adb3

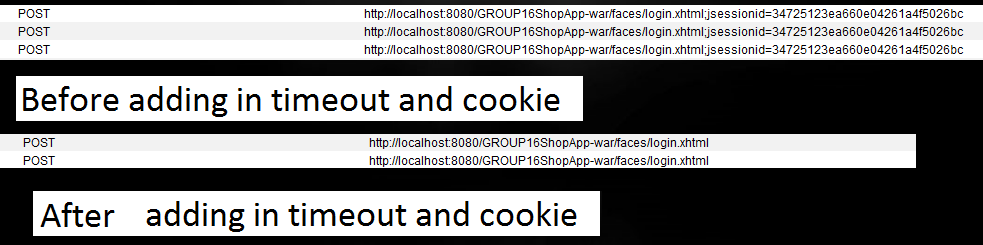
JSESSIONID=44725123ea660e04261a4f5026bc

Below in figure 1, you can see the code that was added so as to stop URL rewriting.



1. Code added to stop URL rewriting

By Adding this session timeout and setting the tracking mode to cookie in the “web.xml” file stopped OWASP ZAP from detecting this by forcing the session to be tracked in a cookie and with a timeout value. As Seen in Figure 2.



1. Before and After code Snippet

## Cross-Site Scripting (XSS)

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



1. The test for XSS



1. The result for XSS test

As is seen in the test the “**<script> alert("XSS attack!!!") </script>**” has been removed from the message.

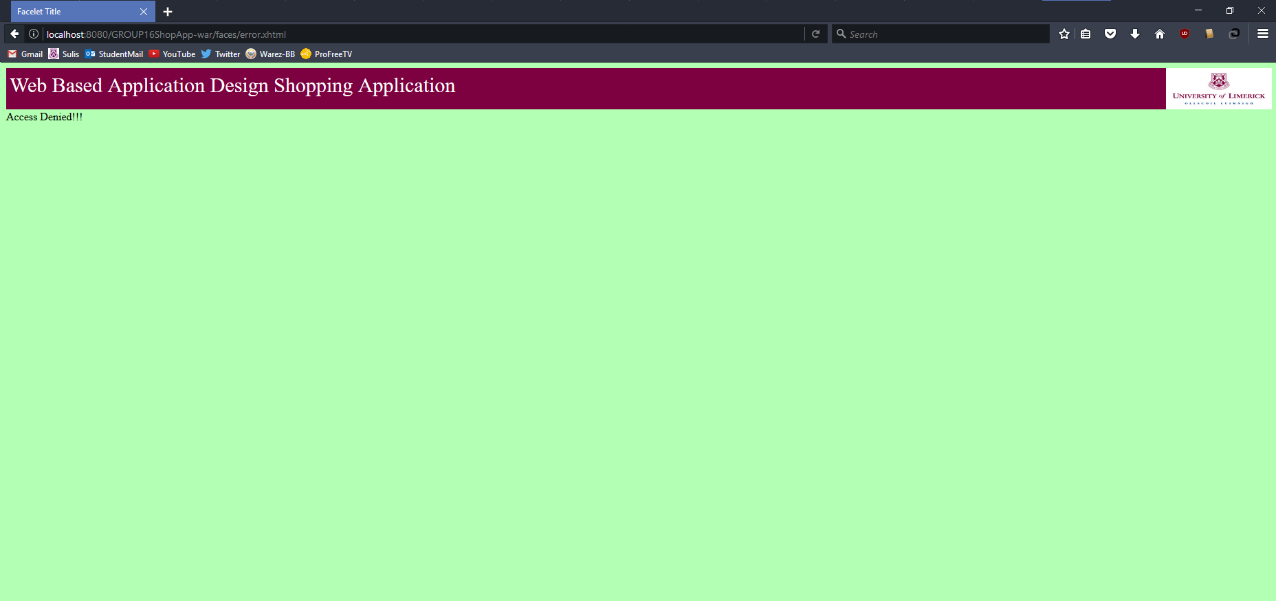
## Missing Function Level Access Control

Joe is a regular user when Joe attempts to access the admin Panel using the admin panel button shown in figure 5 below.



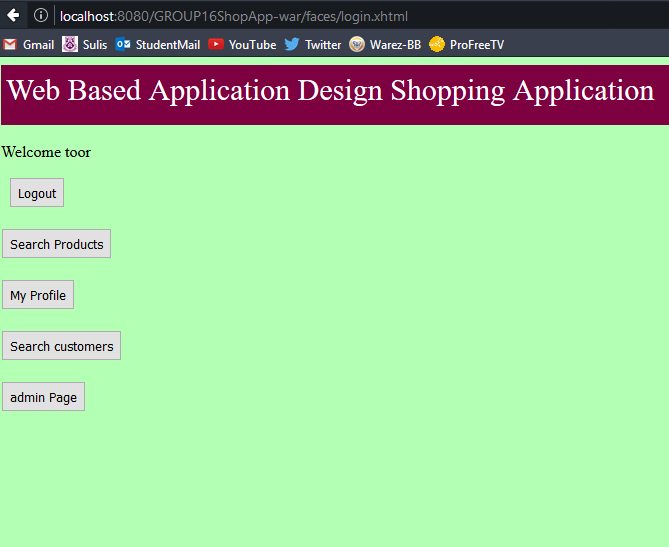
1. Admin Access regular user

He will be redirected to an error page telling him that access is denied Figure 6 below.



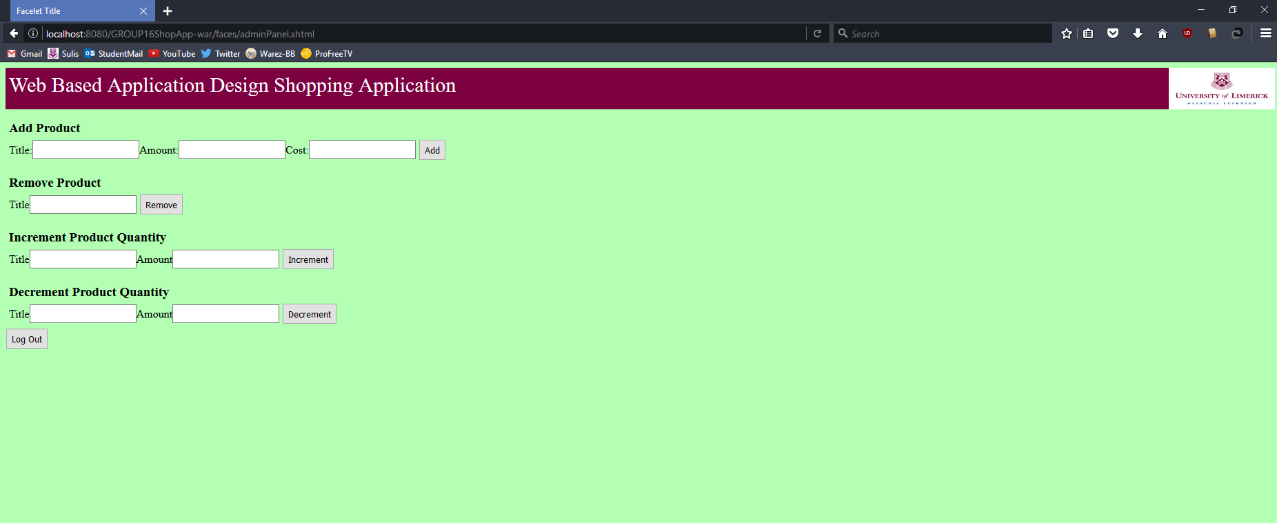
1. Access Denied Page

The account toor is an admin so when he attempts to access the admin panel using the admin panel button shown in figure 7 below.



1. Admin Access regular user admin user

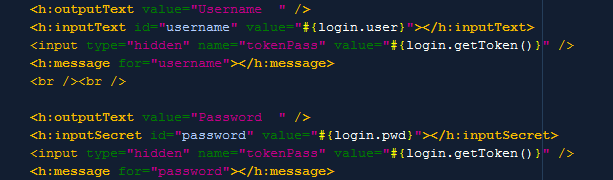
He will be redirected to the admin panel page Figure 8 below.



1. Admin Panel Page

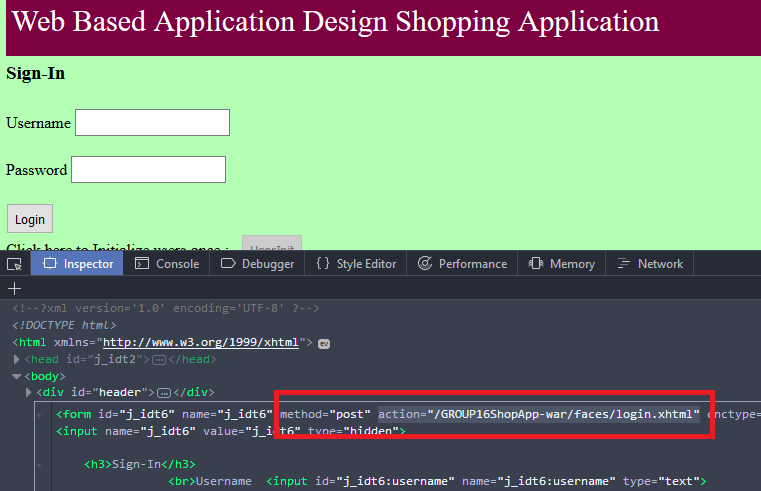
## Cross-Site Request Forgery (CSRF)

CSRF can be avoided by creating a unique token in a hidden field which would be sent in the body of the HTTP request rather than in an URL, which is more prone to exposure. The setup of our unique hidden token present during login can be seen in Figure 9 below:

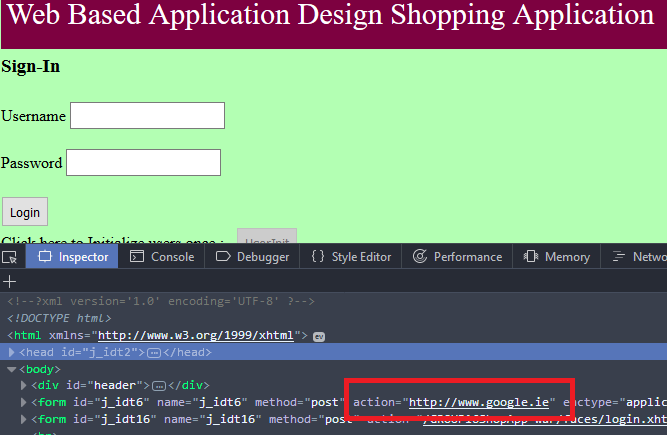


1. Unique hidden token code

If an action like seen in figure 10 is edited to something else like seen in Figure 11 below.

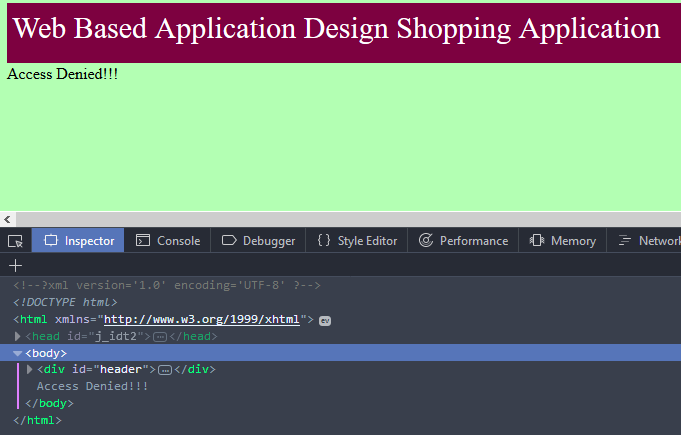


1. Unedited Action



1. Edited Action

Upon clicking the Login button now the user is return to the Access denied error page. As seen in Figure 12 below.



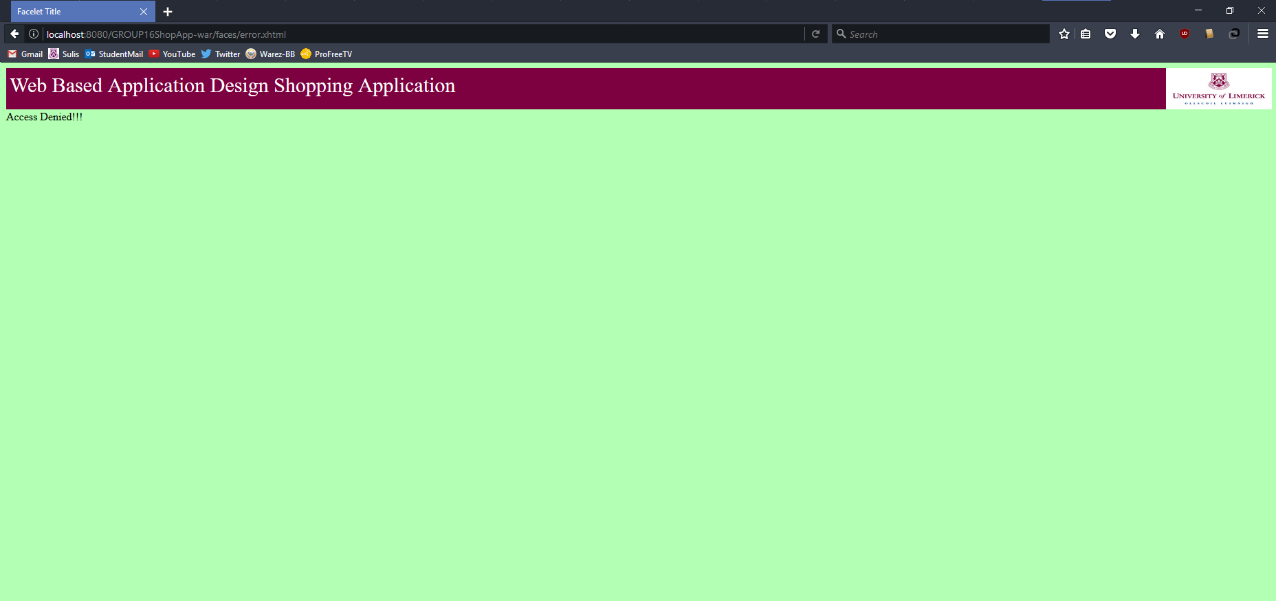
1. Returns Error Page

## Failure to restrict URL Access

Joe is a regular user when Joe steals the following web address, but he doesn’t have the account and password.

**“http://localhost:8080/GROUP16ShopApp-war/faces/adminPanel.xhtml”**

He will be redirected to an error page telling him that access is denied Figure 13 below.

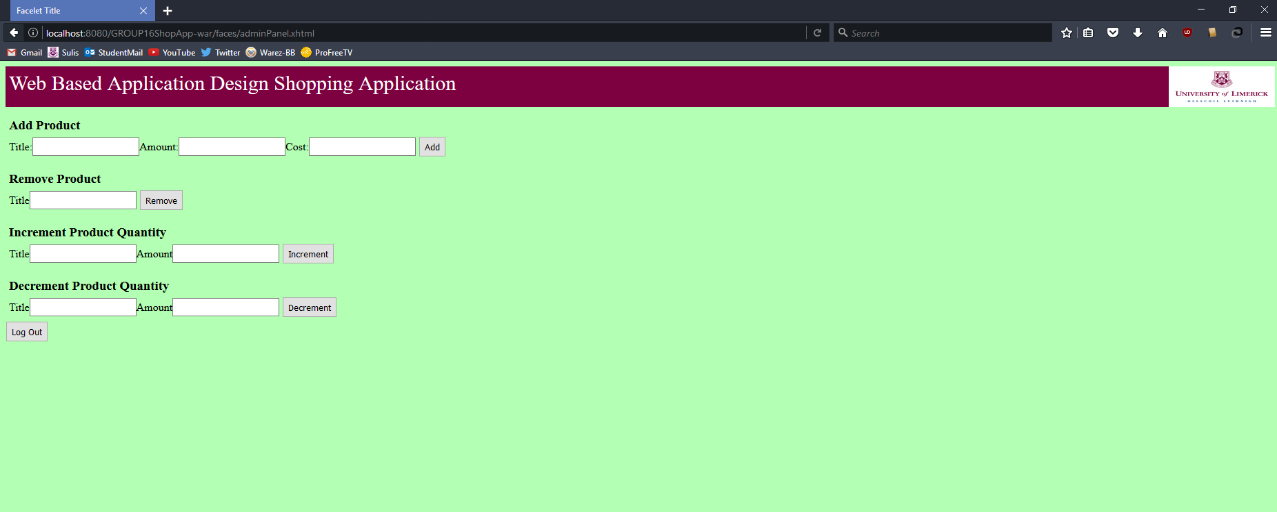


1. Access Denied Page

The account toor is an admin so when he attempts to access the admin Panel using the link:

**“http://localhost:8080/GROUP16ShopApp-war/faces/adminPanel.xhtml”**

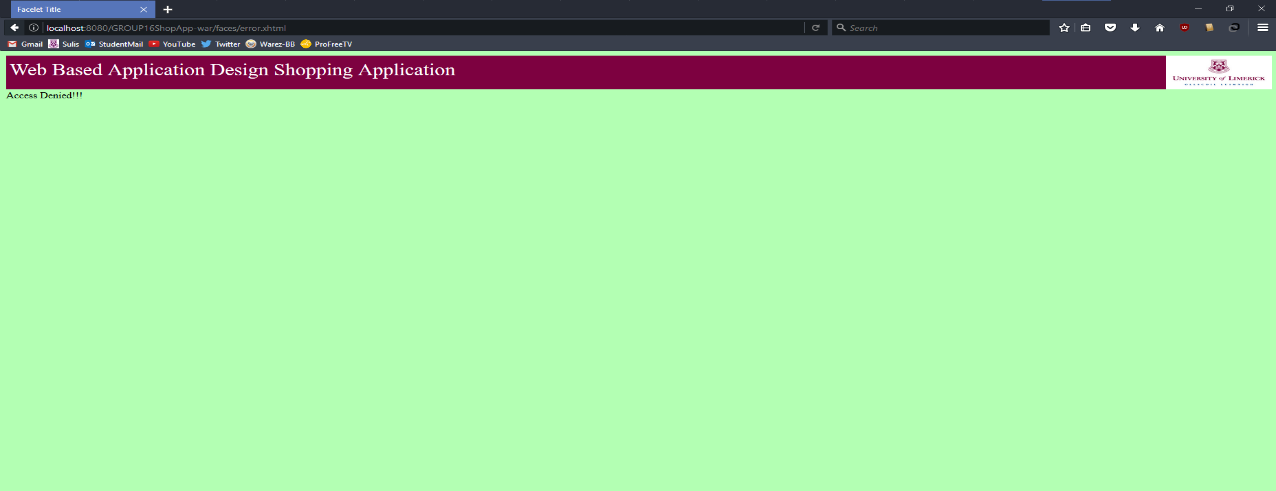
He will be redirected to the admin panel page Figure 14 below.



1. Admin Panel Page

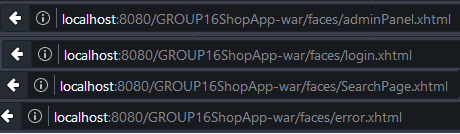
## Insecure Direct Object References

Our web application will check the customer whether have the right to visit the servlet. If not they will be presented with Figure 15:



1. Access Denied Page

Our URL do not use direct object references this can be seen in Figure 16 below:



1. Example URLs