Cairo University
Faculty of Computers and Artificial Intelligence

SCS359-Software Security
Assignment - 2

Supervised By:
DR.Basheer Abdel Fatah Youssef
TA.Khloud Khaled Attia

Prepared & Implemented By:

Abdelrahman Ashraf ELMahdy	(20186012)
Amro Adel Farid	(20206145)
Dalia Gamal Abdelhamed	(20206023)
George Tadros Emil	(20206014)
Mootaz Medhat Ezzat Abdelwahab	(20206074)

Academic Year: 2023/2024

Second Semester

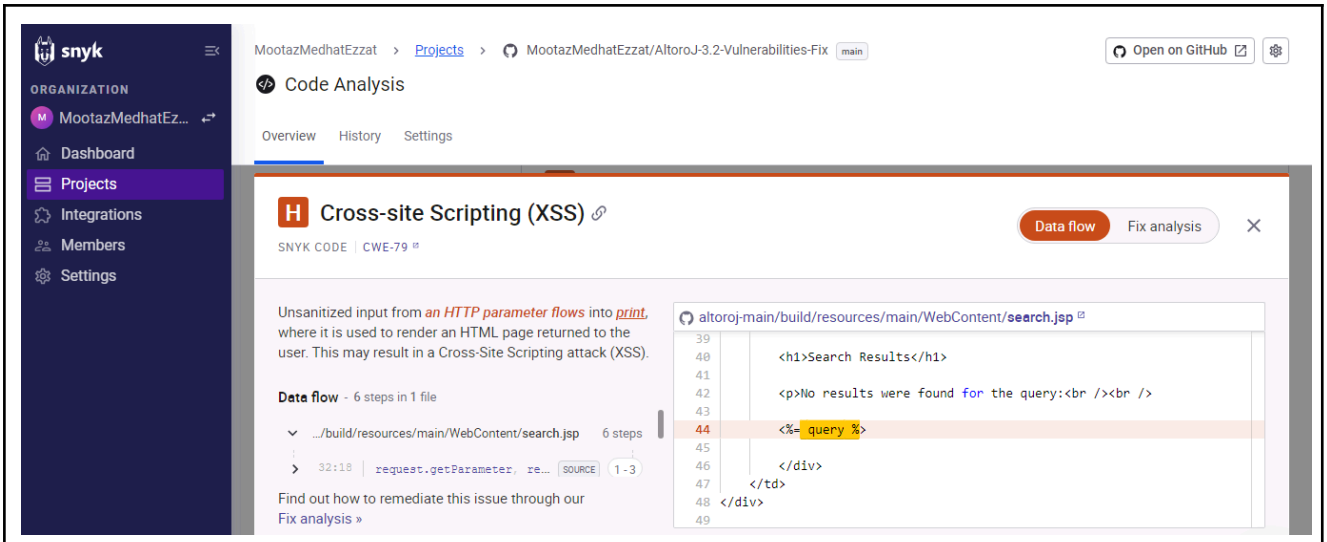
SCS359-Software Security **[Assignment - 2]**

➢ **Findings List with severity, CVSS risk score and risk categorized:**

ID	Name	Original Severity	Snyk score
ID01	Cross Site Scripting **(XSS)** in Search Functionality	High	
ID02	Cross Site Scripting **(XSS)** in Send Feedback Form	High	
ID03	SQL Injection in Login	High	
ID04	Sql injection AdminServlet.java	High	
ID05	Cross Site Scripting **(XSS)** queryxpath.jsp	High	
ID06	Cross site scripting in transaction.jsp	High	
ID07	Open redirect disclaimer.htm	High	**558**
ID08	Cross site scripting serverStatusCheck.html	High	**558**
ID09	Open redirect cusomize.jsp	High	**552**
ID10	Improper Neutralization of CRLF Sequences in HTTP Headers LoginServlet.java	High	**552**
ID11	Code injection serverstatuscheck.html	High	**552**
ID12	Use of hardcoded credentials	High	**504**
ID13	Observable Timing Discrepancy (Timing Attack)	High	**502**
ID14	XML External Entity (XXE) Injection	High	**502**
ID15	Trust Boundary Violation surveyservlet,java	High	**402**
ID16	Sensitive Cookie Without 'HttpOnly' Flag loginservlet.java	High	**402**
ID17	Cross-site Scripting (XSS) balance.jsp	High	**839**
ID18	SQL injection in transaction.jsp	High	**829**

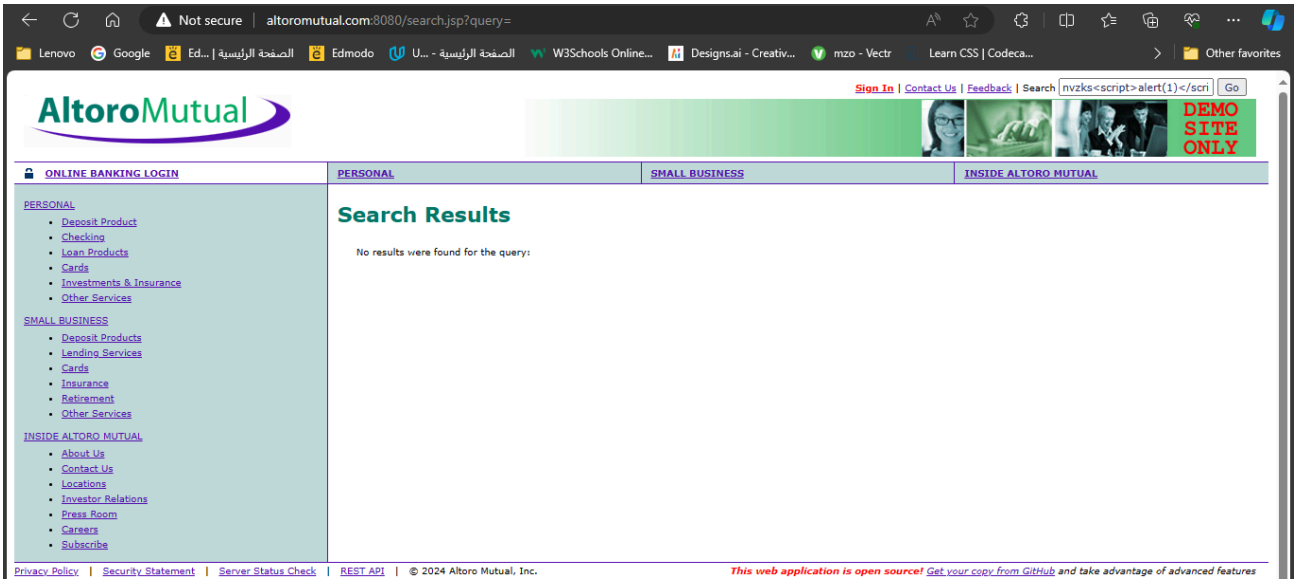
➢ Finding details (test and retest):

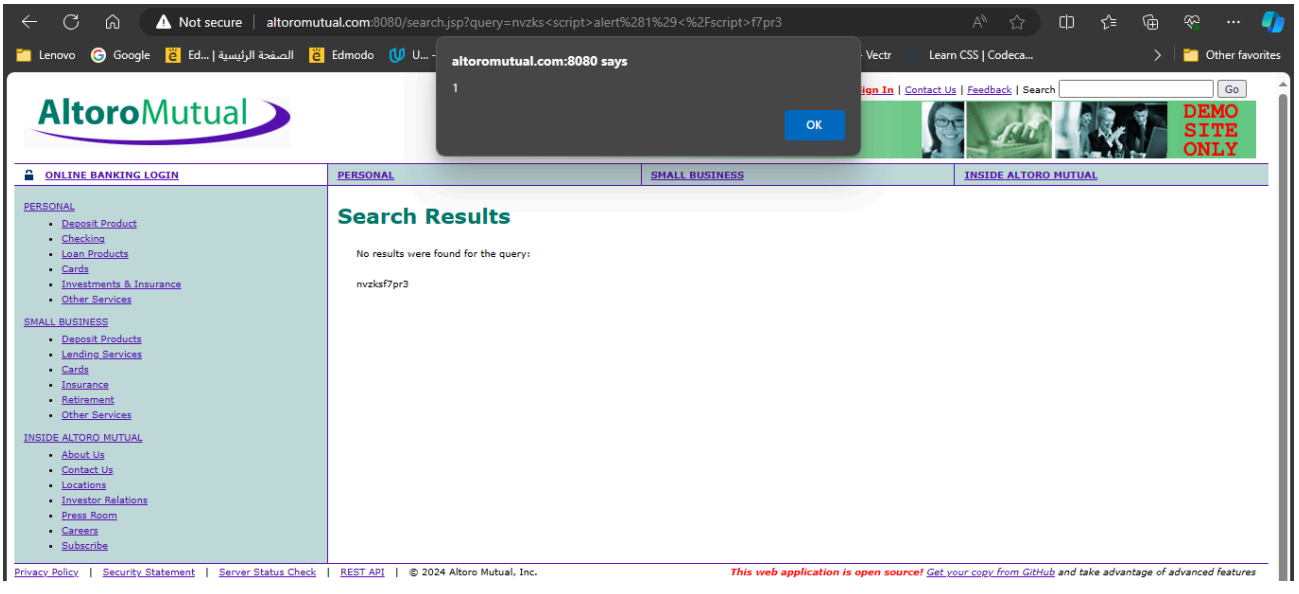
Name (ID01)	Reflected Cross site scripting (XSS) in Search				
Test Severity	High	Test Score	9.4 / Critical	Retest Severity	
Description:					
Upon testing the search functionality of the application located at http://localhost:8080/altoroj-main/search.jsp, it was discovered that the value of the query request parameter is directly reflected back in the HTML response without proper sanitization. This allows an attacker to inject arbitrary JavaScript code into the application's response, potentially leading to XSS attacks (such as the execution of malicious scripts within the context of other users' sessions).					
Definition of Cross-Site Scripting (XSS):					
Cross-Site Scripting vulnerabilities occur when untrusted data is incorporated into a web application's output in an unsafe manner, allowing an attacker to inject malicious scripts into the application's web pages. These scripts can then be executed in the browsers of other users who view the affected pages, potentially leading to various security threats.					
Impact:					
The impact of this vulnerability is severe. An attacker can exploit this flaw to perform various malicious actions, such as stealing session tokens or login credentials, performing unauthorized actions on behalf of users, or even logging their keystrokes. Depending on the application's functionality and the privileges of the affected users, the consequences could range from compromising sensitive data to complete system takeover.					
Recommendations:	To remediate this vulnerability, the following actions are recommended:				
1. Input Validation: Validate all user-controllable input on arrival, enforcing strict criteria based on the expected content. Reject any input that does not meet the validation criteria, rather than attempting to sanitize it.					
2. HTML Encoding: Encode user input whenever it is included in application responses. Replace all HTML metacharacters, such as <, >, ", ', and =, with their corresponding HTML entities (<, >, etc.).					
3. web application firewalls (WAFs): Implement WAFs to provide an additional layer of defense against XSS attacks by filtering and blocking malicious input.					
Finding Test Steps:	☑ Cross-site Scripting (X... 43				
1. Open the the following page located at http://localhost:8080/altoroj-main/search.jsp					



2. Inject the payload **"nvzks<script>alert(1)</script>f7pr3"** into the query parameter.

3. Observed that the injected script executed successfully in the application's response, confirming the presence of an XSS vulnerability.





Fixing Steps:	☑ Cross-site Scripting (X... 42 In the fixed version of **search.jsp**:

1. I've used org.apache.commons.text.StringEscapeUtils.escapeHtml4 method to escape the query parameter before displaying it on the page.

2. This method escapes special characters in the query string to prevent them from being interpreted as HTML or JavaScript code, effectively mitigating the XSS vulnerability.

```
<div id="wrapper" style="width: 99%;">
    <jsp:include page="toc.jspf"/>
    <td valign="top" colspan="3" class="bb">
        <%@page import="com.ibm.security.appscan.altoromutual.util.ServletUtil"%>

        <%
        String query = request.getParameter("query");
        String[] results = null;
        if (query != null && query.trim().length()>0)
            results = ServletUtil.searchSite(query, request.getSession().getServletContext().getRealPath("/static/"));
        %>

        <div class="fl" style="width: 99%;">

        <h1>Search Results</h1>

        <p>No results were found for the query:<br /><br />

        <%= org.apache.commons.text.StringEscapeUtils.escapeHtml4(query) %>

        </div>
    </td>
</div>
```

Name (ID02)	Reflected Cross site scripting (XSS) in Send Feedback Form				
Test Severity	**High**	**Test Score**	**9.4 / Critical**	**Retest Severity**	

Description:

Upon testing the feedback form functionality of the application located at http://localhost:8080/altoroj-main/sendFeedback, it was observed that the value of the **"name"** request parameter is directly reflected back in the HTML response without proper sanitization. This allows an attacker to inject arbitrary JavaScript code into the application's response, potentially leading to the execution of malicious scripts within the context of other users' sessions.

Definition of Cross-Site Scripting (XSS):

Cross-Site Scripting vulnerabilities occur when untrusted data is incorporated into a web application's output in an unsafe manner, allowing an attacker to inject malicious scripts into the application's web pages. These scripts can then be executed in the browsers of other users who view the affected pages, potentially leading to various security threats.

Impact:

The impact of this vulnerability is severe. An attacker can exploit this flaw to perform various malicious actions, such as**:**

- Theft of sensitive information such as session tokens, cookies, or login credentials.
- Session hijacking, allowing the attacker to impersonate legitimate users.
- Performing unauthorized actions on behalf of the victim user.
- Potentially compromising the security of other applications within the same domain or organization.
- Exploiting users' trust in the application to conduct phishing attacks or distribute malware.

Recommendations: To remediate this vulnerability, the following actions are recommended:

1. Strict Input Validation:
Validate and sanitize all user-controllable input upon arrival, ensuring that it adheres to expected formats and does not contain any malicious content. Reject any input that fails validation rather than attempting to sanitize it.

2. HTML Encoding:
Encode user input whenever it is included in application responses. Replace all HTML metacharacters, such as <, >, ", ', and =, with their corresponding HTML entities (<, >, ", ', &).

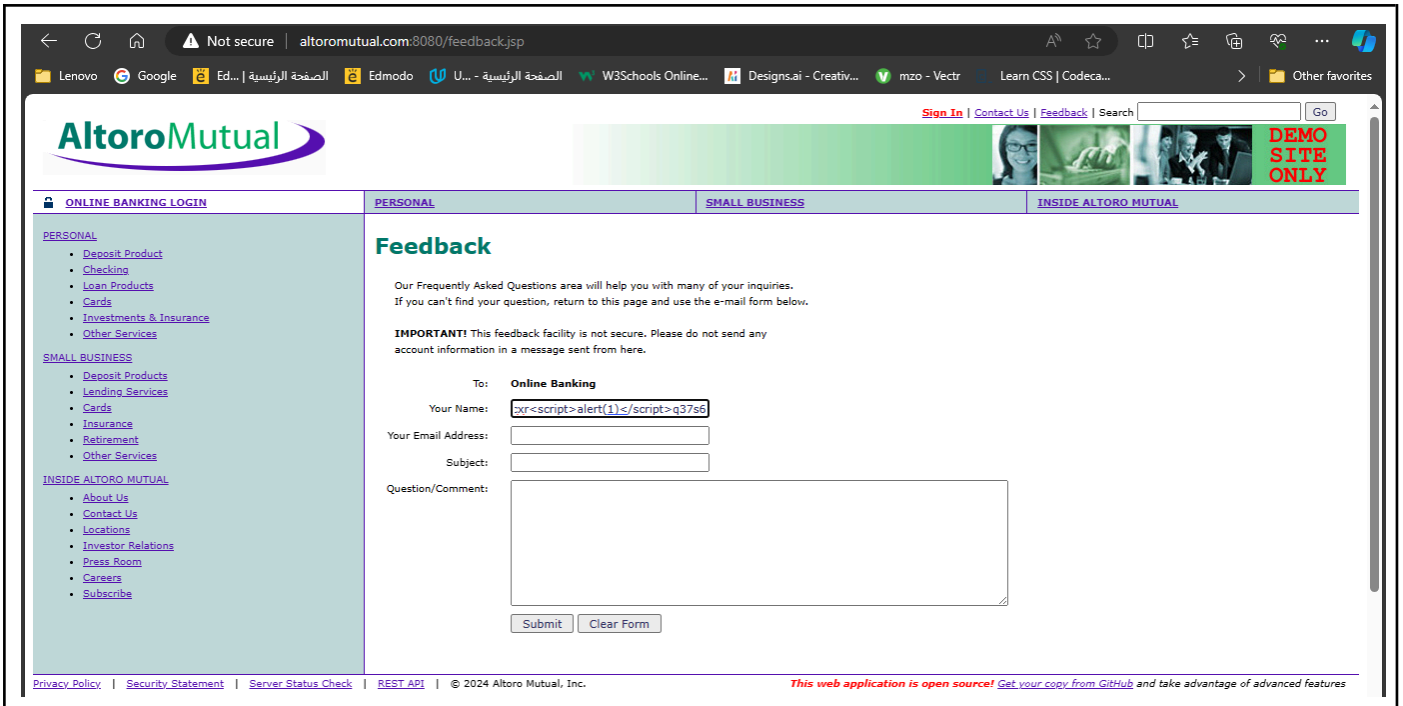
3. Restricted HTML Parsing:
If the application allows users to input HTML content using a restricted subset of tags and attributes (e.g., in blog comments), implement a thorough parsing mechanism to validate that the supplied HTML does not contain any potentially dangerous syntax.

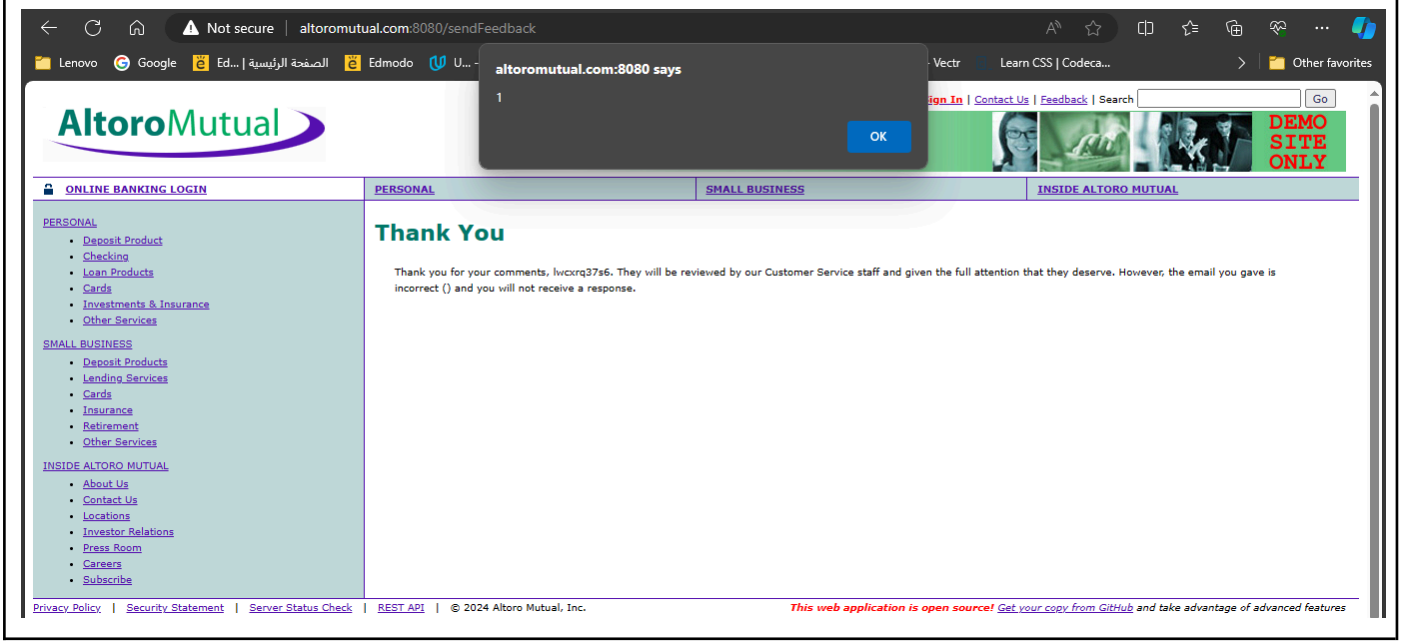
Finding Test Steps:

1. Open the the following page located at http://localhost:8080/altoroj-main/feedback.jsp

2. Inject the payload **"lwcxr<script>alert(1)</script>q37s6"** into the query parameter.

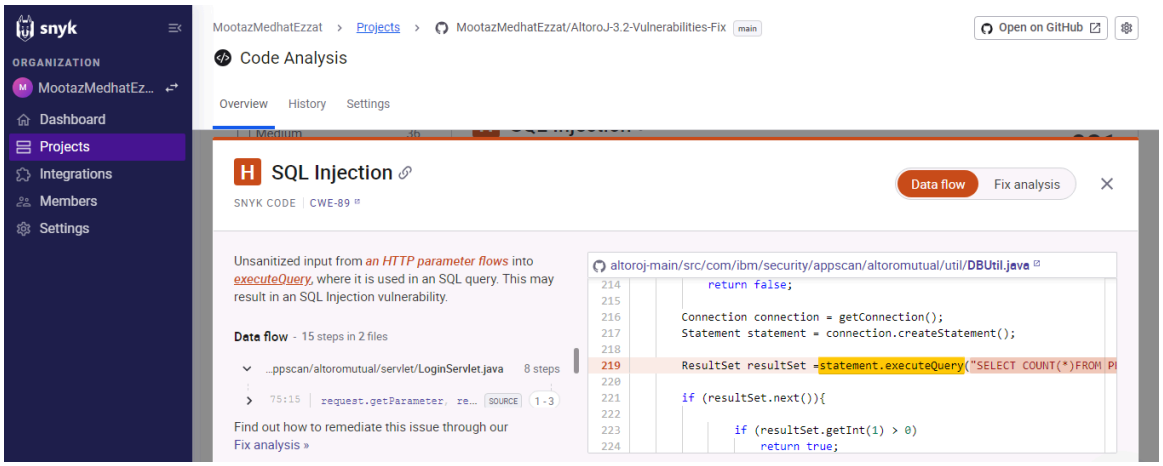
3. Observed that the injected script executed successfully in the application's response, confirming the presence of an XSS vulnerability.


Not secure
altoromutual.com:8080/feedback.jsp
Sign In | Contact Us | Feedback | Search
Go
AltoroMutual
DEMO SITE ONLY
ONLINE BANKING LOGIN
PERSONAL
SMALL BUSINESS
INSIDE ALTORO MUTUAL
Feedback
Our Frequently Asked Questions area will help you with many of your inquiries.
If you can't find your question, return to this page and use the e-mail form below.
IMPORTANT! This feedback facility is not secure. Please do not send any account information in a message sent from here.
To: Online Banking
Your Name: :xr<script>alert(1)</script>q37s6
Your Email Address:
Subject:
Question/Comment:
Submit
Clear Form
Privacy Policy | Security Statement | Server Status Check | REST API | © 2024 Altoro Mutual, Inc.
This web application is open source! Get your copy from GitHub and take advantage of advanced features

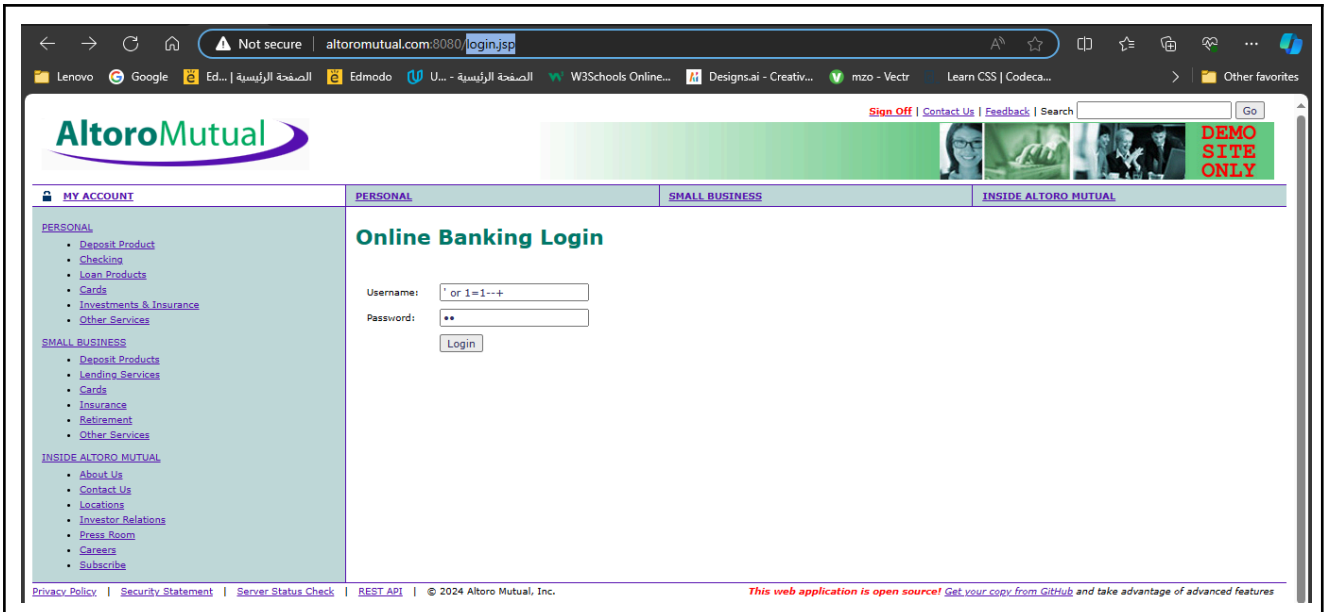

Not secure
altoromutual.com:8080/sendFeedback
altoromutual.com:8080 says
1
OK
AltoroMutual
DEMO SITE ONLY
ONLINE BANKING LOGIN
PERSONAL
SMALL BUSINESS
INSIDE ALTORO MUTUAL
Thank You
Thank you for your comments, lwcxrq37s6. They will be reviewed by our Customer Service staff and given the full attention that they deserve. However, the email you gave is incorrect () and you will not receive a response.
Privacy Policy | Security Statement | Server Status Check | REST API | © 2024 Altoro Mutual, Inc.
This web application is open source! Get your copy from GitHub and take advantage of advanced features

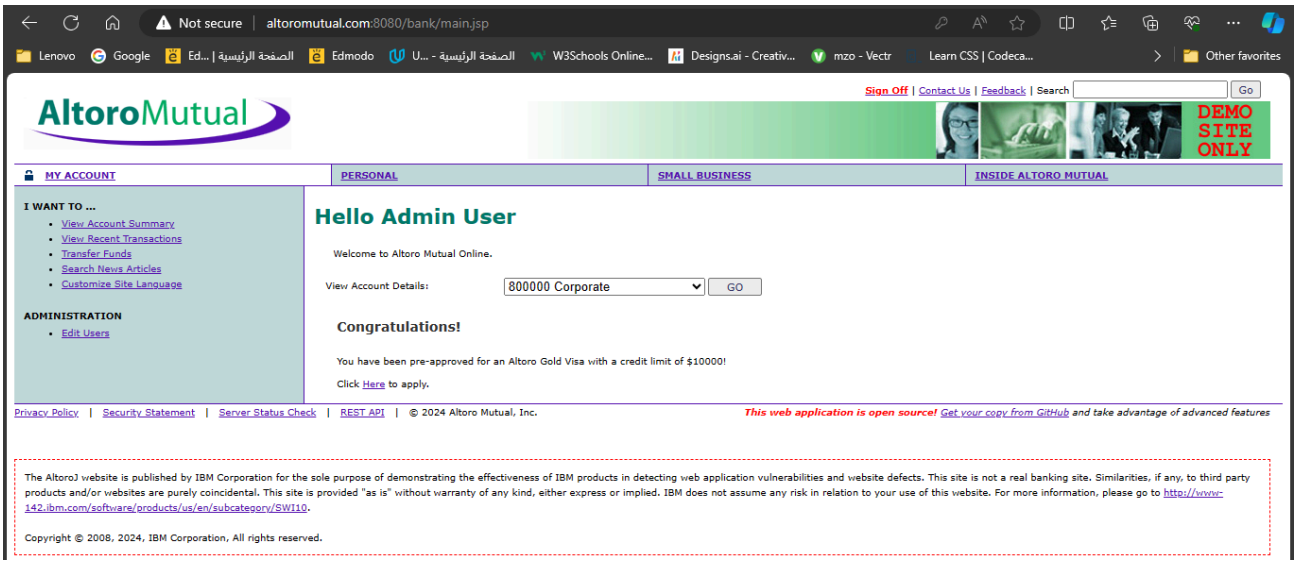
Name (ID03)	SQL Injection in Login				
Test Severity	High	Test Score	9.4 / Critical	Retest Severity	

Description:
Upon testing the search functionality of the application located at http://localhost:8080/altoroj-main/login.jsp, it was discovered that the application is vulnerable to SQL injection attacks. Specifically, it was found that by inserting malicious SQL code into the username field, it's possible to manipulate the authentication mechanism and gain unauthorized access to the application, potentially accessing sensitive data or performing administrative actions.
Definition of SQL Injection (SQLi):
SQL injection (or SQLi) is one of the most widespread code vulnerabilities. To perform a SQL injection attack, an attacker inserts or "injects" malicious SQL code via the input data of the application. SQL injection allows the attacker to read, change, or delete sensitive data as well as execute administrative operations on the database.
Impact:
This SQL injection vulnerability allows attackers to bypass the authentication mechanism of the application and log in as an administrator or any other user without valid credentials. As a result, attackers can gain unauthorized access to sensitive information, manipulate data, or perform administrative actions within the application.
Recommendations: To remediate this vulnerability, the following actions are recommended:
Implement input validation and parameterized queries to prevent SQL injection attacks.
Finding Test Steps: ☑ SQL Injection 17
1. Open the the following page located at http://localhost:8080/altoroj-main/login.jsp



2. Inject the payload **"' or 1=1--+"** into the username field.
3. Attempt to log in.
4. Observed that the application accepts the input without proper validation and allows authentication as an admin user, indicating a successful SQL injection attack.





Fixing Steps:	SQL Injection 16 In the fixed version of **LoginServlet.java**:

1. Input parameters (username and password) are validated for null and empty values.

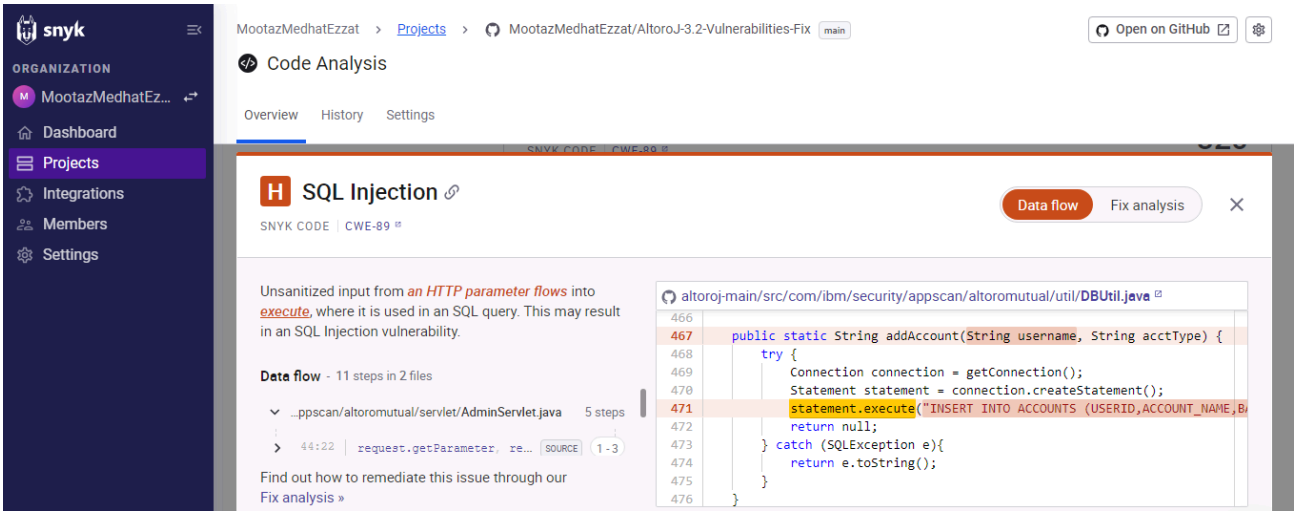
2. User inputs are sanitized by trimming leading and trailing whitespaces.

3. A parameterized query is used to validate user credentials, preventing SQL injection.

4. Exceptions are handled appropriately with error messages or HTTP error codes.

```
// Validate user credentials using parameterized query
private boolean isValidUser(String username, String password) throws SQLException {
    String query = "SELECT COUNT(*) FROM users WHERE username = ? AND password = ?";
    try (PreparedStatement statement = DBUtil.getConnection().prepareStatement(query)) {
        statement.setString(1, username);
        statement.setString(2, password);
        try (ResultSet resultSet = statement.executeQuery()) {
            if (resultSet.next()) {
                int count = resultSet.getInt(1);
                return count > 0;
            }
        }
    }
    return false;
}
```

Name (ID04)	Sql injection AdminServlet.java				
Test Severity	**High**	**Test Score**	**9.4 / Critical**	**Retest Severity**	
Definition of SQL Injection (SQLi):					
SQL injection (or SQLi) is one of the most widespread code vulnerabilities. To perform a SQL injection attack, an attacker inserts or "injects" malicious SQL code via the input data of the application. SQL injection allows the attacker to read, change, or delete sensitive data as well as execute administrative operations on the database.					
Impact:					
This SQL injection vulnerability allows attackers to bypass the authentication mechanism of the application and log in as an administrator or any other user without valid credentials. As a result, attackers can gain unauthorized access to sensitive information, manipulate data, or perform administrative actions within the application.					
Recommendations:	To remediate this vulnerability, the following actions are recommended:				
Implement input validation and parameterized queries to prevent SQL injection attacks.					
Finding Test Steps:	☑ SQL Injection 17				



Id05

Unsanitized input from *an HTTP parameter flows* into *execute*, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Severity high

Id06 Unsanitized input from *an HTTP parameter flows* into *print*, where it is used to render an HTML page returned to the user. This may result in a Cross-Site Scripting attack (XSS).

Severity high

Id07 Unsanitized input from *the document location flows* into *window.location*, where it is used as an URL to redirect the user. This may result in an Open Redirect vulnerability.

Severity high

Id08

Unsanitized input from *data from a remote resource flows* into *eval*, where it is executed as JavaScript code. This may result in a Code Injection vulnerability.

Severity high

Id09

Unsanitized input from *data from a remote resource flows* into *eval*, where it is executed as JavaScript code. This may result in a Code Injection vulnerability.

Severity high

Id10

Unsanitized input from *a database flows* into *addCookie* and reaches an HTTP header returned to the user. This may allow a malicious input that contain CR/LF to split the http response into two responses and the second response to be controlled by the attacker. This may be used to mount a range of attacks such as cross-site scripting or cache poisoning.

Severity high

Id12

Do not hardcode passwords in code. Found hardcoded password used in *Password*.

Severity high

Id13

An attacker can guess the secret value of *password* because it is compared using *equals*, which is vulnerable to timing attacks. Use java.security.MessageDigest.isEqual to compare values securely.

Sserity high

Id14

A file is loaded by *parse*, which allows expansion of external entity references. This may result in an XXE attack leading to the disclosure of confidential data or denial of service.

Id15

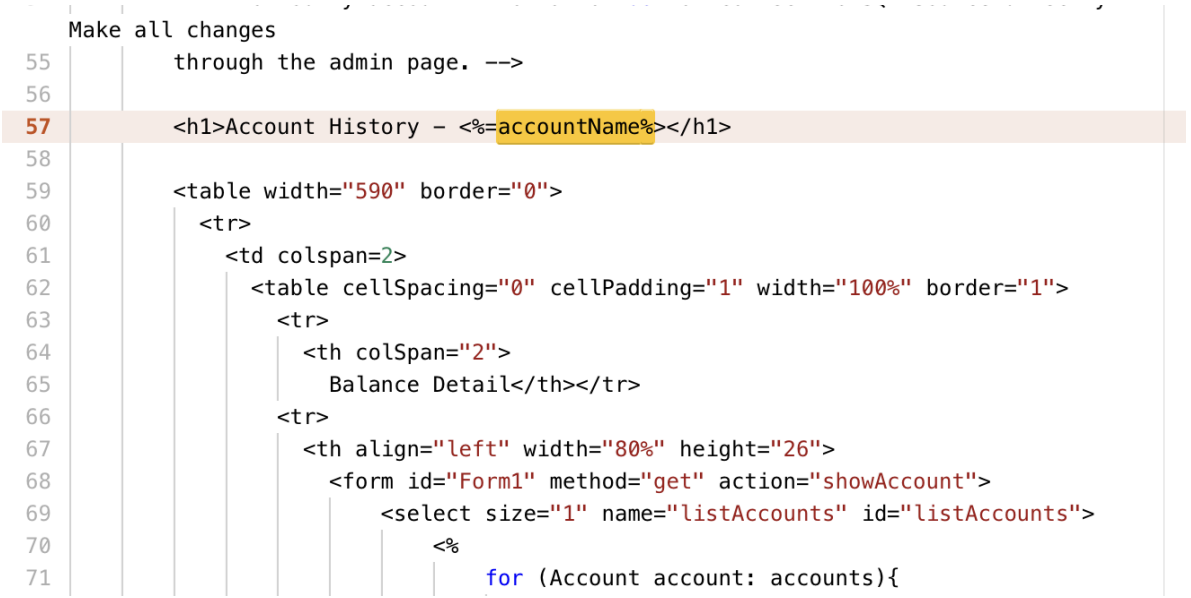
Unsanitized input from *an HTTP parameter flows* into *setAttribute* where it is used to modify the HTTP session object. This could result in mixing trusted and untrusted data in the same data structure, thus increasing the likelihood to mistakenly trust unvalidated data.

Medium

Id17

Unsanitized input from *an HTTP parameter flows* into *print*, where it is used to render an HTML page returned to the user. This may result in a Cross-Site Scripting attack (XSS).

Severity high



Fix:

<h1>Account History - <%= org.apache.commons.lang.StringEscapeUtils.escapeHtml(accountName) %></h1>

<%@ page import="org.apache.commons.lang.StringEscapeUtils" %>

In this fix, I used StringEscapeUtils.escapeHtml() from Apache Commons Lang library to HTML encode the accountName. This will ensure that any potentially harmful HTML characters in accountName are properly encoded and displayed as plain text in the HTML response, preventing XSS attacks.

Id18 Unsanitized input from an HTTP parameter flows into executeQuery, where it is used in an SQL query. This may result in an SQL Injection vulnerability.

Severity high

```
390
391             if (startDate != null && startDate.length()>0 && endDate != null && endDat
392                 dateString = "DATE BETWEEN '" + startDate + " 00:00:00' AND '" + endDa
393             } else if (startDate != null && startDate.length()>0){
394                 dateString = "DATE > '" + startDate +" 00:00:00'";
395             } else if (endDate != null && endDate.length()>0){
396                 dateString = "DATE < '" + endDate + " 23:59:59'";
397             }
398
399             String query = "SELECT * FROM TRANSACTIONS WHERE (" + acctIds.toString() +
400             ResultSet resultSet = null;
401
402             try {
403                 resultSet = statement.executeQuery(query);
404             } catch (SQLException e){
405                 int errorCode = e.getErrorCode();
406                 if (errorCode == 30000)
407                     throw new SQLException("Date-time query must be in the format of y
408
```