WEB APPLICATION SECURITY & VULNERABILITY ASSESSMENT DEMONSTRATION

DAVID ODZA

TABLE OF CONTENTS

Table of Contents 2

Target Introduction 3

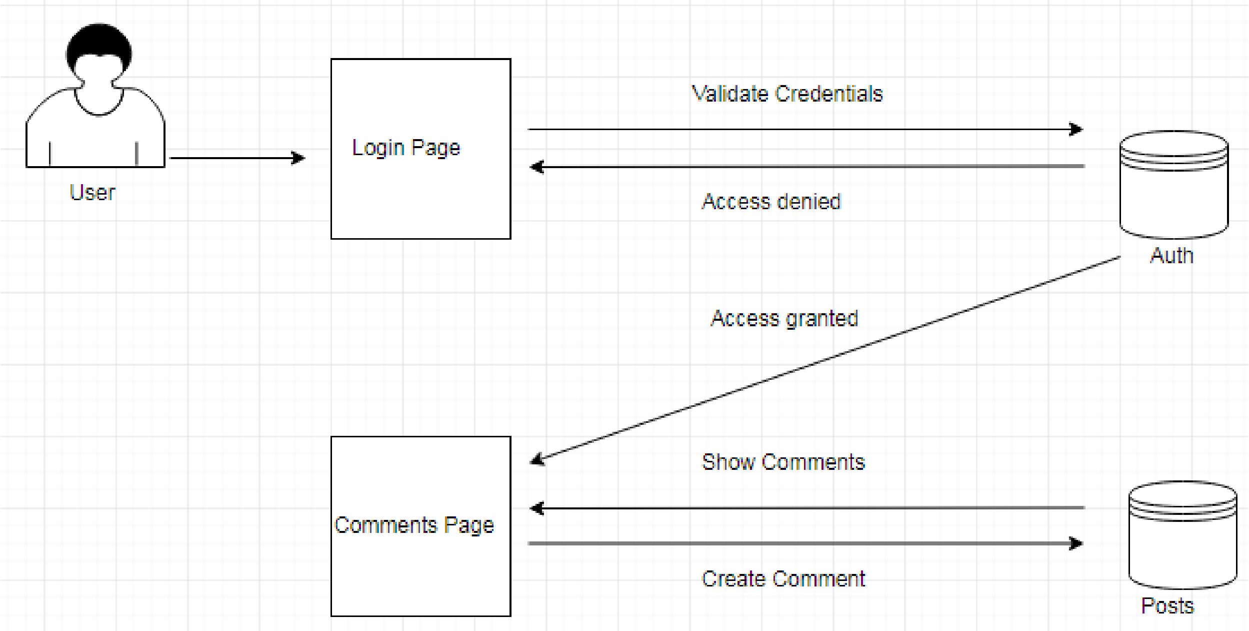
Cross Site Scripting 6

SQL Injection 9

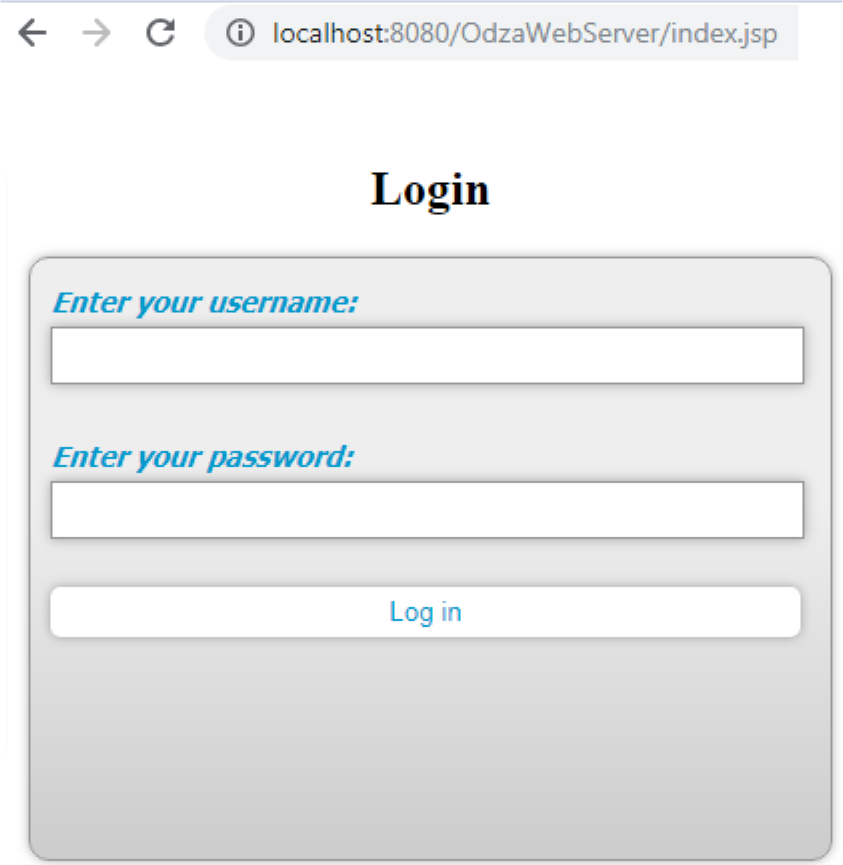
Username Harvesting & Password Attacks 13

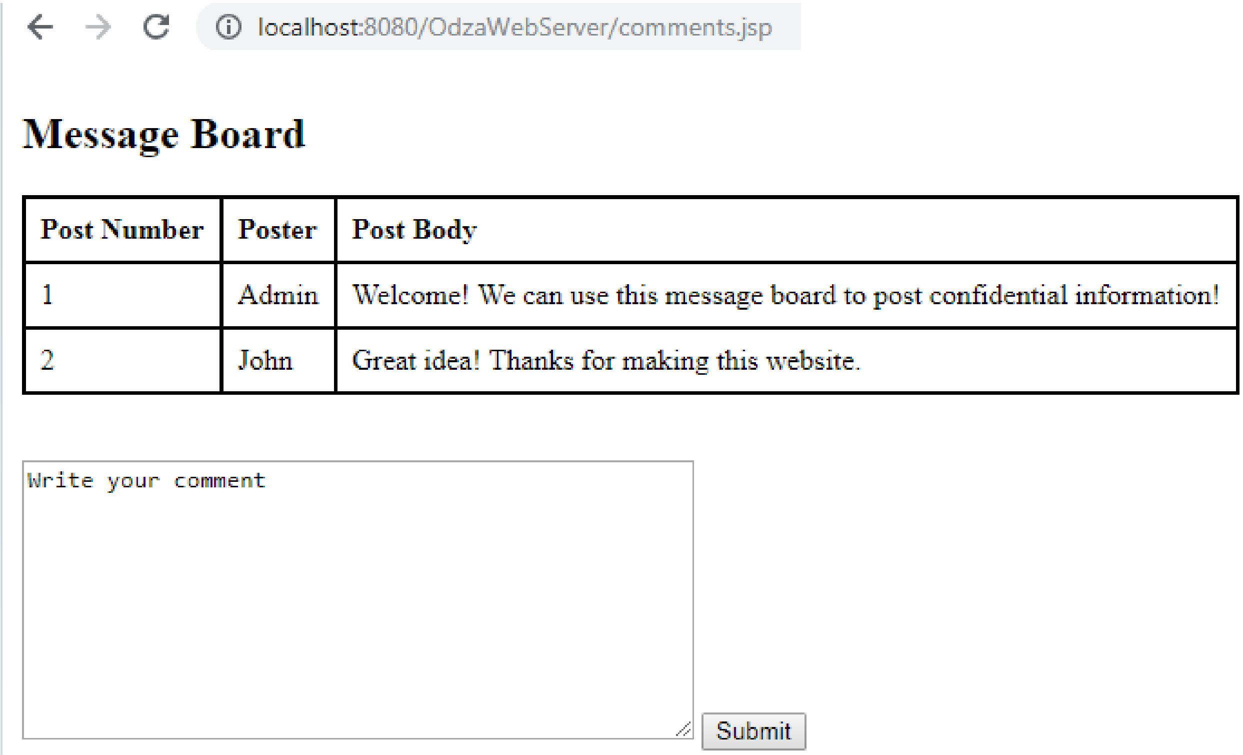
TARGET INTRODUCTION

The target of this security & vulnerability assessment is a web application called “Simple Login and Comments Board” created by David Odza. The target web application uses the Model, View, Controller architectural pattern. The model is a MySQL schema, the views are JSP pages, and the controllers are Java classes.



The application has two views: login page and message board page.



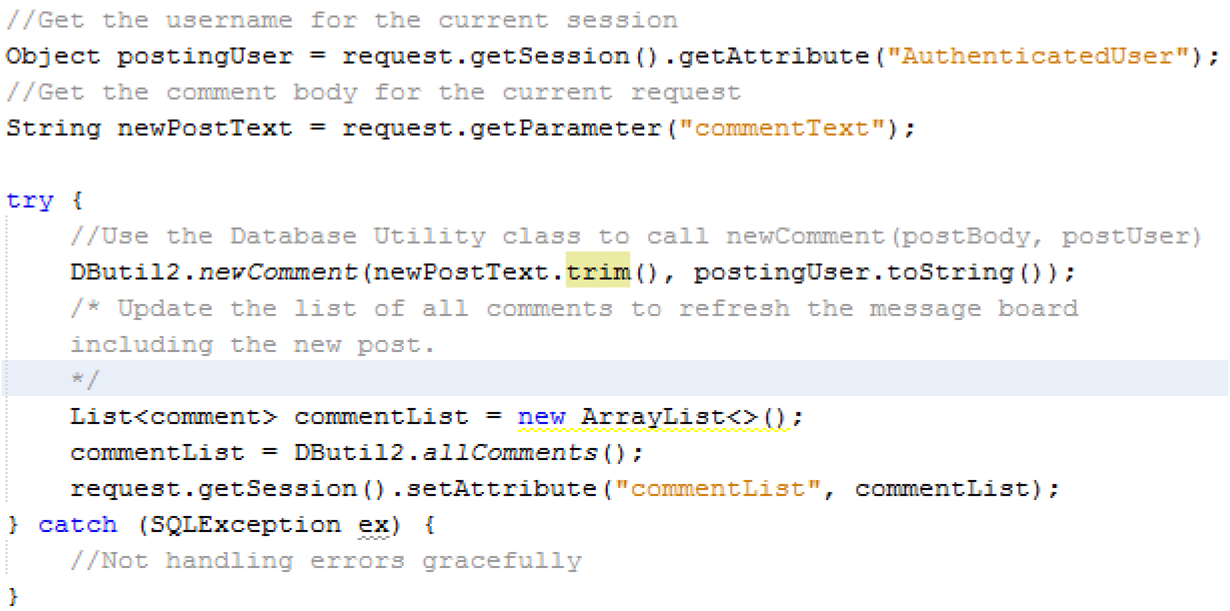


Message Board

Post Number	Poster	Post Body
1	Admin	Welcome! We can use this message board to post confidential information!
2	John	Great idea! Thanks for making this website.

Usernames and passwords are stored in the 'auth' database, while comment poster and comment body are stored in the 'posts' database.

Below is an example of the java controller source code which provides the business logic to the application.



```java
//Get the username for the current session
Object postingUser = request.getSession().getAttribute("AuthenticatedUser");
//Get the comment body for the current request
String newPostText = request.getParameter("commentText");

try {
    //Use the Database Utility class to call newComment(postBody, postUser)
    DButil2.newComment(newPostText.trim(), postingUser.toString());
    /* Update the list of all comments to refresh the message board
    including the new post.
    */
    List<comment> commentList = new ArrayList<>();
    commentList = DButil2.allComments();
    request.getSession().setAttribute("commentList", commentList);
} catch (SQLException ex) {
    //Not handling errors gracefully
}
```

CROSS SITE SCRIPTING

Severity
High

Application Vulnerable to Stored Cross Site Scripting on Comments Page

RESOURCE(S)

http://10.0.0.11:8080/OdzaWebServer/comments.jsp

DESCRIPTION

Likelihood
Medium

Impact
High

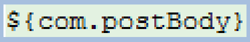
The application has a comments page where users post and read comments. User comments are stored in a MySQL database. The comments page queries the database and displays all user comments. The application does not use any form of input validation or encoding. As a result, the application contains a stored cross-site scripting vulnerability in the comments page.

The following proof-of-concept example can be placed in a comment post to carry out an attack. Note that this simple example requires the end-user to view the comments page. A real-world attack would use malicious JavaScript.

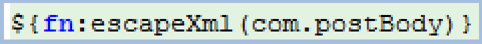
Example: The following link will display an alert dialog containing the text ‘XSS vulnerability’
Affected Browsers: Internet Explorer, FireFox, Chrome
Attack: <script>alert("XSS vulnerability")</script>

RECOMMENDED SOLUTION

The recommended solution for this XSS vulnerability is to escape user provided input when displaying it. This can be achieved with the fn:escapeXml() function. This should be implemented everywhere that user-controlled input is displayed.

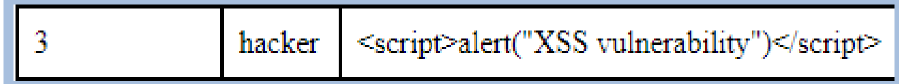
Current Source Code Example: 

```
${com.postBody}
```

Recommended Source Code Example: 

```
${fn:escapeXml(com.postBody)}
```

Result: The attack is displayed as text instead of executed as JavaScript.



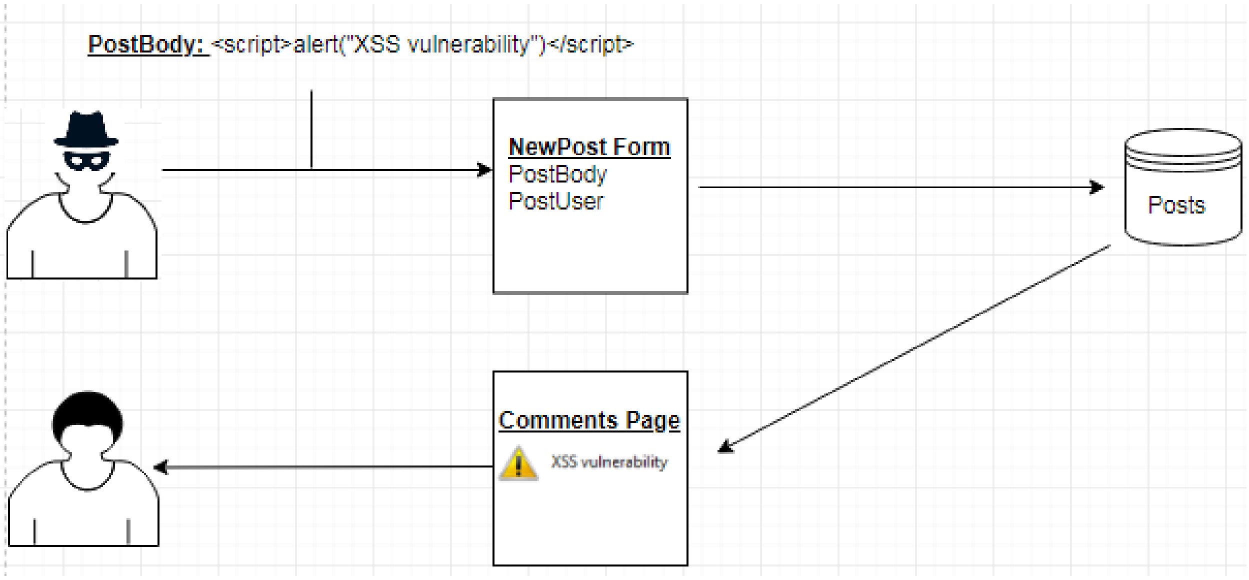
3	hacker	<script>alert("XSS vulnerability")</script>

Additional Recommendations

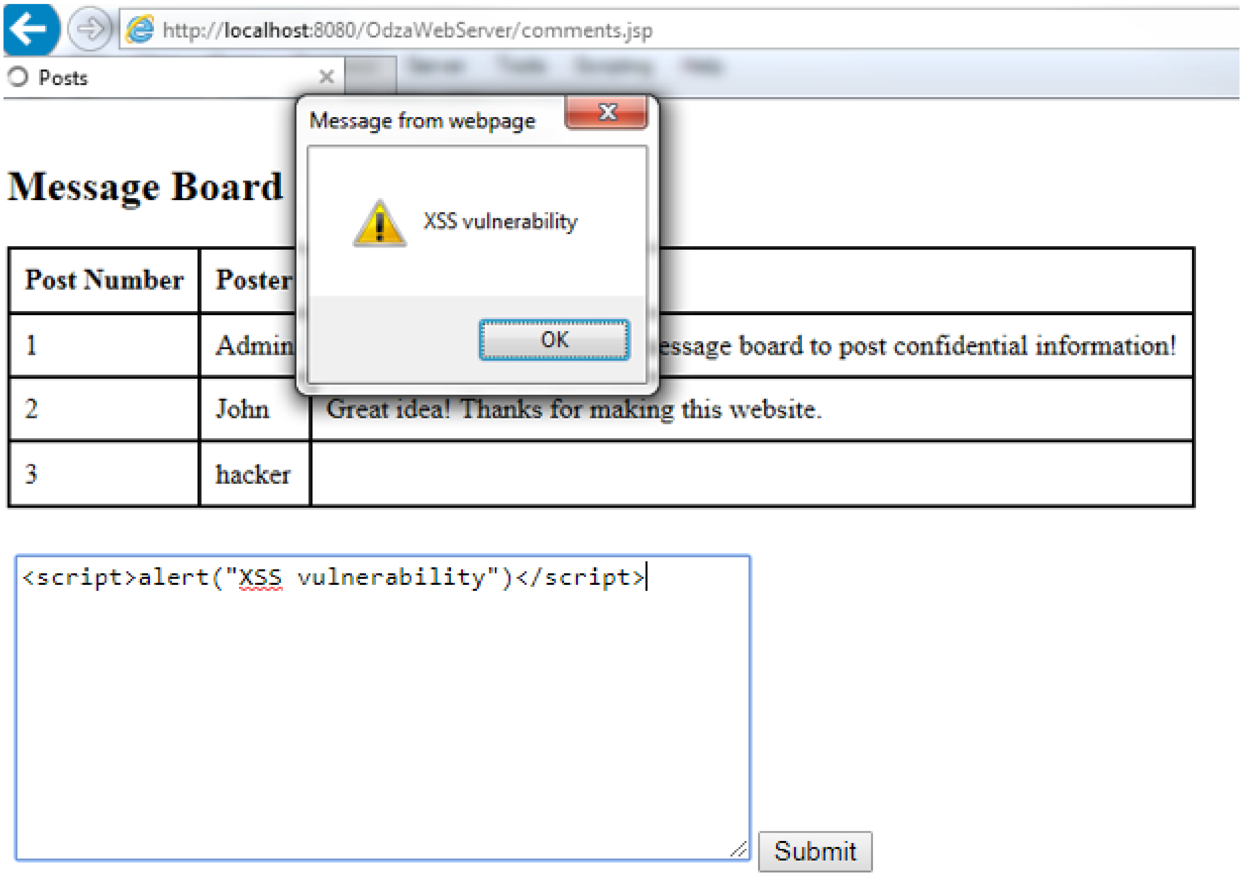
If allowing user input HTML into the application is required, it’s recommended to use the OWASP Java HTML Sanitizer Project. This HTML Sanitizer lets you include HTML authored by third-parties in your web application while protecting against XSS.

It’s also recommended to validate user-provided input with a whitelist. This should be implemented by defining permitted characters and denying user input containing any characters that aren’t explicitly permitted. For instance, <script> can be omitted from the whitelist.

REFERENCE(S)	https://www.ibm.com/support/knowledgecenter/SSZLC2_9.0.0/com.ibm.commerce.developer.doc/refs/rsdjspbpescapexml.htm https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project



Screenshot of vulnerabillity exploitation



XSS vulnerability

Message Board

Post Number	Poster	
1	Admin	essage board to post confidential information!
2	John	Great idea! Thanks for making this website.
3	hacker	

```
<script>alert("XSS vulnerability")</script>
```

SQL INJECTION

Severity
High

Application Vulnerable to SQL Injection on Login Page

RESOURCE(S)

http://10.0.0.11:8080/OdzaWebServer/index.jsp

DESCRIPTION

Likelihood
Medium

Impact
High

The application has a login page where users authenticate with username and password. The application makes queries to the MySQL database to verify user provided credentials. User provided input is concatenated directly to the database queries and executed. As a result, the application contains a SQL injection vulnerability in the login page.

The following proof-of-concept examples can be placed in the username field of the login page.

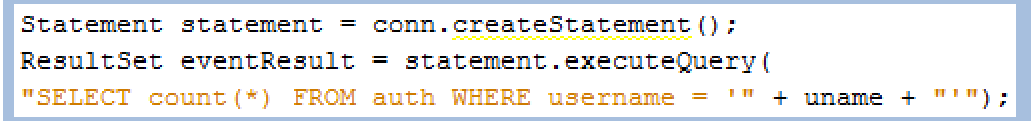
Example #1: The following SQL injection will change "Admin" password to "hacked" in the auth database.
Affected Browsers: Internet Explorer, FireFox, Chrome
Attack: anything'; UPDATE auth SET password = 'hacked' WHERE username = 'Admin';#

Example #2: The following SQL injection will insert a new user into the auth database.
Affected Browsers: Internet Explorer, FireFox, Chrome
Attack: anything'; INSERT into auth (username, password) VALUES ('hacker', 'muahaha');#

RECOMMENDED SOLUTION

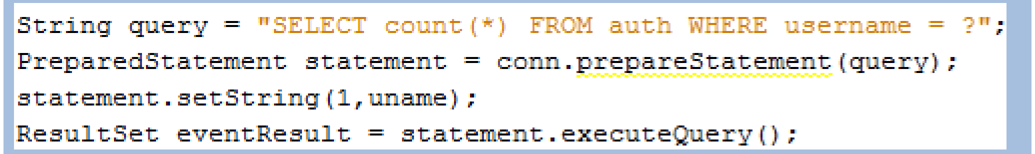
The recommended solution for this SQL injection vulnerability is to use PreparedStatement instead of Statement. This should be implemented using the PreparedStatement setter methods. This solution should be applied to all SQL statements that contain user provided input.

Current Source Code Example:



```
Statement statement = conn.createStatement();
ResultSet eventResult = statement.executeQuery(
"SELECT count(*) FROM auth WHERE username = '" + uname + "'");
```

Recommended Source Code Example:



```
String query = "SELECT count(*) FROM auth WHERE username = ?";
PreparedStatement statement = conn.prepareStatement(query);
statement.setString(1,uname);
ResultSet eventResult = statement.executeQuery();
```

Result: The user input is correctly treated as data. The malicious 2nd query is not executed.

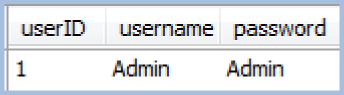
Additional Recommendations

Input Validation: It’s recommended to validate user-provided input with a whitelist. This should be implemented by defining permitted characters and denying user input containing any characters that aren’t explicitly permitted. For instance, the character “;” can be omitted from the whitelist. Only if the user provides input with all characters on the whitelist, should it be sent to the database utility and query executed.

Least Privilege: To reduce the impact of a successful SQL injection attack, it’s recommended to apply the concept of Least Privilege to the MySQL database from the application. The connection from the application to the MySQL database should only have permissions to execute the required queries and nothing further. For instance, the utility that validates the username authenticity has no business reason to be dropping tables or inserting users.

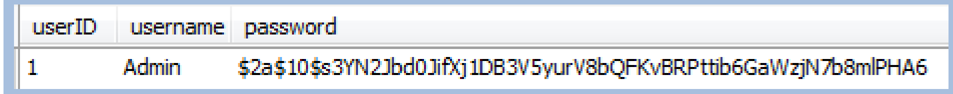
Password Storage: To reduce the impact of a successful SQL injection attack, it’s recommended to store passwords as randomly-salted hashes. An option for this is Java BCrypt. Below I’ve provided an example of password storage using BCrypt to generate a randomly-salted hash.

Current password storage (clear text):



userID	username	password
1	Admin	Admin

Recommended password storage (randomly-salted hash):



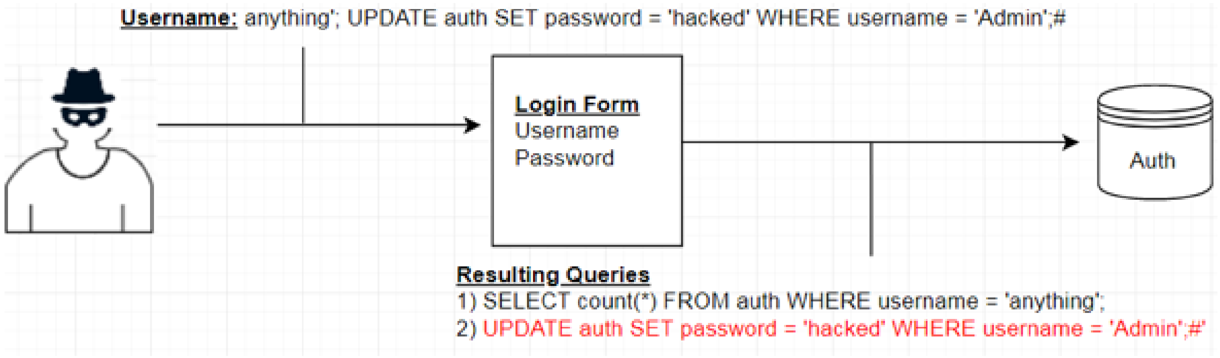
userID	username	password
1	Admin	$2a$10$s3YN2Jbd0JifXj1DB3V5yurV8bQFKvBRPttib6GaWzjN7b8mlPHA6

REFERENCE(S)

https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet

https://docs.oracle.com/javase/6/docs/api/java/sql/PreparedStatement.html

https://www.mindrot.org/projects/jBCrypt/



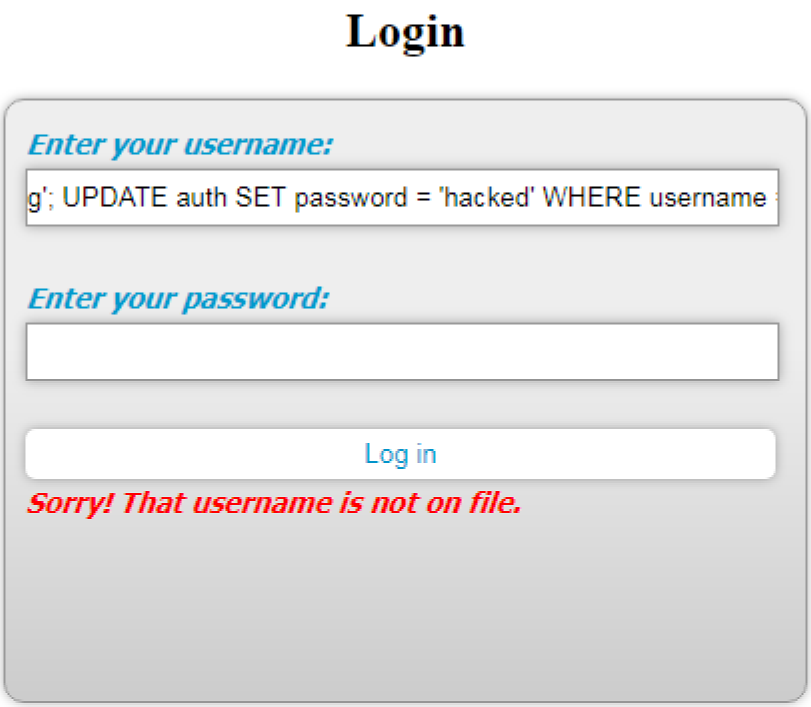
Username: anything'; UPDATE auth SET password = 'hacked' WHERE username = 'Admin';#

Resulting Queries

1) SELECT count(*) FROM auth WHERE username = 'anything';

2) UPDATE auth SET password = 'hacked' WHERE username = 'Admin';#'

Screenshot of vulnerabillity exploitation



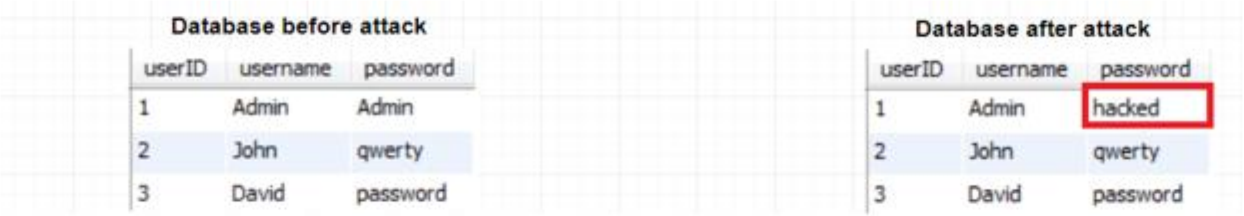
Login

Enter your username:

g'; UPDATE auth SET password = 'hacked' WHERE username

Enter your password:

Sorry! That username is not on file.



Database before attack

userID	username	password
1	Admin	Admin
2	John	qwerty
3	David	password

Database after attack

userID	username	password
1	Admin	hacked
2	John	qwerty
3	David	password

USERNAME HARVESTING AND PASSWORD ATTACKS

Severity
Medium

Application Vulnerable to Username Harvesting and Password Attacks

RESOURCE(S)

http://10.0.0.11:8080/OdzaWebServer/index.jsp

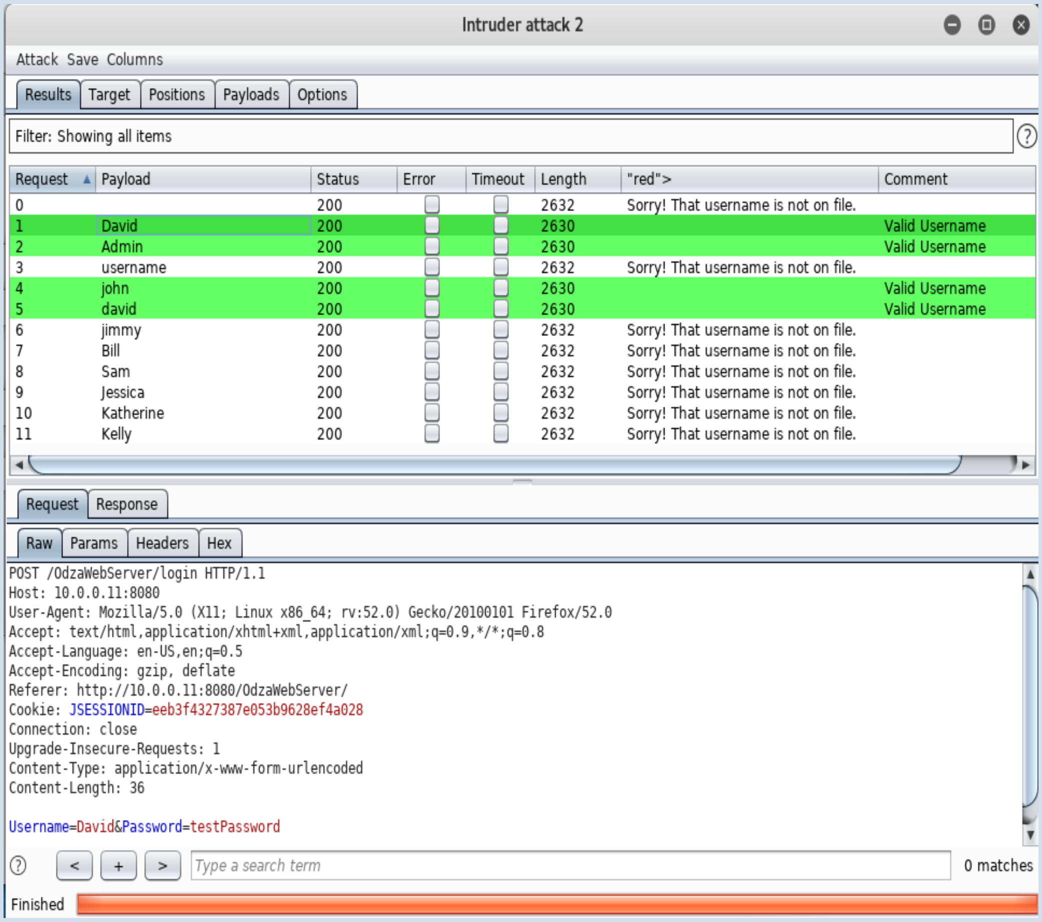
DESCRIPTION

Likelihood
Medium

Impact
Medium

The application responds differently to the user whether they input an incorrect username or incorrect password. This vulnerability allows an attacker to harvest usernames, which is to collect a list of valid usernames by testing them and noting the applications response.

Username Harvesting: The following proof-of-concept example shows how an attacker using Burp Suite can harvest usernames. The below attack has determined that “David”, “Admin”, “john” and “david” are valid usernames. Attackers use large lists of common usernames to harvest usernames.



Request	Payload	Status	Error	Timeout	Length	"red">	Comment
0		200	□	□	2632	Sorry! That username is not on file.	
1	David	200	□	□	2630		Valid Username
2	Admin	200	□	□	2630		Valid Username
3	username	200	□	□	2632	Sorry! That username is not on file.	
4	john	200	□	□	2630		Valid Username
5	david	200	□	□	2630		Valid Username
6	jimmy	200	□	□	2632	Sorry! That username is not on file.	
7	Bill	200	□	□	2632	Sorry! That username is not on file.	
8	Sam	200	□	□	2632	Sorry! That username is not on file.	
9	Jessica	200	□	□	2632	Sorry! That username is not on file.	
10	Katherine	200	□	□	2632	Sorry! That username is not on file.	
11	Kelly	200	□	□	2632	Sorry! That username is not on file.	

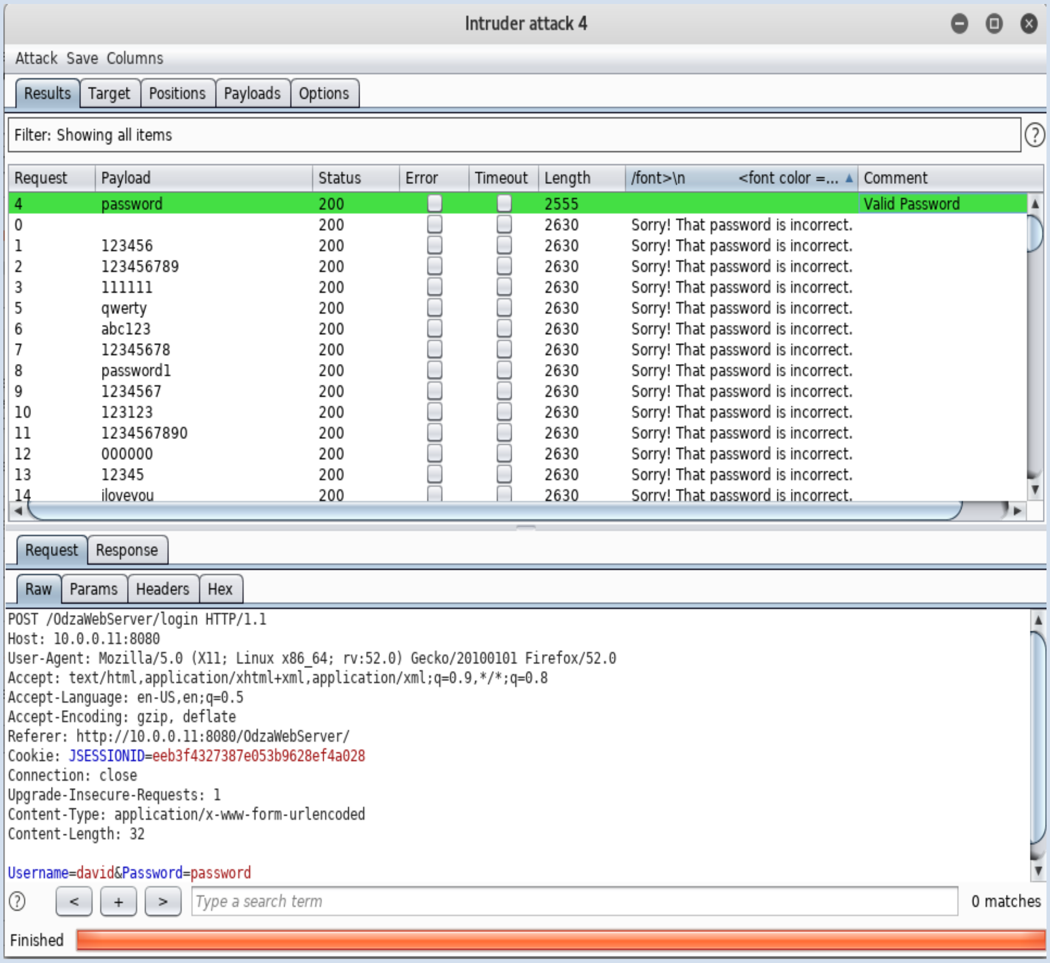
```
POST /OdzaWebServer/login HTTP/1.1
Host: 10.0.0.11:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://10.0.0.11:8080/OdzaWebServer/
Cookie: JSESSIONID=eeb3f4327387e053b9628ef4a028
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 36

Username=David&Password=testPassword
```

The next step for an attacker who has harvested usernames is a password attack. The application is vulnerable to password attacks against individual users because there is no threshold of failed logins that trigger an account lockout.

Password Dictionary Attack: From the previous proof-of-concept I learned that “david” is a valid username. The next proof-of-concept will show how an attacker could get david’s password.

I loaded Burp Suite with the with a list of 10,000 commonly used passwords. After a few minutes the 10,000 passwords were checked against the password for “david” and it found a match. The password for “david” is “password”.



Filter: Showing all items

Request	Payload	Status	Error	Timeout	Length	/font>\n <font color =... ▲	Comment
4	password	200	☐	☐	2555		Valid Password
0		200	☐	☐	2630	Sorry! That password is incorrect.	
1	123456	200	☐	☐	2630	Sorry! That password is incorrect.	
2	123456789	200	☐	☐	2630	Sorry! That password is incorrect.	
3	111111	200	☐	☐	2630	Sorry! That password is incorrect.	
5	qwerty	200	☐	☐	2630	Sorry! That password is incorrect.	
6	abc123	200	☐	☐	2630	Sorry! That password is incorrect.	
7	12345678	200	☐	☐	2630	Sorry! That password is incorrect.	
8	password1	200	☐	☐	2630	Sorry! That password is incorrect.	
9	1234567	200	☐	☐	2630	Sorry! That password is incorrect.	
10	123123	200	☐	☐	2630	Sorry! That password is incorrect.	
11	1234567890	200	☐	☐	2630	Sorry! That password is incorrect.	
12	000000	200	☐	☐	2630	Sorry! That password is incorrect.	
13	12345	200	☐	☐	2630	Sorry! That password is incorrect.	
14	iloveyou	200	☐	☐	2630	Sorry! That password is incorrect.	

```
POST /OdzaWebServer/login HTTP/1.1
Host: 10.0.0.11:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:52.0) Gecko/20100101 Firefox/52.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://10.0.0.11:8080/OdzaWebServer/
Cookie: JSESSIONID=eeb3f4327387e053b9628ef4a028
Connection: close
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
Content-Length: 32

Username=david&Password=password
```

RECOMMENDED SOLUTION

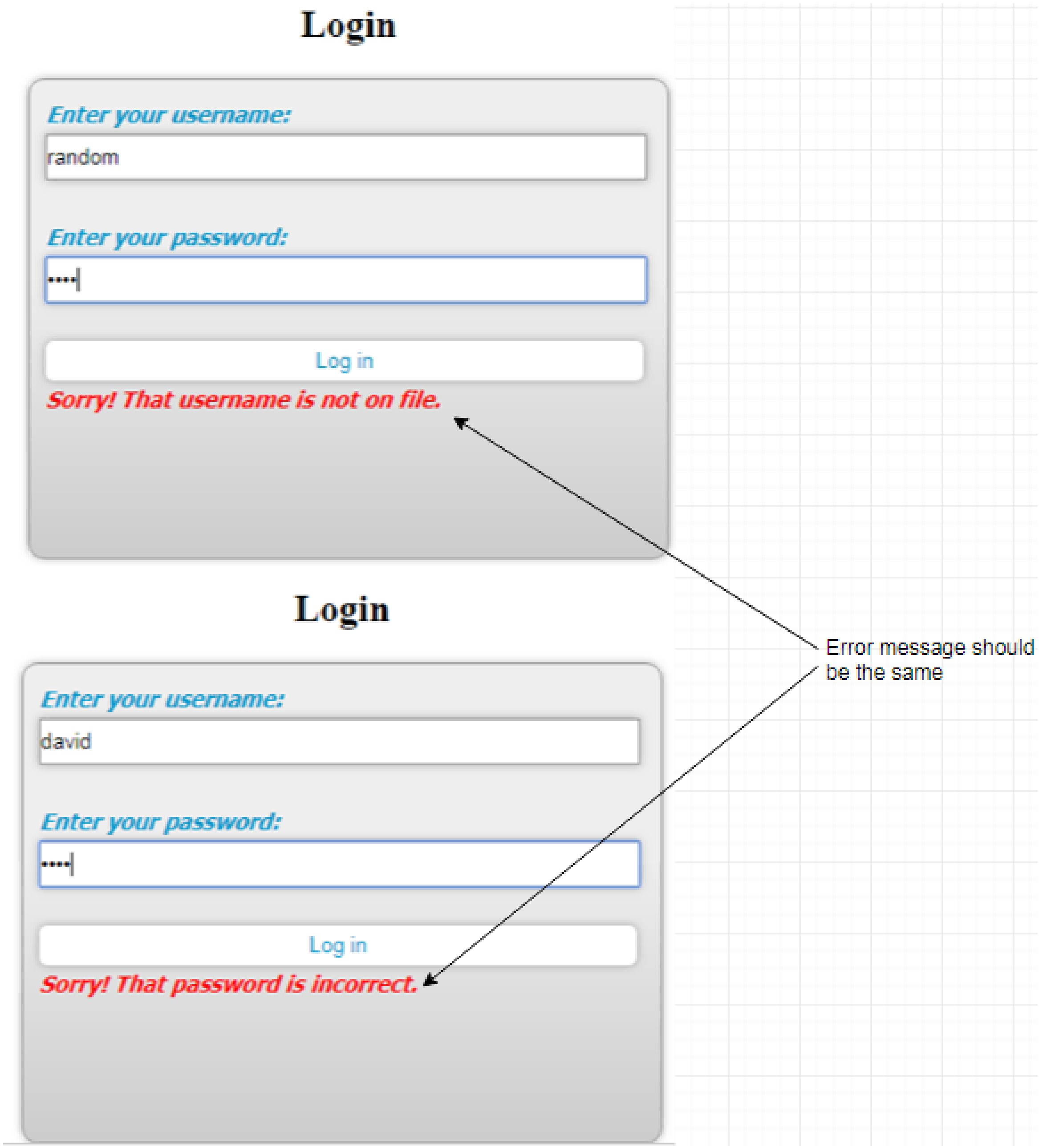
Generic Error Messages: Amend the application functionality to present a generic error message that does not disclose whether the username, password or both are incorrect.

<u>Example:</u> “The username or password you entered does not match our records.”

Password Complexity Requirements: Implement password complexity requirements for passwords when they are created by the users. Check user provided password hash against the hashes of the 10,000 weakest passwords and before accepting them.

Login Failure Threshold: Trigger an account lockout when an incorrect password is provided

	too many times in a row to a certain username. For instance, some companies allow 5 incorrect passwords before locking the user account.
REFERENCE(S)	https://github.com/danielmiessler/SecLists/tree/master/Passwords



Login

Enter your username: random

Enter your password: ••••

Sorry! That username is not on file.

Login

Enter your username: david

Enter your password: ••••

Sorry! That password is incorrect.

Error message should be the same