Preventing Injection Attacks Report

Project Preventing Injection Attacks

Jai Vang

ITIS 4221-091

Secure Programing and Penetration Testing

November 2nd, 2023

TABLE OF CONTENTS

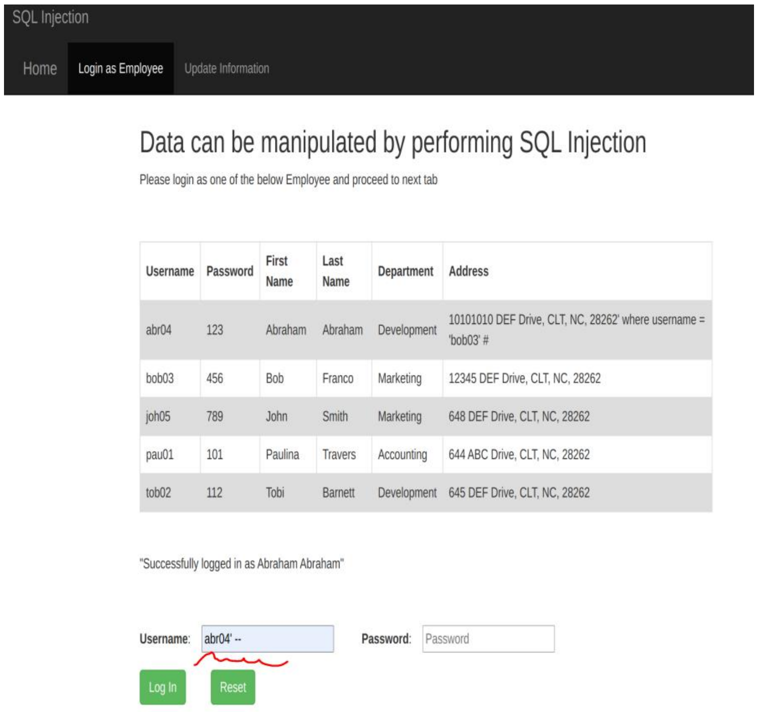
1.0 General Information

1.1 Purpose

The objective of this assessment is to identify and address vulnerabilities within the application. It aims to identify and fix vulnerabilities to safeguard sensitive data and maintain user trust. In this report, we will address vulnerabilities in the "penetration_test" application by implementing mitigations for SQL Injection, XSS, Path Manipulation, Command Injection, Log Forging, XPath Injection, and Simple Mail Transfer Protocol Injection. Ensuring the security of the “penetration_test” application is imperative in today’s interconnected digital landscape.

2.0 SQL Injection

SQL Injection is a critical security vulnerability that arises when a hacker injects malicious SQL code into a query. In the application, this injection becomes evident when an attacker inputs 'abr04' –' into the 'Username' field, enabling unauthorized access, as depicted in **Figure 1**. Another vulnerability surfaces within the 'Update Information' tab, where a malicious user can input '12345 DEF Drive, CLT, NC, 28262' where username = 'bob03' #' in the 'Address' field, as illustrated in **Figure 1A**. This exploit allows hackers to automatically alter Bob's address. Implementing SQL injection mitigations is crucial to curbing and eliminating such attacks, thereby preventing, and minimizing their impact. Below, we outline SQL injection vulnerabilities identified in the SQL_injectionController.java class, and we will employ mitigations to rectify these exposures.



Data can be manipulated by performing SQL Injection

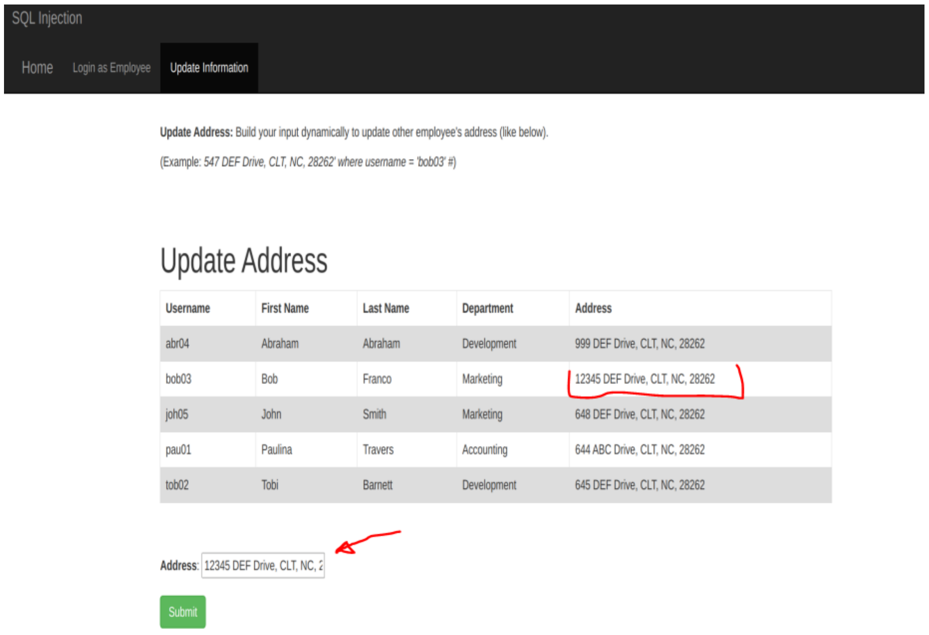
Please login as one of the below Employee and proceed to next tab

Username	Password	First Name	Last Name	Department	Address
abr04	123	Abraham	Abraham	Development	10101010 DEF Drive, CLT, NC, 28262' where username = 'bob03' #
bob03	456	Bob	Franco	Marketing	12345 DEF Drive, CLT, NC, 28262
joh05	789	John	Smith	Marketing	648 DEF Drive, CLT, NC, 28262
pau01	101	Paulina	Travers	Accounting	644 ABC Drive, CLT, NC, 28262
tob02	112	Tobi	Barnett	Development	645 DEF Drive, CLT, NC, 28262

"Successfully logged in as Abraham Abraham"

Username: abr04' -- Password: Password

Figure 1. SQL Injection on Username input



Update Address: Build your input dynamically to update other employee's address (like below).

(Example: *547 DEF Drive, CLT, NC, 28262' where username = 'bob03' #*)

Update Address

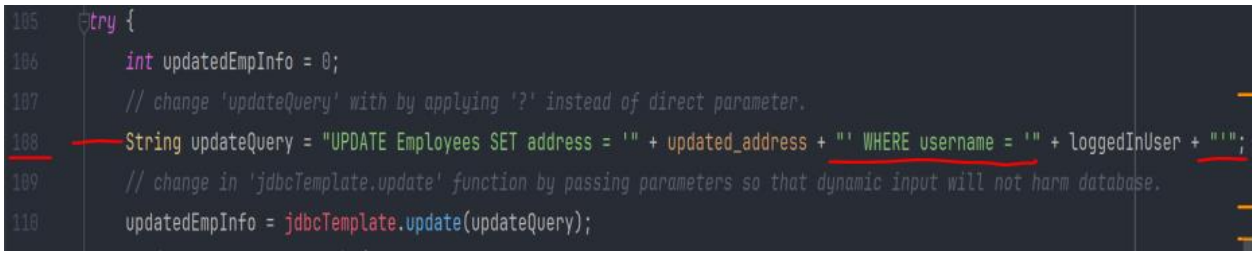
Username	First Name	Last Name	Department	Address
abr04	Abraham	Abraham	Development	999 DEF Drive, CLT, NC, 28262
bob03	Bob	Franco	Marketing	12345 DEF Drive, CLT, NC, 28262
joh05	John	Smith	Marketing	648 DEF Drive, CLT, NC, 28262
pau01	Paulina	Travers	Accounting	644 ABC Drive, CLT, NC, 28262
tob02	Tobi	Barnett	Development	645 DEF Drive, CLT, NC, 28262

Address: 12345 DEF Drive, CLT, NC, 2

Figure 1A. SQL Injection on Address input

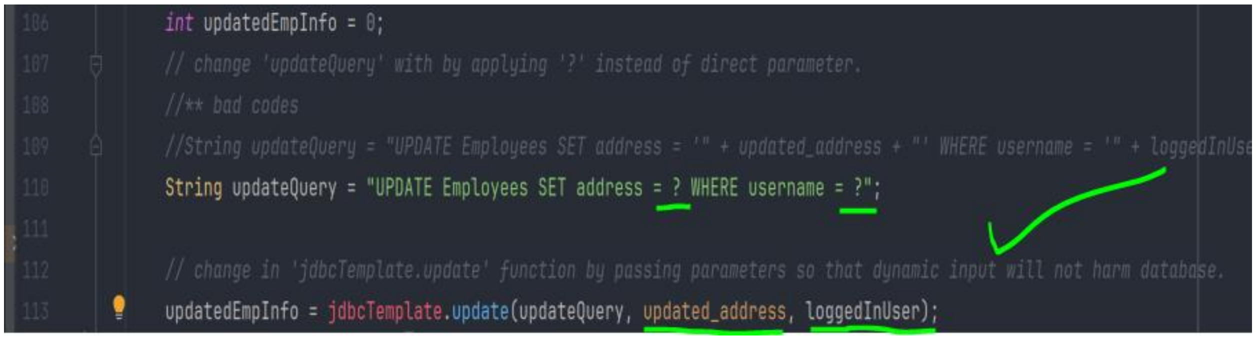
2.1 SQL Injection Mitigation

In the update_address method at line 108, it is evident from **Figure 2** that the construction of the update address is susceptible to SQL injection attacks due to the absence of parameterization. To mitigate this vulnerability, we will implement parameterized queries, utilizing placeholders ('?') for parameters instead of directly incorporating update-address and loggedInUser values. Additionally, we will include update_address and loggedInUser as extra arguments in the jdbcTemplate.update() method. This modification serves to safeguard against SQL injection by treating user input as data rather than executable code, as depicted in **Figure 2A**. The idea behind passing update_address and loggedInUser into jdbcTemplate.update() is to counteract potential injection vulnerabilities that may arise if they contain malicious SQL code. Treating them as parameters ensures their secure binding to the placeholders within the SQL query.



```
105  try {
106      int updatedEmpInfo = 0;
107      // change 'updateQuery' with by applying '?' instead of direct parameter.
108      String updateQuery = "UPDATE Employees SET address = '" + updated_address + "' WHERE username = '" + loggedInUser + "'";
109      // change in 'jdbcTemplate.update' function by passing parameters so that dynamic input will not harm database.
110      updatedEmpInfo = jdbcTemplate.update(updateQuery);
```

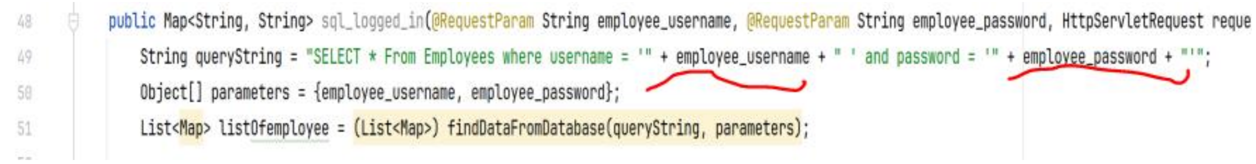
Figure 2. Update Adress Without Parameterization



```
106      int updatedEmpInfo = 0;
107      // change 'updateQuery' with by applying '?' instead of direct parameter.
108      //** bad codes
109      //String updateQuery = "UPDATE Employees SET address = '" + updated_address + "' WHERE username = '" + loggedInUse
110      String updateQuery = "UPDATE Employees SET address = ? WHERE username = ?";
111
112      // change in 'jdbcTemplate.update' function by passing parameters so that dynamic input will not harm database.
113      updatedEmpInfo = jdbcTemplate.update(updateQuery, updated_address, loggedInUser);
```

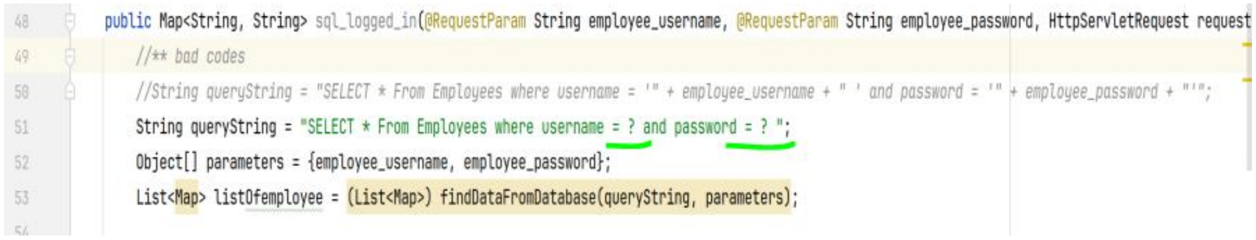
Figure 2A. SQL Injection Mitigation

Furthermore, within the sql_logged_in method, vulnerabilities are evident on line 49, as illustrated in **Figure 3**. To address this issue, we will rectify the situation by employing direct concatenation with placeholders ('?') in the SQL query. We will utilize an array of parameters to supply the actual values, as demonstrated in **Figure 3A**.



```
48   public Map<String, String> sql_logged_in(@RequestParam String employee_username, @RequestParam String employee_password, HttpServletRequest reque
49       String queryString = "SELECT * From Employees where username = '" + employee_username + "' and password = '" + employee_password + "'";
50       Object[] parameters = {employee_username, employee_password};
51       List<Map> listOfemployee = (List<Map>) findDataFromDatabase(queryString, parameters);
```

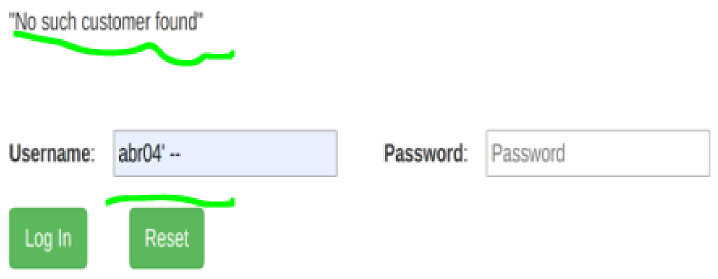
Figure 3. Username and password vulnerabilities



```
48   public Map<String, String> sql_logged_in(@RequestParam String employee_username, @RequestParam String employee_password, HttpServletRequest request
49       //** bad codes
50       //String queryString = "SELECT * From Employees where username = '" + employee_username + "' and password = '" + employee_password + "'";
51       String queryString = "SELECT * From Employees where username = ? and password = ? ";
52       Object[] parameters = {employee_username, employee_password};
53       List<Map> listOfemployee = (List<Map>) findDataFromDatabase(queryString, parameters);
```

Figure 3A. Fixed vulnerabilities

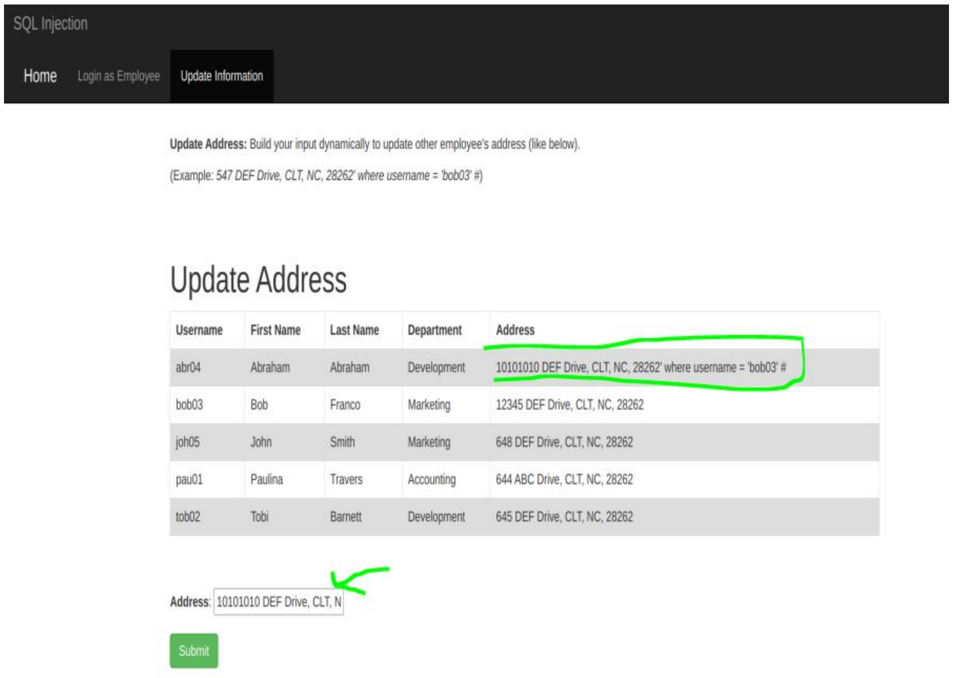
Below, we display the resolved vulnerabilities in **Figure 4** and **Figure 4A**. When an attacker tries to log into the user's account with 'abr04' --', an error message is displayed stating 'No such customer found'. Similarly, if a hacker attempts to modify another user's address, only the attacker's address is altered, not bob03's. In this instance, only the address associated with user abr04 was modified. These vulnerabilities have been successfully addressed and fixed.



"No such customer found"

Username: abr04' --

Figure 4. Fixed vulnerabilities #1



Update Address: Build your input dynamically to update other employee's address (like below).

(Example: *547 DEF Drive, CLT, NC, 28262' where username = 'bob03' #*)

Update Address

Username	First Name	Last Name	Department	Address
abr04	Abraham	Abraham	Development	10101010 DEF Drive, CLT, NC, 28262' where username = 'bob03' #
bob03	Bob	Franco	Marketing	12345 DEF Drive, CLT, NC, 28262
joh05	John	Smith	Marketing	648 DEF Drive, CLT, NC, 28262
pau01	Paulina	Travers	Accounting	644 ABC Drive, CLT, NC, 28262
tob02	Tobi	Barnett	Development	645 DEF Drive, CLT, NC, 28262

Address: 10101010 DEF Drive, CLT, N

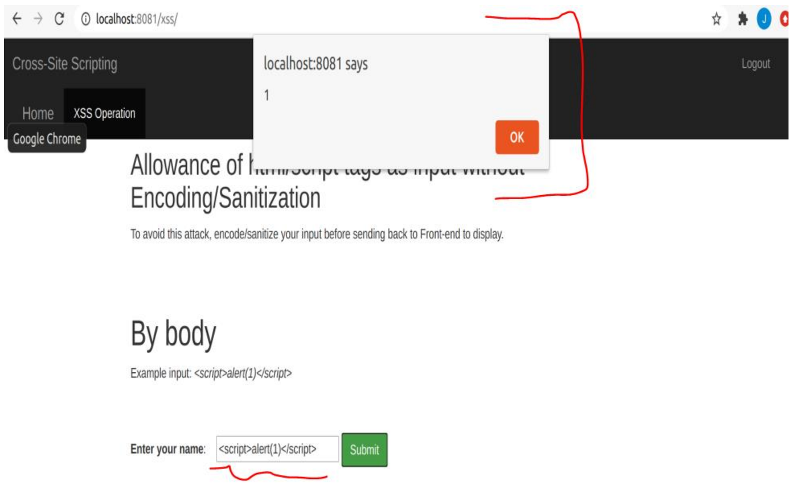
Figure 4A. Hacker address was changed and instead of bob03's

3.0 XSS Vulnerability

XSS attacks transpire when a perpetrator injects malicious code into a website, subsequently delivering it to an unsuspecting user. Deceived into thinking the script is starting, the user unintentionally executes it, assuming it is secure and an integral part of the application. Such attacks are capable of extracting session tokens and browser cookies, and in more severe cases, redirecting users to a different site under the attacker's control.

3.1 XSS Vulnerability Mitigation

In the application, an attacker could input malicious scripts, such as '<script>alert(1)</script>', into an input box and then click the submit button. Subsequently, a pop-up window displaying coded content will appear, indicating the success of the attack (refer to **Figure 5**). Similarly, an attacker can input malicious JavaScript code, like 'xyz.pdf' onClick='alert(1)', into the input box and click the submit button. Following this, a deceptive URL will emerge, prompting the user to select 'Click to download'. If the user is unaware of the potential risks associated with clicking the URL, they might proceed, unwittingly allowing the execution of scripts within the application without their knowledge (see **Figure 6**).



Allowance of html/script tags as input without Encoding/Sanitization

To avoid this attack, encode/sanitize your input before sending back to Front-end to display.

By body

Example input: <script>alert(1)</script>

Enter your name: <script>alert(1)</script> Submit

Figure 5. Successful XSS attack

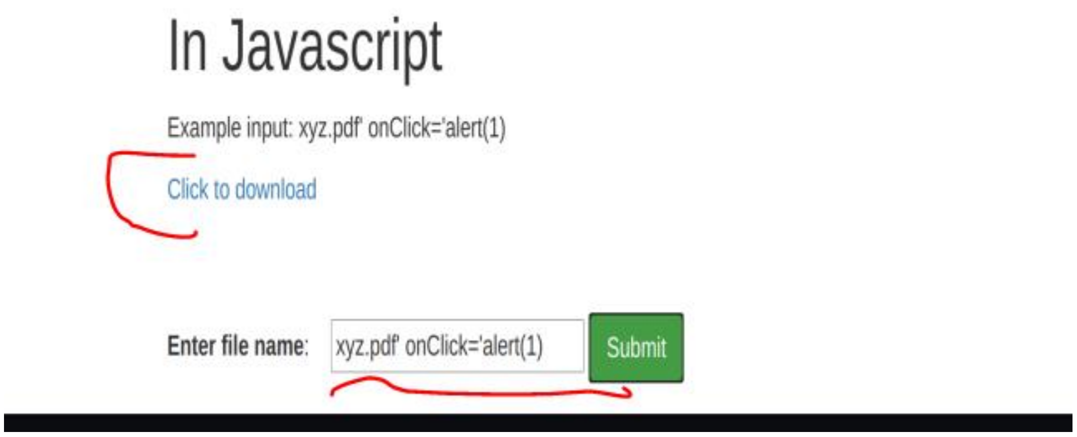
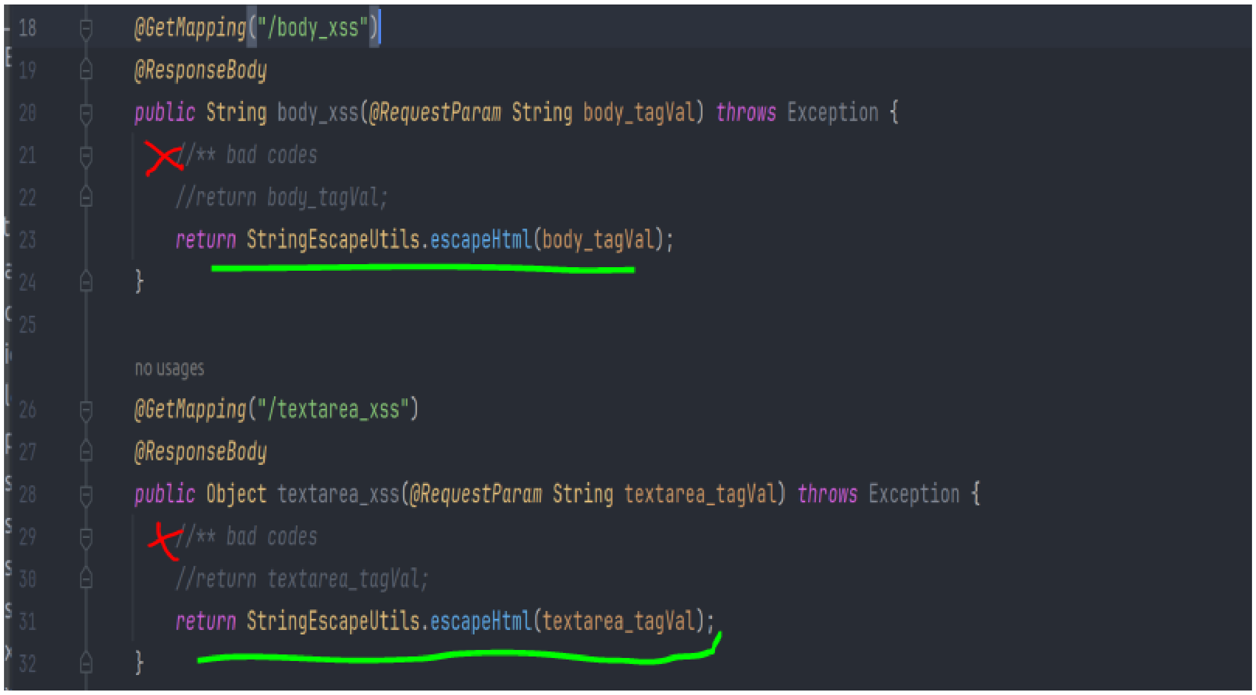

In Javascript
Example input: xyz.pdf' onClick='alert(1)
Click to download
Enter file name:
xyz.pdf' onClick='alert(1)
Submit

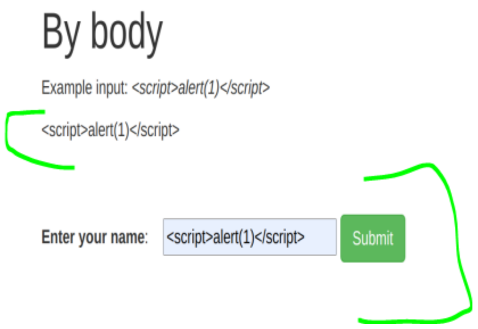
Figure 6. Malicious URL

To prevent XSS vulnerabilities, it is essential to make modifications in the backend codes, as depicted in **Figure 7**. Employing StringEscapeUtils.escapeHTML() is a crucial step in this process, as it ensures the replacement of any HTML special characters (such as '<', '>', '&', etc.) with their corresponding HTML entities. By escaping these HTML characters, the method prevents attacks and safeguards against the interpretation of values as HTML code by the browser. This proactive measure prevents attackers from injecting harmful scripts into a webpage. As illustrated in **Figure 8**, implementing this approach sanitizes user input, ensuring that what is displayed on a webpage is plain text rather than potentially malicious code.



```
@GetMapping("/body_xss")
@ResponseBody
public String body_xss(@RequestParam String body_tagVal) throws Exception {
    //** bad codes
    //return body_tagVal;
    return StringEscapeUtils.escapeHtml(body_tagVal);
}

no usages
@GetMapping("/textarea_xss")
@ResponseBody
public Object textarea_xss(@RequestParam String textarea_tagVal) throws Exception {
    //** bad codes
    //return textarea_tagVal;
    return StringEscapeUtils.escapeHtml(textarea_tagVal);
}
```

Figure 7. Comment out bad codes and add good codes.



By body

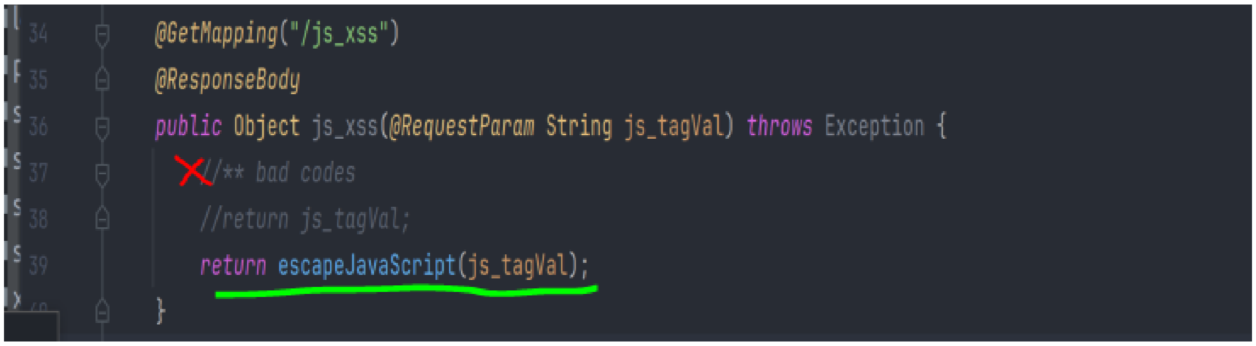
Example input: *<script>alert(1)</script>*

<script>alert(1)</script>

Enter your name: <script>alert(1)</script>

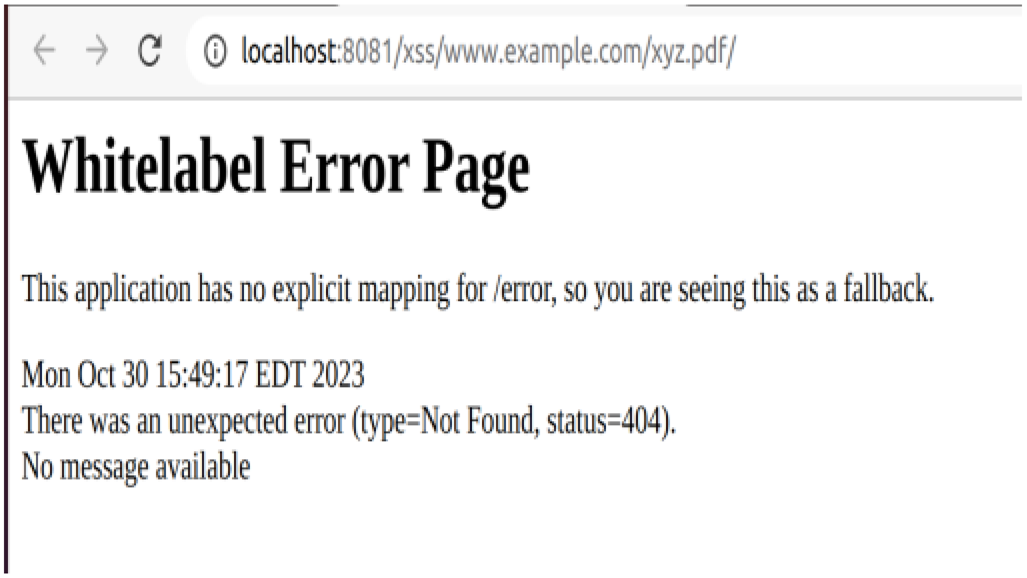
Figure 8. Malicious script treated as plain text.

Similarly, we will comment out the bad codes in line 38 and add a escapeJavaScript() method to line 39 as shown in **Figure 9.** The escapeJavaScript() method is similar to the StringEscapeUtils.escapeHTML() method. If a malicious script gets executed and the user select the URL, the user will be directed to an error message and will not see an alter box as shown in **Figure 10**.



```
@GetMapping("/js_xss")
@ResponseBody
public Object js_xss(@RequestParam String js_tagVal) throws Exception {
    //** bad codes
    //return js_tagVal;
    return escapeJavaScript(js_tagVal);
}
```

Figure 9. Comment out bad codes/Added good codes.



localhost:8081/xss/www.example.com/xyz.pdf/

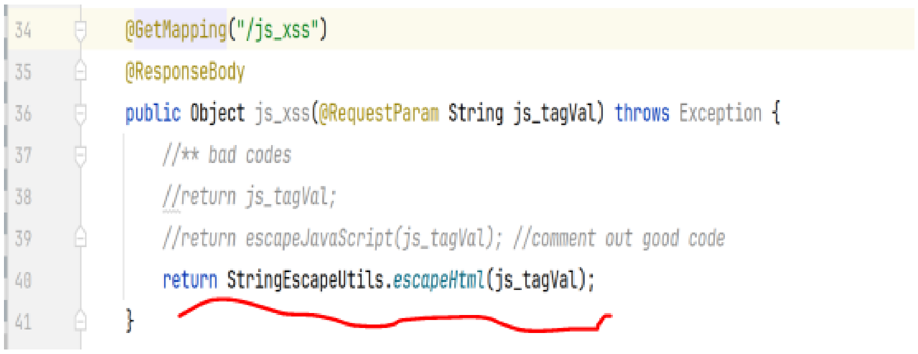
Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Mon Oct 30 15:49:17 EDT 2023
There was an unexpected error (type=Not Found, status=404).
No message available

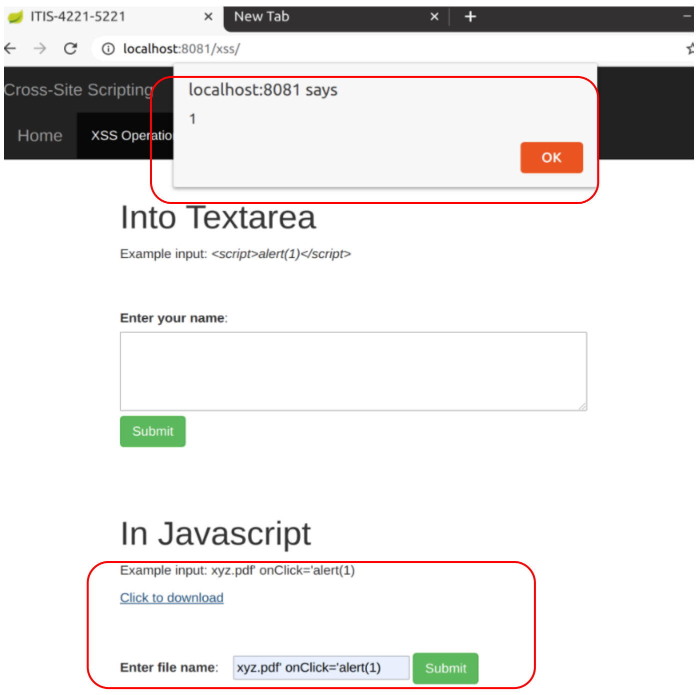
Figure 10. Users see error message in another tab instead of an alter box.

However, if we use the StringEscapeUtils.escapeHtml() instead of escapeJavaScript() (**Figure 10A**), then the malicious code will execute as shown in **Figure 10B.**



```java
@GetMapping("/js_xss")
@ResponseBody
public Object js_xss(@RequestParam String js_tagVal) throws Exception {
    //** bad codes
    //return js_tagVal;
    //return escapeJavaScript(js_tagVal); //comment out good code
    return StringEscapeUtils.escapeHtml(js_tagVal);
}
```

Figure 10A. Using StringEscapeUtils.escapeHtml() instead of escapeJavaScript()



localhost:8081 says

1

Into Textarea

Example input: <script>alert(1)</script>

Enter your name:

In Javascript

Example input: xyz.pdf' onClick='alert(1)

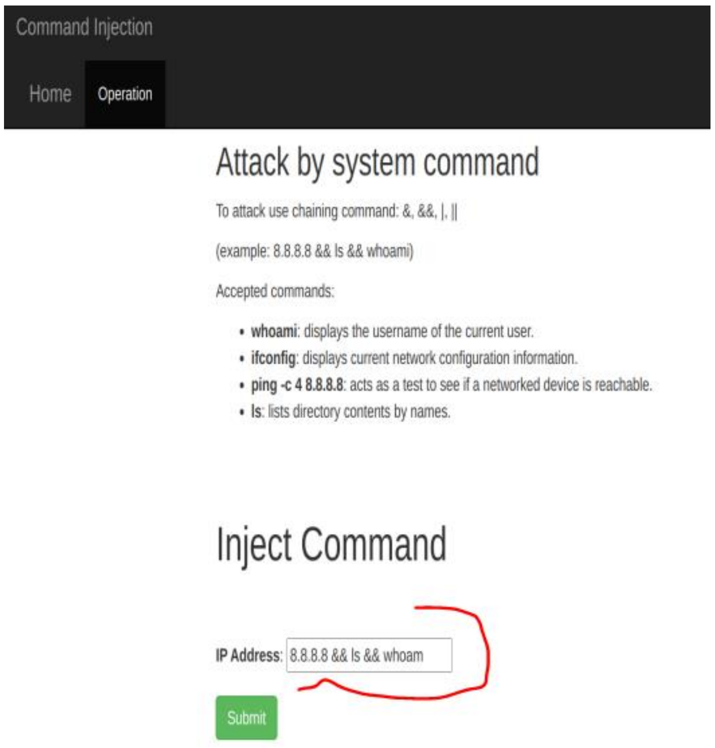
Click to download

Enter file name: xyz.pdf' onClick='alert(1)

Figure 10B. Malicious code executed because of StringEscapeUtils.escapeHtml() usage

4.0 Command Injection

Command Injection vulnerabilities arise when an application incorporates user input into a system-executed command. In this scenario, attackers manipulate input to execute commands, potentially gaining unauthorized access and triggering security concerns. For example, inputting '8.8.8.8 && ls && whoami' into the IP Address field (refer to **Figure 11**) and clicking the submit button can display a list of sensitive information (see **Figure 12**) due to the execution of additional commands beyond the intended 'ping' command. To prevent such risks, we will implement mitigation measures by thoroughly validating and sanitizing the input for the IP address.



Attack by system command

To attack use chaining command: &, &&, |, ||

(example: 8.8.8.8 && ls && whoami)

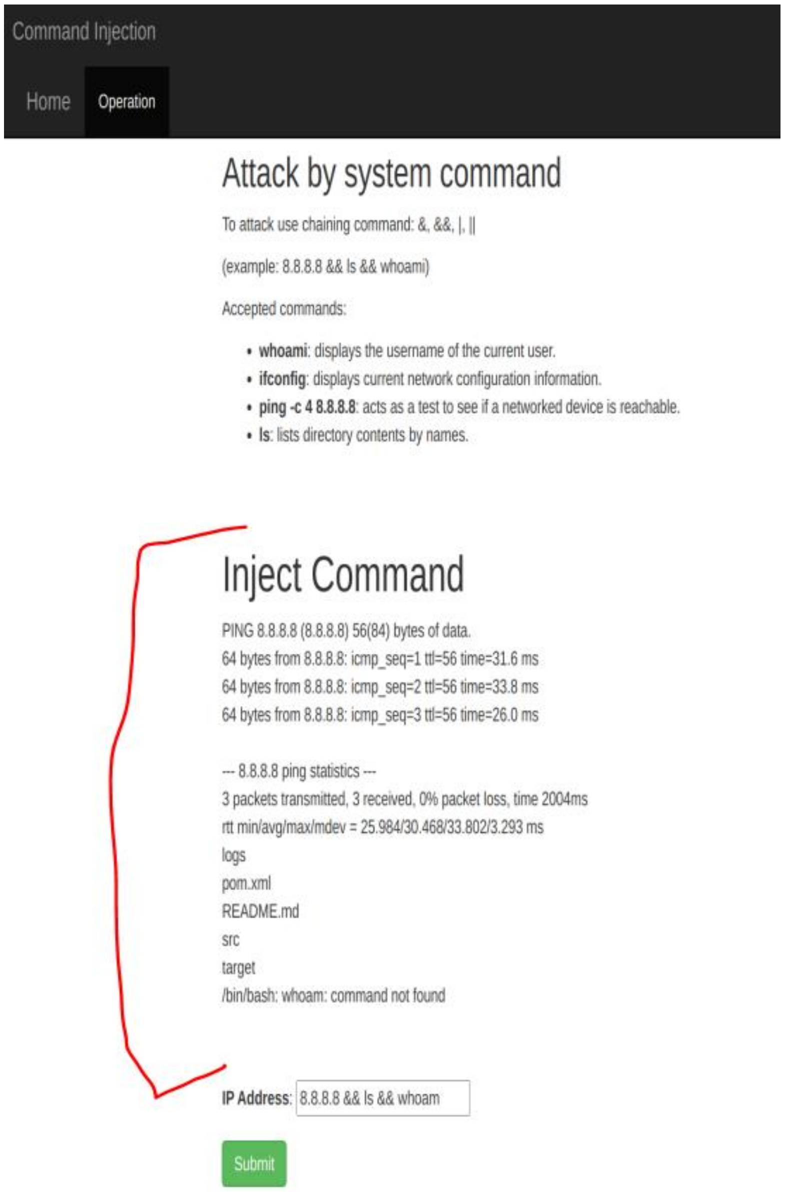
Accepted commands:

- **whoami**: displays the username of the current user.
- **ifconfig**: displays current network configuration information.
- **ping -c 4 8.8.8.8**: acts as a test to see if a networked device is reachable.
- **ls**: lists directory contents by names.

Inject Command

IP Address: 8.8.8.8 && ls && whoam

Figure 11. Malicious Input



Attack by system command

To attack use chaining command: &, &&, |, ||

(example: 8.8.8.8 && ls && whoami)

Accepted commands:

- **whoami**: displays the username of the current user.
- **ifconfig**: displays current network configuration information.
- **ping -c 4 8.8.8.8**: acts as a test to see if a networked device is reachable.
- **ls**: lists directory contents by names.

Inject Command

```
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=56 time=31.6 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=56 time=33.8 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=56 time=26.0 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 25.984/30.468/33.802/3.293 ms
logs
pom.xml
README.md
src
target
/bin/bash: whoam: command not found
```

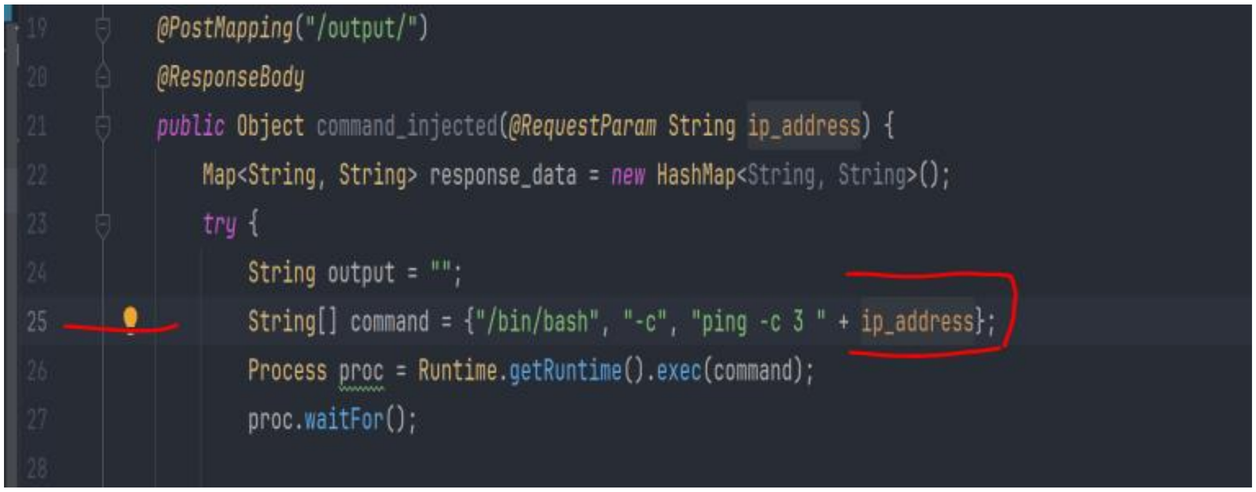
IP Address: 8.8.8.8 && ls && whoam

Figure 12. List of Sensitive Information Displayed

4.1 Command Injection Mitigation

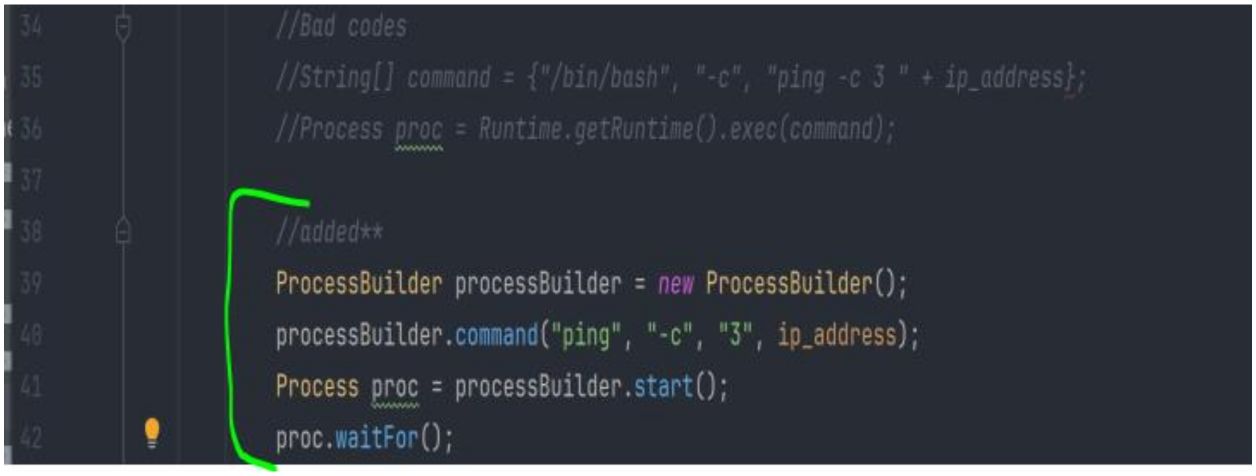
In line 25 of the command_injected method of the Command_injectionCOntroller.java class, there is a Command Injection vulnerability. The issue lies in the way the user can input an (‘ip_address’) without proper validation or sanitization as seen in **Figure 13**. The ‘ip_address’ variable will be concatenated into the command array string. An attacker could manipulate the ‘ip_address’ input and inject malicious commands. To fix this vulnerability, we will validate and sanitize the user input by replacing the vulnerable codes (line 25 and 26) with some new codes as seen in **Figure 14.**

At line 25 within the command_injected method of the Command_injectionController.java class, a Command Injection vulnerability is identified. The problem stems from the user's ability to input an 'ip_address' without sufficient validation or sanitization, as illustrated in **Figure 13**. The 'ip_address' variable is directly concatenated into the command array string, creating an avenue for potential manipulation by attackers who could inject malicious commands. To address this vulnerability, we will enhance security by replacing the susceptible code at lines 25 and 26 with new, more secure code, as demonstrated in **Figure 14**.



```
19  @PostMapping("/output/")
20  @ResponseBody
21  public Object command_injected(@RequestParam String ip_address) {
22      Map<String, String> response_data = new HashMap<String, String>();
23      try {
24          String output = "";
25          String[] command = {"/bin/bash", "-c", "ping -c 3 " + ip_address};
26          Process proc = Runtime.getRuntime().exec(command);
27          proc.waitFor();
28
```

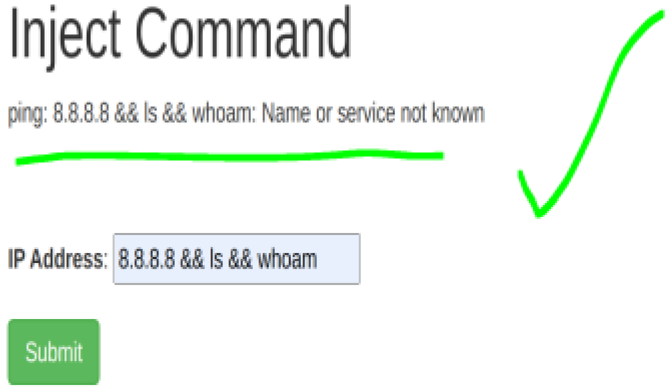
Figure 13. IP Address Without Proper Validation/Sanitization



```
//Bad codes
//String[] command = {"/bin/bash", "-c", "ping -c 3 " + ip_address};
//Process proc = Runtime.getRuntime().exec(command);

//added**
ProcessBuilder processBuilder = new ProcessBuilder();
processBuilder.command("ping", "-c", "3", ip_address);
Process proc = processBuilder.start();
proc.waitFor();
```

Figure 14. Implementing New Codes in Placed Of Vulnerbale Codes

As depicted in **Figure 15** below, the vulnerability has been successfully addressed through input validation and sanitization. Attempting to input 'ping 8.8.8.8 && ls &&whoamI' into the designated field and clicking the submit button now returns the response 'ping 8.8.8.8 && ls &&whoamI: Name or service not known' to the attacker. This outcome indicates the prevention of command injection for this application, ensuring a more secure environment.



Inject Command

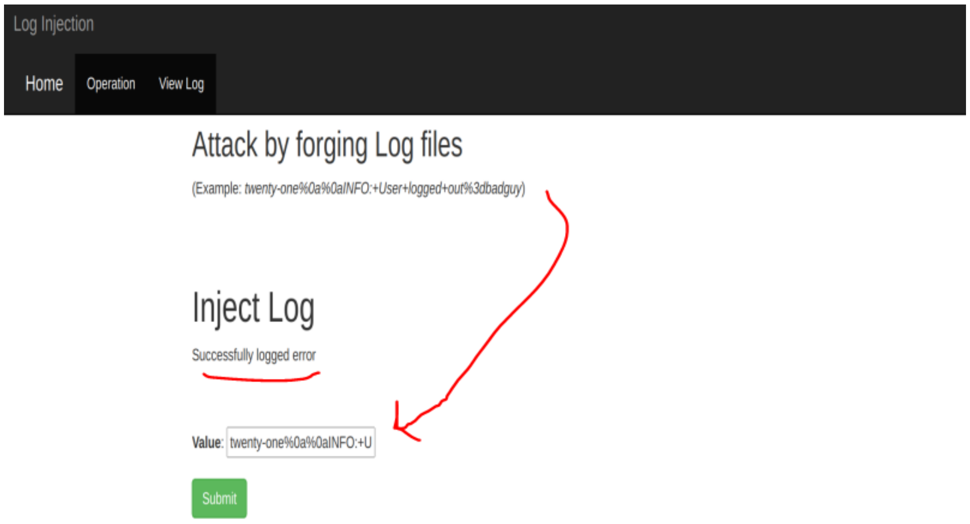
ping: 8.8.8.8 && ls && whoam: Name or service not known

IP Address: 8.8.8.8 && ls && whoam

Figure 15. Vulnerability Fixed

5.0 Log Forging Injection

In the realm of computer security, logs serve as detailed records of events within a computer system, playing a crucial role in monitoring, diagnosing problems, and investigating security incidents. However, when logs are tampered with by attackers using malicious code, Log Forging occurs. Log Forging vulnerabilities denote weaknesses in a system that empower hackers to manipulate and create false log entries. **Figure 16** illustrates an instance where an attacker can input malicious code, like 'twenty-one%0a%0aINFO:+User+logged+out%3dbadguy', into the 'Value' input box and, upon clicking the submit button, successfully engage in Log Forging. To mitigate log forging vulnerabilities, it is important to implement input validation, adhere to secure logging practices, and restrict log writing access to only authorized users. In the following section, we will undertake log forging mitigation to address and prevent these vulnerabilities effectively.



Attack by forging Log files

(Example: twenty-one%0a%0aINFO:+User+logged+out%3dbadguy)

Inject Log

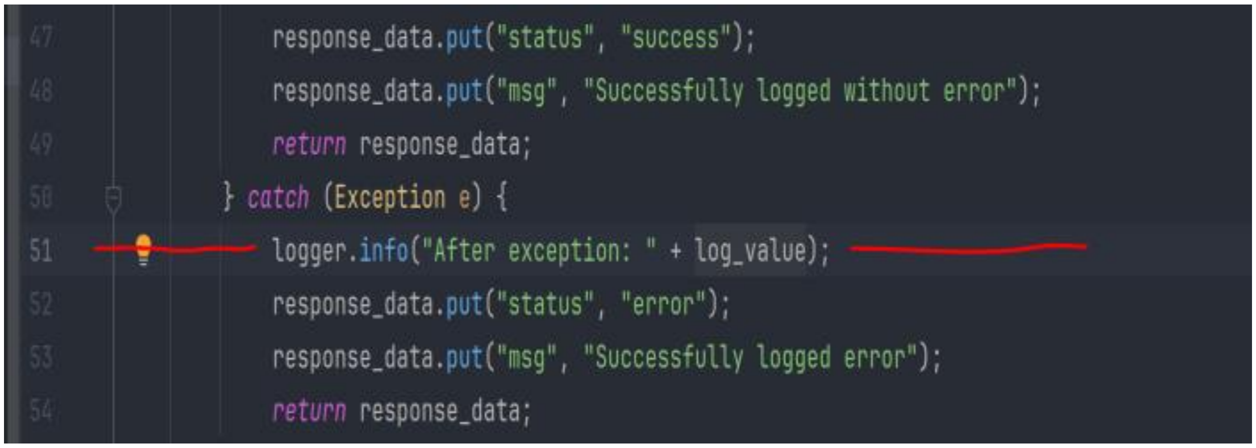
Successfully logged error

Value: twenty-one%0a%0aINFO:+U

Figure 16. Log Forging

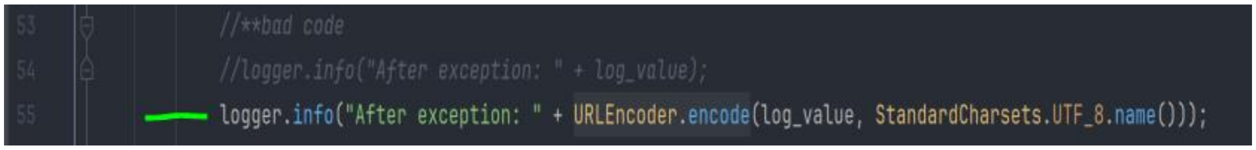
5.1 Log Forging Mitigation

Within the log_injected() method of the Log_injectionController.java class, a potential log forging vulnerability is evident. The 'log_value' parameter lacks proper validation or sanitization, as illustrated in **Figure 17**. It is directly concatenated with the string 'After exception' and logged without appropriate validation. This vulnerability opens the door for attackers to manipulate the 'log_value' parameter, injecting malicious content into the log and leading to log forging. To address and mitigate this vulnerability, we will implement sanitization for the 'log_value' by employing 'java.net.URLEncoder.encode()' as depicted in **Figure 18**. This encoding process ensures that the 'log_value' is encoded before inclusion in the log, thereby reducing the risk of log forging vulnerabilities.



```
            response_data.put("status", "success");
            response_data.put("msg", "Successfully logged without error");
            return response_data;
        } catch (Exception e) {
            logger.info("After exception: " + log_value);
            response_data.put("status", "error");
            response_data.put("msg", "Successfully logged error");
            return response_data;
```

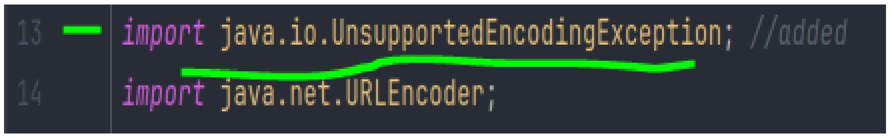
Figure 17. Log Forging Vulnerabilities



```
            //**bad code
            //logger.info("After exception: " + log_value);
            logger.info("After exception: " + URLEncoder.encode(log_value, StandardCharsets.UTF_8.name()));
```

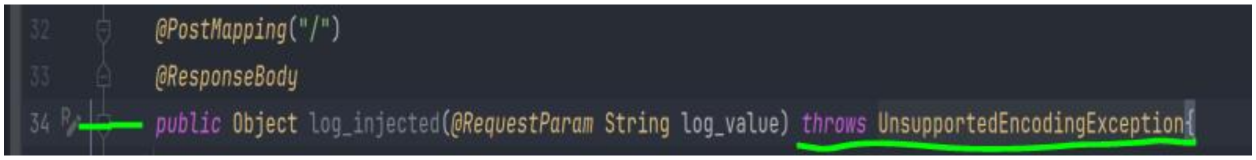
Figure 18. URLEncorder.encode().

To fix this vulnerability, we will implement changes to specific lines of code. On line 55 (refer to **Figure 18**), we will introduce 'URLEncoder.encode()'. Additionally, we will import 'UnsupportedEncodingException' (see **Figure 19**). Finally, we will incorporate an 'UnsupportedEncodingException' handler on line 34 (refer to **Figure 20**). These modifications collectively fortify the code and mitigate the risk of log forging vulnerabilities.



```java
import java.io.UnsupportedEncodingException; //added
import java.net.URLEncoder;
```

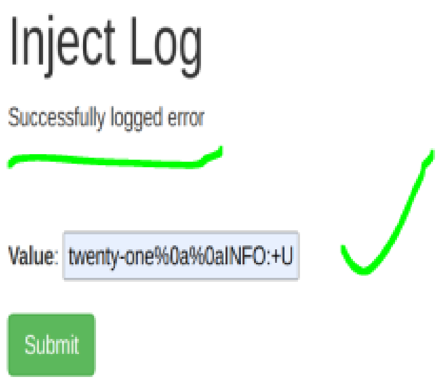
Figure 19. Import UnsupportedEncodingException.



```java
32      @PostMapping("/")
33      @ResponseBody
34      public Object log_injected(@RequestParam String log_value) throws UnsupportedEncodingException{
```

Figure 20. Throws UnsupportedEncodingException.

Upon implementing these code changes, an attacker will no longer be able to successfully log anything, as illustrated in **Figure 21** below. Examining **Figure 22**, which displays the Custom_log_file.log class, confirms the success of the mitigation. The log forging vulnerability has been effectively addressed and resolved.

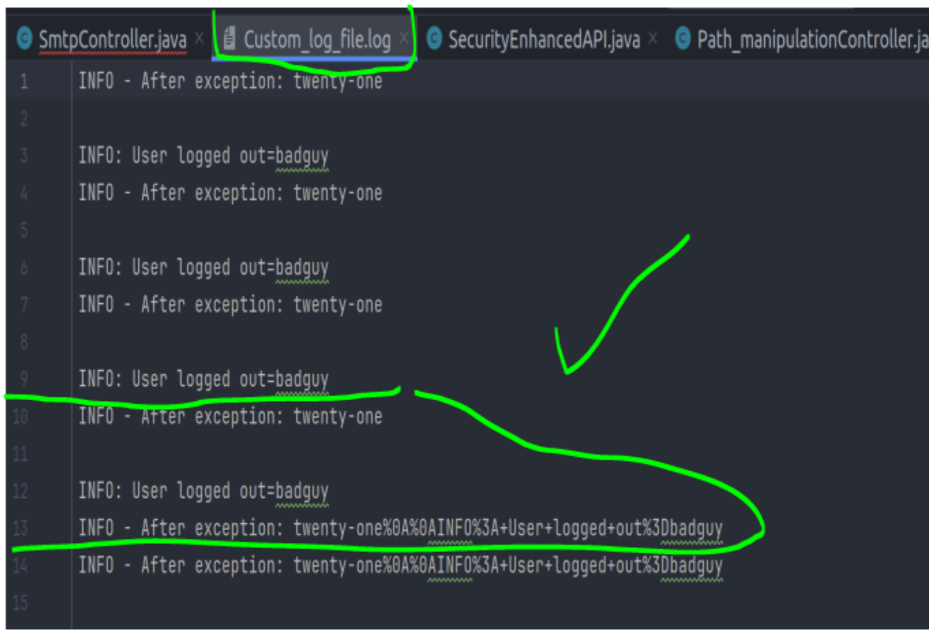


Inject Log

Successfully logged error

Value: twenty-one%0a%0aINFO:+U

Figure 21. Error Message


SmtpController.java | Custom_log_file.log | SecurityEnhancedAPI.java | Path_manipulationController.ja

```
INFO - After exception: twenty-one

INFO: User logged out=badguy
INFO - After exception: twenty-one

INFO: User logged out=badguy
INFO - After exception: twenty-one

INFO: User logged out=badguy
INFO - After exception: twenty-one

INFO: User logged out=badguy
INFO - After exception: twenty-one%0A%0AINFO%3A+User+logged+out%3Dbadguy
INFO - After exception: twenty-one%0A%0AINFO%3A+User+logged+out%3Dbadguy
```

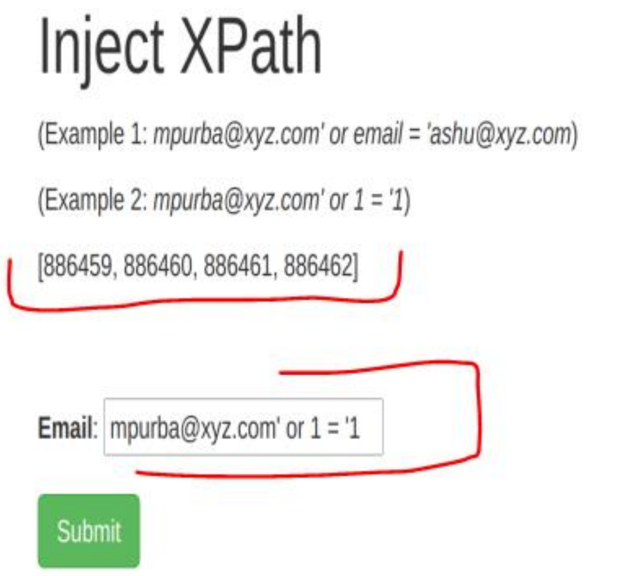
Figure 22. Log forgery vulnerability mitigation successful.

6.0 XPath Injection

XPath injection is an attack that capitalizes on vulnerabilities in web applications utilizing XPath (XML Path Language). This typically occurs when a web application incorporates user-supplied data to construct an XPath Query for XML data without adequate validation and sanitization of the input. In such an injection, attackers can manipulate input data to insert malicious XPath queries into the application's code. This manipulation allows attackers to potentially access and/or modify sensitive information, as they circumvent authentication mechanisms and elevate their privileges on the website.

6.1 XPath Injection Mitigation

As seen in **Figure 23** below, an attacker can input ‘mpurba@xyz.com’ or 1 = ‘1’ into the ‘Email’ input box. Upon executing the script, sensitive information is revealed, with user IDs being displayed in this instance.


Inject XPath

(Example 1: *mpurba@xyz.com' or email = 'ashu@xyz.com*)

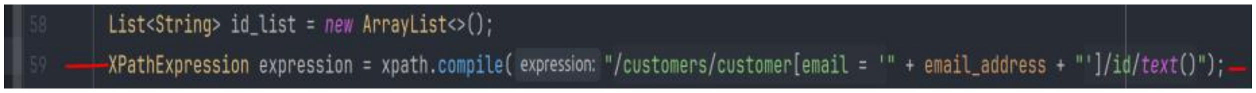
(Example 2: *mpurba@xyz.com' or 1 = '1*)

[886459, 886460, 886461, 886462]

Email: mpurba@xyz.com' or 1 = '1

Figure 23. XPath Injection Vulnerability

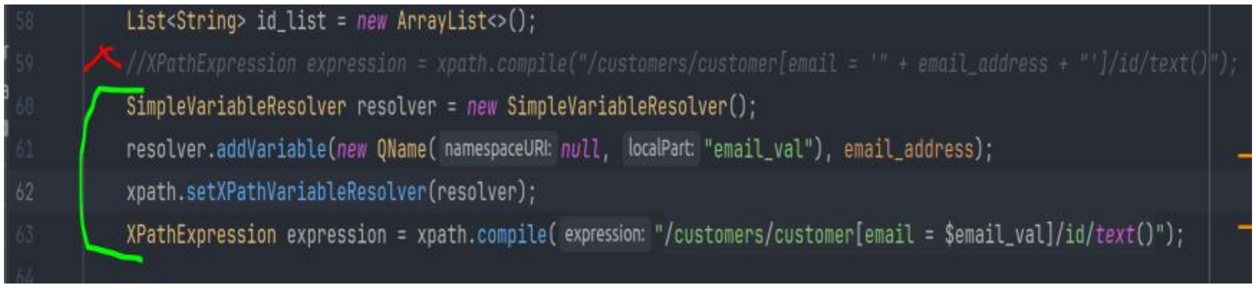
In line 59 of the XPath_injectionController.java class, we find the vulnerable code (see **Figure 24**). To mitigate the XPath vulnerability, we need to add some new lines of codes.



```
58    List<String> id_list = new ArrayList<>();
59    XPathExpression expression = xpath.compile( expression: "/customers/customer[email = '" + email_address + "']/id/text()");
```

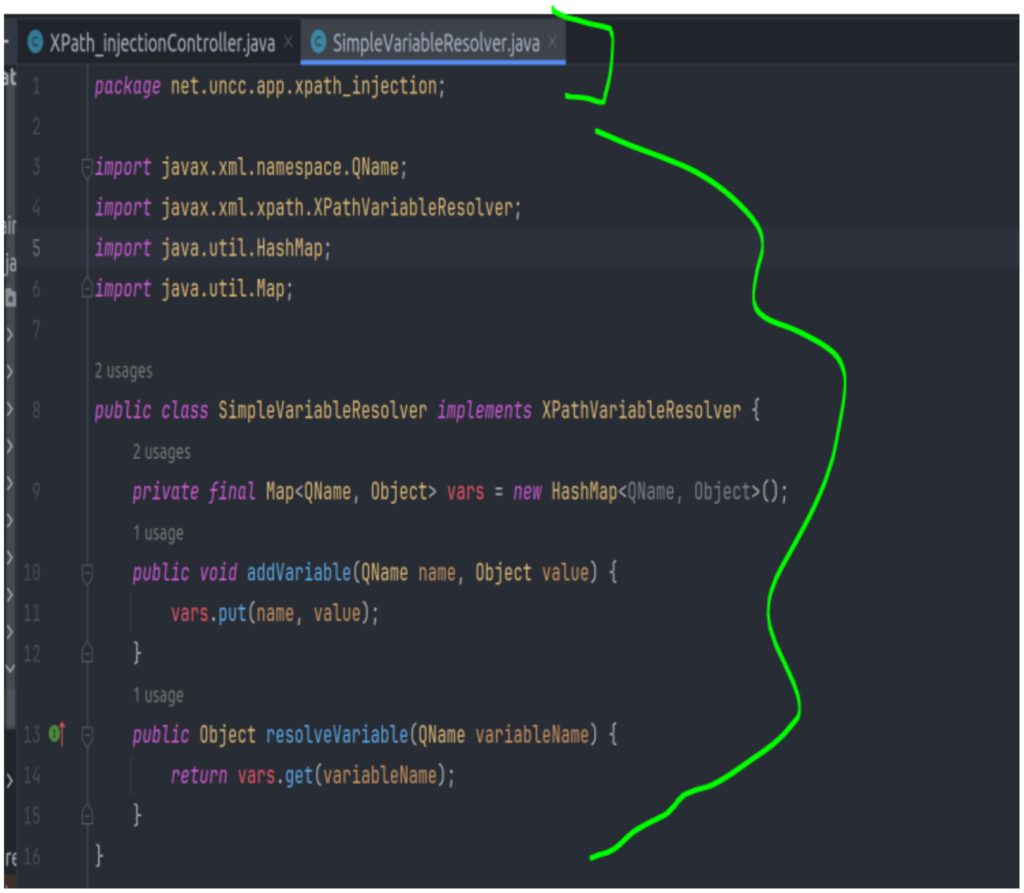
Figure 24. Vulnerable code

To address XPath vulnerabilities, we will update our codes. Specifically, we will comment out the line containing the problematic code and introduce four new lines (refer to **Figure 25**). Additionally, a new class named ‘SimpleVariableResolver.java’ will be added (refer to **Figure 26**), and an 'import...QName' will be included on line 22 (see **Figure 27**). These modifications collectively enhance the codebase and contribute to the mitigation of XPath vulnerabilities.



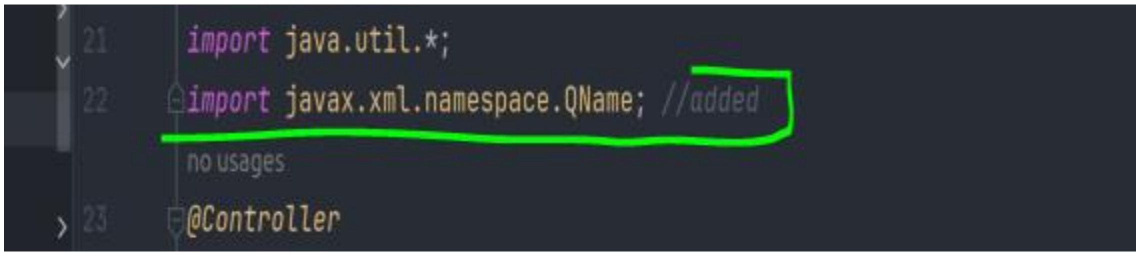
```
58    List<String> id_list = new ArrayList<>();
59    //XPathExpression expression = xpath.compile("/customers/customer[email = '" + email_address + "']/id/text()");
60    SimpleVariableResolver resolver = new SimpleVariableResolver();
61    resolver.addVariable(new QName( namespaceURI: null,  localPart: "email_val"), email_address);
62    xpath.setXPathVariableResolver(resolver);
63    XPathExpression expression = xpath.compile( expression: "/customers/customer[email = $email_val]/id/text()");
64
```

Figure 25. New lines of codes



```
XPath_injectionController.java    SimpleVariableResolver.java

package net.uncc.app.xpath_injection;

import javax.xml.namespace.QName;
import javax.xml.xpath.XPathVariableResolver;
import java.util.HashMap;
import java.util.Map;

2 usages
public class SimpleVariableResolver implements XPathVariableResolver {
    2 usages
    private final Map<QName, Object> vars = new HashMap<QName, Object>();
    1 usage
    public void addVariable(QName name, Object value) {
        vars.put(name, value);
    }
    1 usage
    public Object resolveVariable(QName variableName) {
        return vars.get(variableName);
    }
}
```

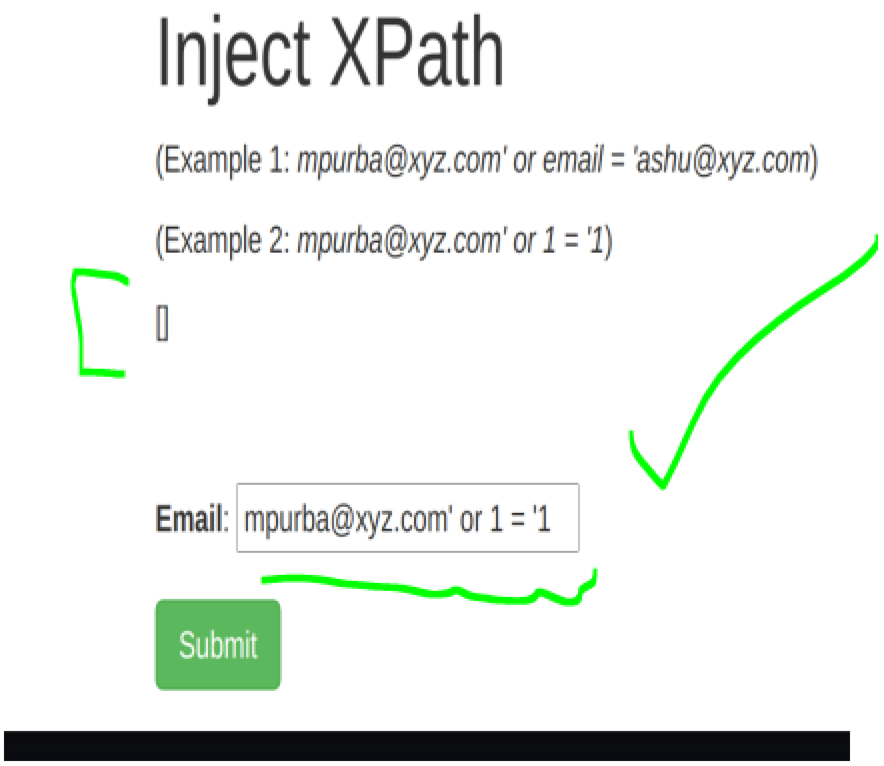
Figure 26. New SimpleVariableResolver.java class



```
import java.util.*;
import javax.xml.namespace.QName; //added
no usages
@Controller
```

Figure 27. Add import….QName

New, when an attacker tried to inject as script into the input box, all they will see is '[]' instead of the user's ID (see **Figure 28**). We have successfully implemented XPath injection mitigation.



Inject XPath

(Example 1: *mpurba@xyz.com' or email = 'ashu@xyz.com*)

(Example 2: *mpurba@xyz.com' or 1 = '1*)

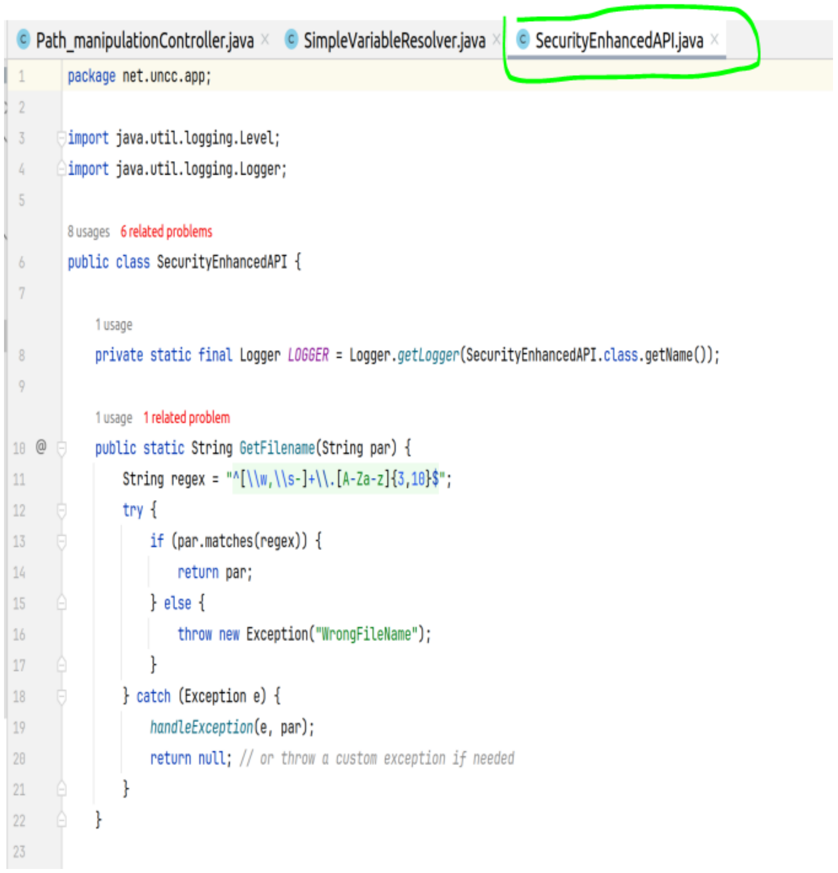
[]

Email: mpurba@xyz.com' or 1 = '1

Figure 28. XPath Injection Fixed

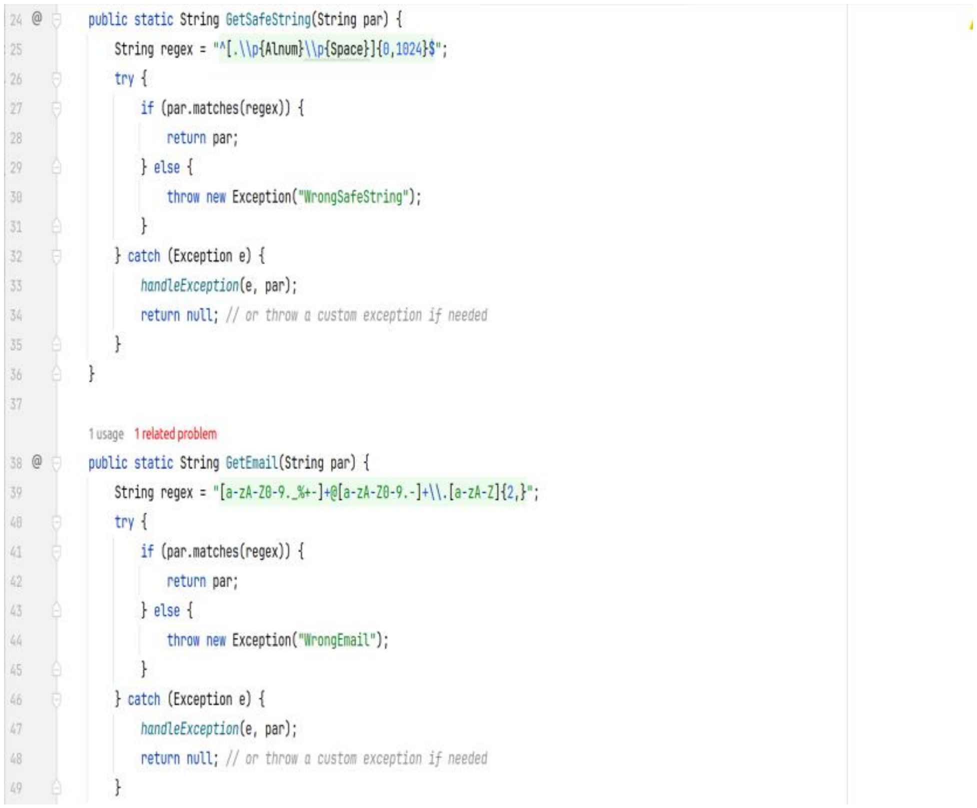
7.0 Security Enhanced API Class

Below, we will create a new Security Enhanced API class with three methods in the net.uncc.app of the java.mian folder of our codes. We will also add exceptions handlers to log error message, encode string input, and reject any malicious input. Refer to **Figure 29**, **Figure 30,** and **Figure 30A** to view implementations.


Path_manipulationController.java
SimpleVariableResolver.java
SecurityEnhancedAPI.java
8 usages 6 related problems
1 usage
1 usage 1 related problem

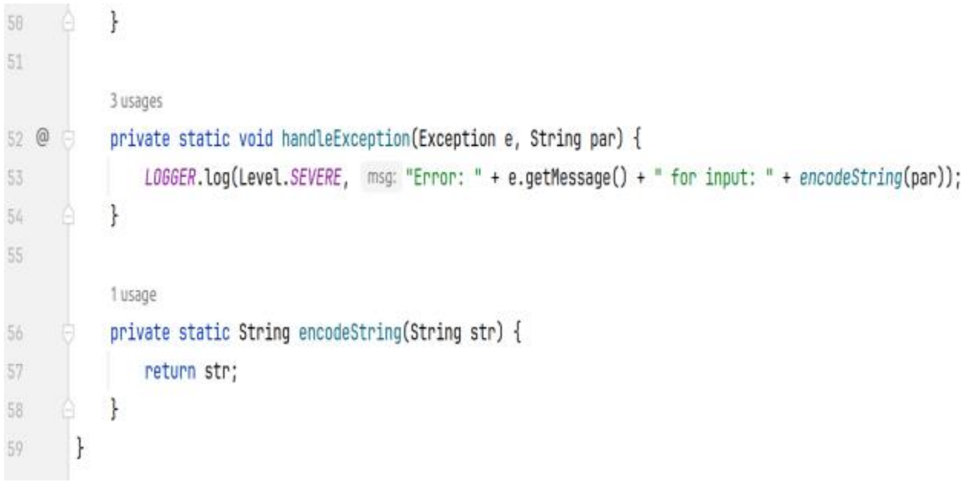
```java
package net.uncc.app;

import java.util.logging.Level;
import java.util.logging.Logger;

public class SecurityEnhancedAPI {

    private static final Logger LOGGER = Logger.getLogger(SecurityEnhancedAPI.class.getName());

    public static String GetFilename(String par) {
        String regex = "^[\\w,\\s-]+\\.[A-Za-z]{3,10}$";
        try {
            if (par.matches(regex)) {
                return par;
            } else {
                throw new Exception("WrongFileName");
            }
        } catch (Exception e) {
            handleException(e, par);
            return null; // or throw a custom exception if needed
        }
    }

```

Figure 29. SecurityEnhancedAPI.java class Part 1



```java
public static String GetSafeString(String par) {
    String regex = "^[.\\p{Alnum}\\p{Space}]{0,1024}$";
    try {
        if (par.matches(regex)) {
            return par;
        } else {
            throw new Exception("WrongSafeString");
        }
    } catch (Exception e) {
        handleException(e, par);
        return null; // or throw a custom exception if needed
    }
}

1 usage   1 related problem
public static String GetEmail(String par) {
    String regex = "[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}";
    try {
        if (par.matches(regex)) {
            return par;
        } else {
            throw new Exception("WrongEmail");
        }
    } catch (Exception e) {
        handleException(e, par);
        return null; // or throw a custom exception if needed
    }
```

Figure 30. SecurityEnhancedAPI.java class Part 2

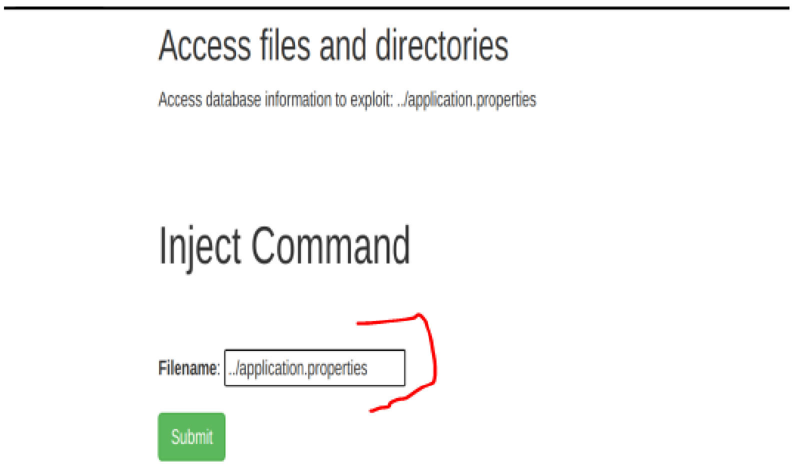


```java
    }

    3 usages
    private static void handleException(Exception e, String par) {
        LOGGER.log(Level.SEVERE,  msg: "Error: " + e.getMessage() + " for input: " + encodeString(par));
    }

    1 usage
    private static String encodeString(String str) {
        return str;
    }
}
```

Figure 30A. SecurityEnhancedAPI.java class Part 3

8.0 Path Manipulation

Path Manipulation vulnerabilities occur when an application permits user input to influence file or resource paths, potentially leading to unauthorized access to files or directories on a server. In **Figure 31** and **Figure 32**, it is evident that entering '../application.properties' into the input box allows an attacker to display the properties, illustrating a Path Manipulation vulnerability. To mitigate Path Manipulation vulnerabilities, secure measures must be implemented to prevent unauthorized access through manipulative techniques. The Path_manipuationController.java class reveals various vulnerabilities, and we will employ Path Manipulation Mitigations to fix these issues.

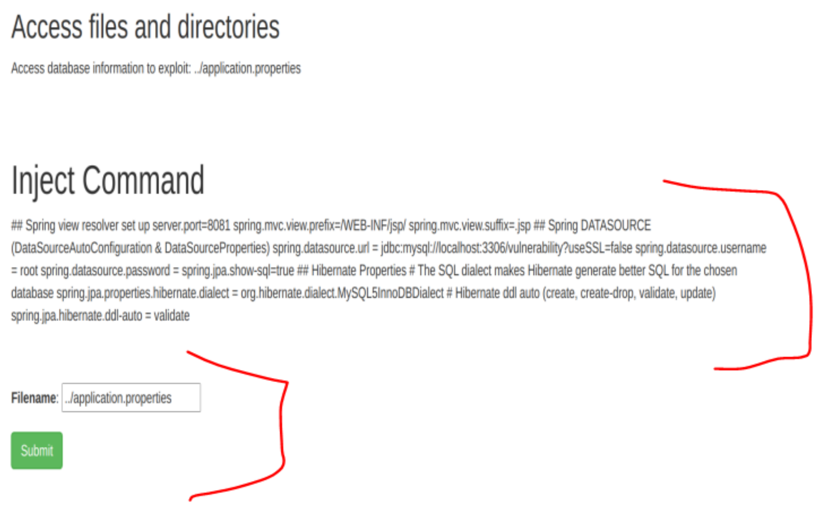

Access files and directories

Access database information to exploit: ../application.properties

Inject Command

Filename: ../application.properties

Figure 31. Input of ../application.properites



Access files and directories

Access database information to exploit: ../application.properties

Inject Command

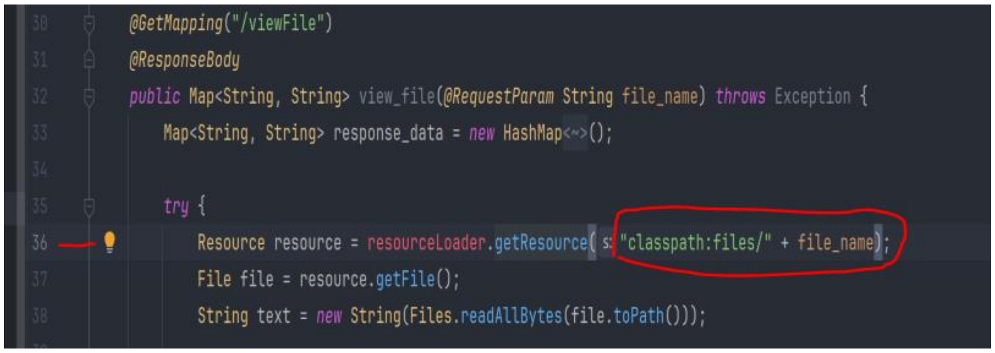
Spring view resolver set up server.port=8081 spring.mvc.view.prefix=/WEB-INF/jsp/ spring.mvc.view.suffix=.jsp ## Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties) spring.datasource.url = jdbc:mysql://localhost:3306/vulnerability?useSSL=false spring.datasource.username = root spring.datasource.password = spring.jpa.show-sql=true ## Hibernate Properties # The SQL dialect makes Hibernate generate better SQL for the chosen database spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect # Hibernate ddl auto (create, create-drop, validate, update) spring.jpa.hibernate.ddl-auto = validate

Filename: ../application.properties

Figure 32. Display Properties of Application

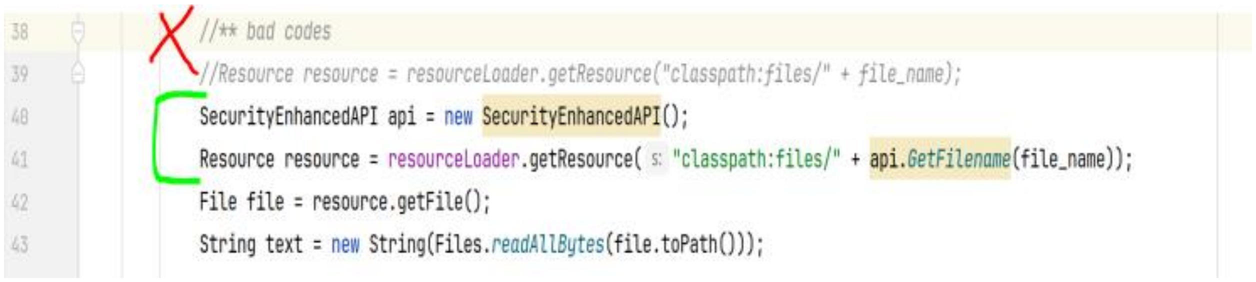
8.1 Path Manipulation Mitigation

Within the view_file() method of the Path_manipulationController.java class, concerns arise at line 36, where 'file_name' is concatenated with 'classpath:files/' (refer to **Figure 33**). This poses a security risk, as a malicious user could potentially manipulate the 'file_name' parameter to access sensitive files in directories. An attacker might input something like "../../very_sensitive_info_file" into the file name. To address this, we will implement a solution by validating and sanitizing the 'file_name' parameter before its use. This involves commenting out line 37 and introducing lines 38 and 39 (see **Figure 34**) to the application, effectively resolving the issue.



```
30  @GetMapping("/viewFile")
31  @ResponseBody
32  public Map<String, String> view_file(@RequestParam String file_name) throws Exception {
33      Map<String, String> response_data = new HashMap<~>();
34
35      try {
36          Resource resource = resourceLoader.getResource( s: "classpath:files/" + file_name);
37          File file = resource.getFile();
38          String text = new String(Files.readAllBytes(file.toPath()));
```

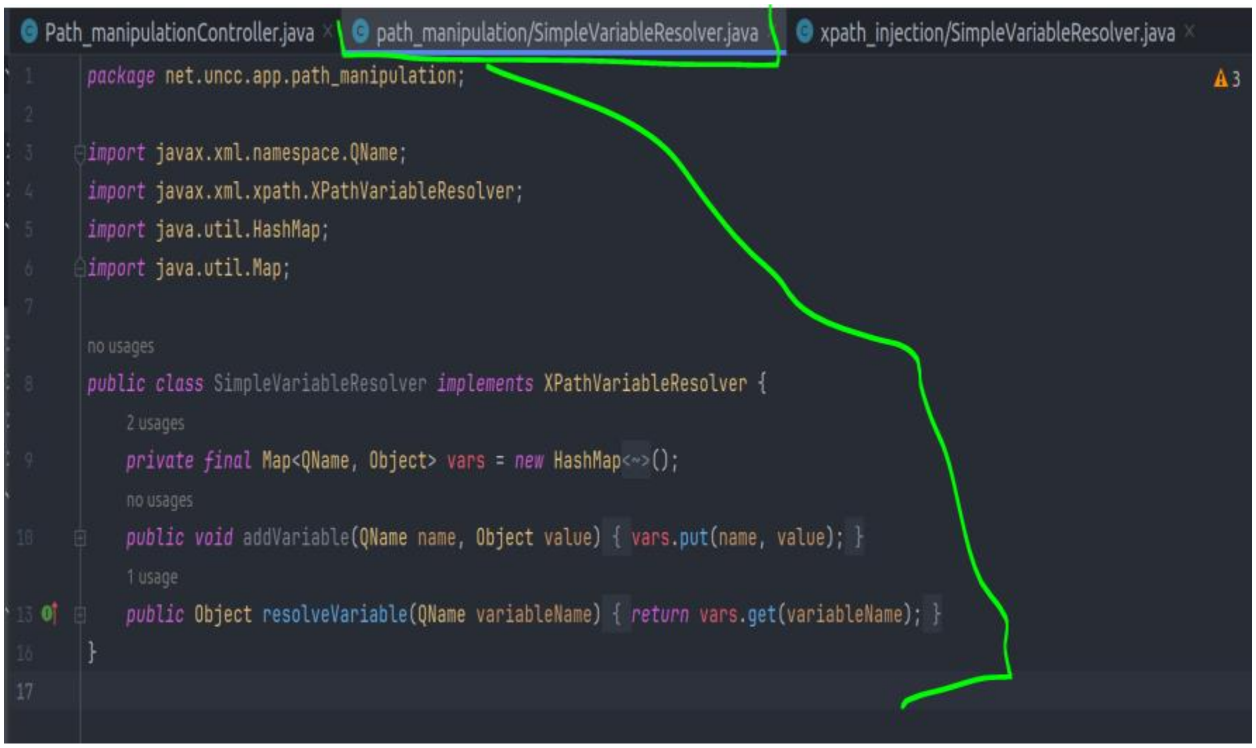
Figure 33. Filename Concatenated with Class Path



```java
//** bad codes
//Resource resource = resourceLoader.getResource("classpath:files/" + file_name);
SecurityEnhancedAPI api = new SecurityEnhancedAPI();
Resource resource = resourceLoader.getResource( s: "classpath:files/" + api.GetFilename(file_name));
File file = resource.getFile();
String text = new String(Files.readAllBytes(file.toPath()));
```

Figure 34. Comment out line and add two new lines of code**.**

We will also input these codes into the SimpleVariableResolver.java class for the xpath_injection.java class (see **Figure 35**) to help fix the issue.

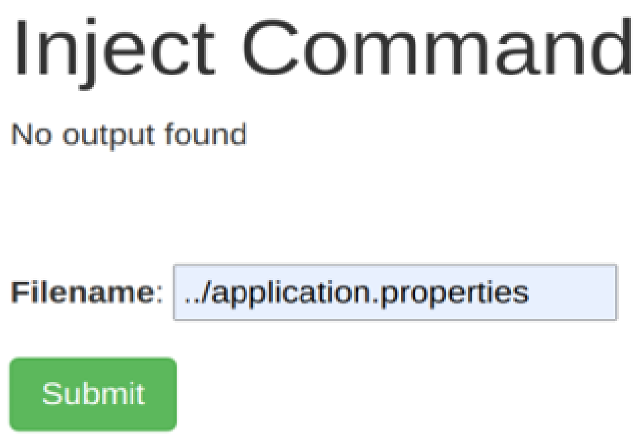


Path_manipulationController.java | path_manipulation/SimpleVariableResolver.java | xpath_injection/SimpleVariableResolver.java

```java
package net.uncc.app.path_manipulation;

import javax.xml.namespace.QName;
import javax.xml.xpath.XPathVariableResolver;
import java.util.HashMap;
import java.util.Map;

no usages
public class SimpleVariableResolver implements XPathVariableResolver {
    2 usages
    private final Map<QName, Object> vars = new HashMap<~>();
    no usages
    public void addVariable(QName name, Object value) { vars.put(name, value); }
    1 usage
    public Object resolveVariable(QName variableName) { return vars.get(variableName); }
}
```

Figure 35. Input codes into SimpleVariableResolver.java

As illustrated in Figure 36, when the attacker inputs '../application.properties' and clicks the submit button, the message "No output found" appears. This serves as a clear demonstration of the effectiveness of the path manipulation mitigation.


Inject Command

No output found

Filename: ../application.properties

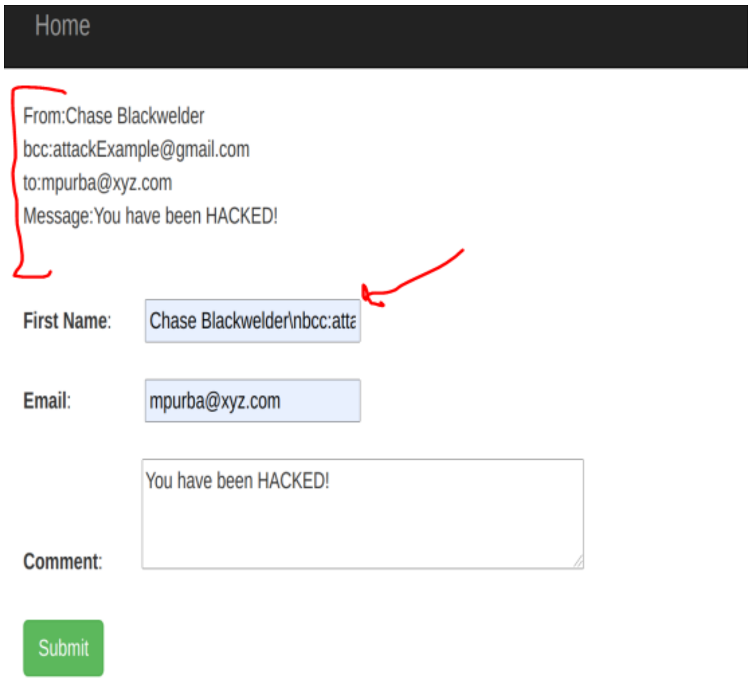
Figure 36. Path Manipulation Fixed

9.0 Simple Mail Transfer Protocol (SMTP) Mitigation

Simple mail transfer protocol (SMTP) is a widely employed method for transmitting email messages across the internet. However, SMTP is susceptible to various vulnerabilities that malicious individuals can exploit. One common vulnerability is email spoofing, where an attacker forges the sender's email address to appear as the legitimate sender. An attacker can harness this for activities such as phishing, spamming, and the distribution of malware.

9.1 SMTR Mitigation

If a hacker inputs "Chase Blackwelder\nbcc:attackExample@gmail.com" into the 'First Name' input box and clicks the submit button, as depicted in **Figure 36**, the output allows them to inject commands to someone or redirect it to a different destination. To address this vulnerability, it is crucial to rectify the code in the backend.



From:Chase Blackwelder
bcc:attackExample@gmail.com
to:mpurba@xyz.com
Message:You have been HACKED!

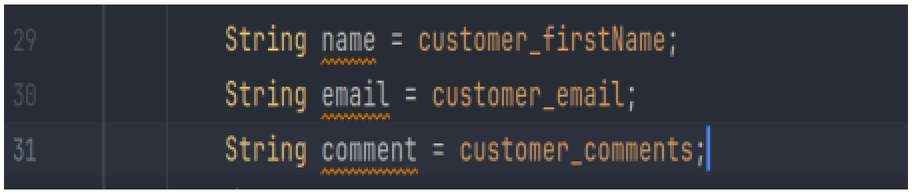
First Name: Chase Blackwelder\nbcc:atta

Email: mpurba@xyz.com

Comment: You have been HACKED!

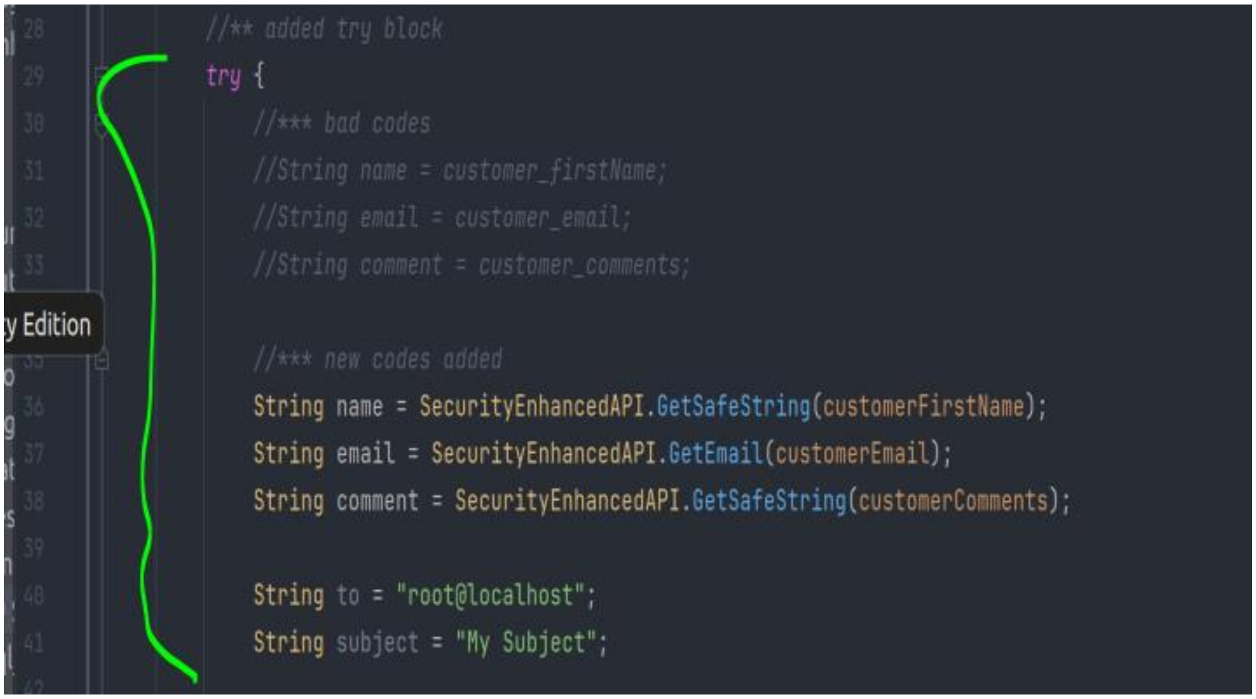
Figure 36. SMTP vulnerability

In lines 29–31 of the SmtpController.java class, vulnerable codes are evident (see **Figure 37**). The program accepts user inputs without proper sanitization. To address this, we will comment out lines 31–33 and introduce new lines of code and imports, as shown in **Figure 38** and **39**. The GetSafeString function sanitizes and validates the input, ensuring it only contains alphanumeric characters, spaces, and dots. The GetEmail function validates the email address using regex patterns, ensuring the variable contains the user's email address. GetSafeString guarantees that the string consists only of valid characters, and it also sanitizes and validates the comments. These codes are enclosed within a try block, with a catch block added from line 56 to 59 to handle any validation exceptions (see **Figure 40**)



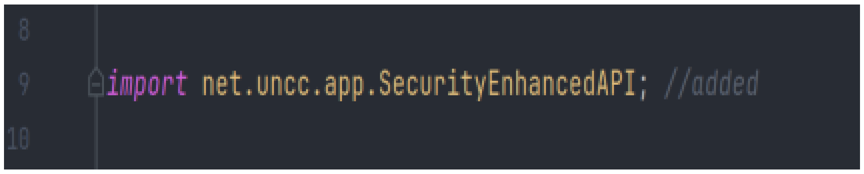
```
29        String name = customer_firstName;
30        String email = customer_email;
31        String comment = customer_comments;
```

Figure 37. SMTP vulnerabilities



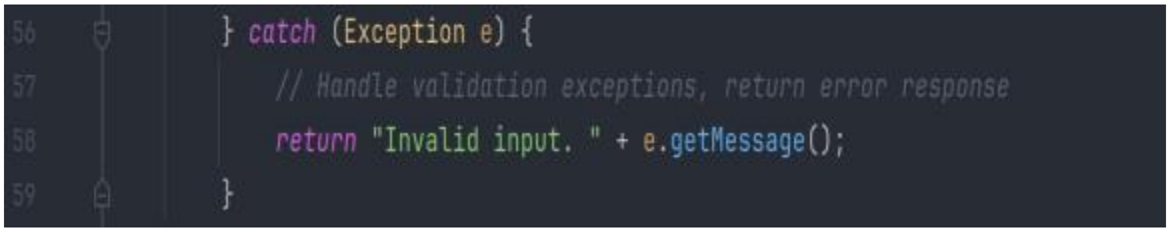
```
//** added try block
try {
    //*** bad codes
    //String name = customer_firstName;
    //String email = customer_email;
    //String comment = customer_comments;

    //*** new codes added
    String name = SecurityEnhancedAPI.GetSafeString(customerFirstName);
    String email = SecurityEnhancedAPI.GetEmail(customerEmail);
    String comment = SecurityEnhancedAPI.GetSafeString(customerComments);

    String to = "root@localhost";
    String subject = "My Subject";
```

Figure 38. Comment lines and new added codes



```
import net.uncc.app.SecurityEnhancedAPI; //added
```

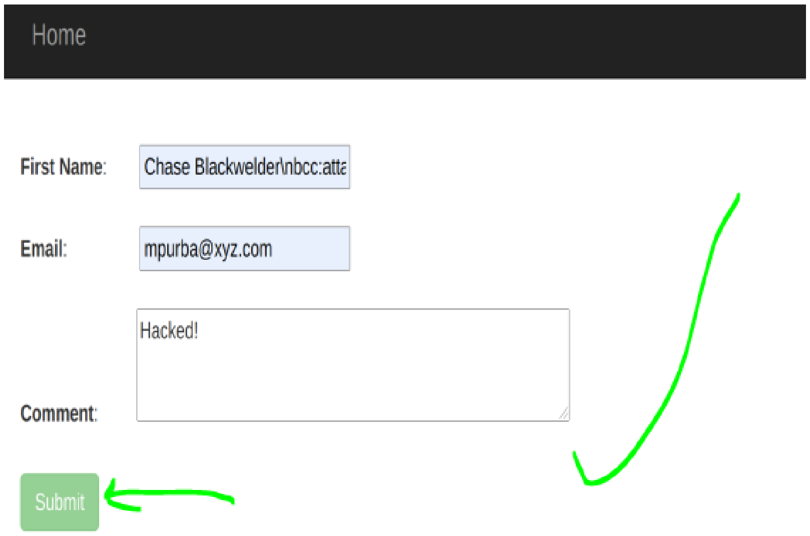
Figure 39. Adding 'Import….SecurityEnhancedAPI'



```
} catch (Exception e) {
    // Handle validation exceptions, return error response
    return "Invalid input. " + e.getMessage();
}
```

Figure 40. Catch Exception

As we can see now, if a hacker tries to attack again, the submit button will turn light green (see **Figure 41**). An attacker will not be able to click it again. The vulnerability has successfully been mitigated.



First Name: Chase Blackwelder\nbcc:atta

Email: mpurba@xyz.com

Comment: Hacked!

Figure 41. Submit button turns light green.