Vulnerability Assessment and Systems Assurance Report - Project Pentesting

Project Pentesting

Amanuel Haile

November 3rd, 2024

VULNERABILITY ASSESSMENT AND SYSTEMS ASSURANCE REPORT

TABLE OF CONTENTS

1.0 General Information

1.1 Purpose

The purpose of this vulnerability assessment and penetration test is to analyze the security of this application. The objective of this report is to discover and demonstrate the exploitation of various security vulnerabilities, specifically SQL Injection, XSS, XPath Injection, Command Injection, Path Manipulation, Log Injection, SMTP header injection, and improper Exception Handling.

2.0 SQL Injection

An SQL injection is a vulnerability that allows an attacker to manipulate queries to bypass authentication, retrieve sensitive information, or even modify data. In this application, SQL injection vulnerabilities were identified in the address update function. Specifically, the address update feature allowed an attacker to change another user's information by injecting SQL code.

2.1 SQL Injection - Vulnerability

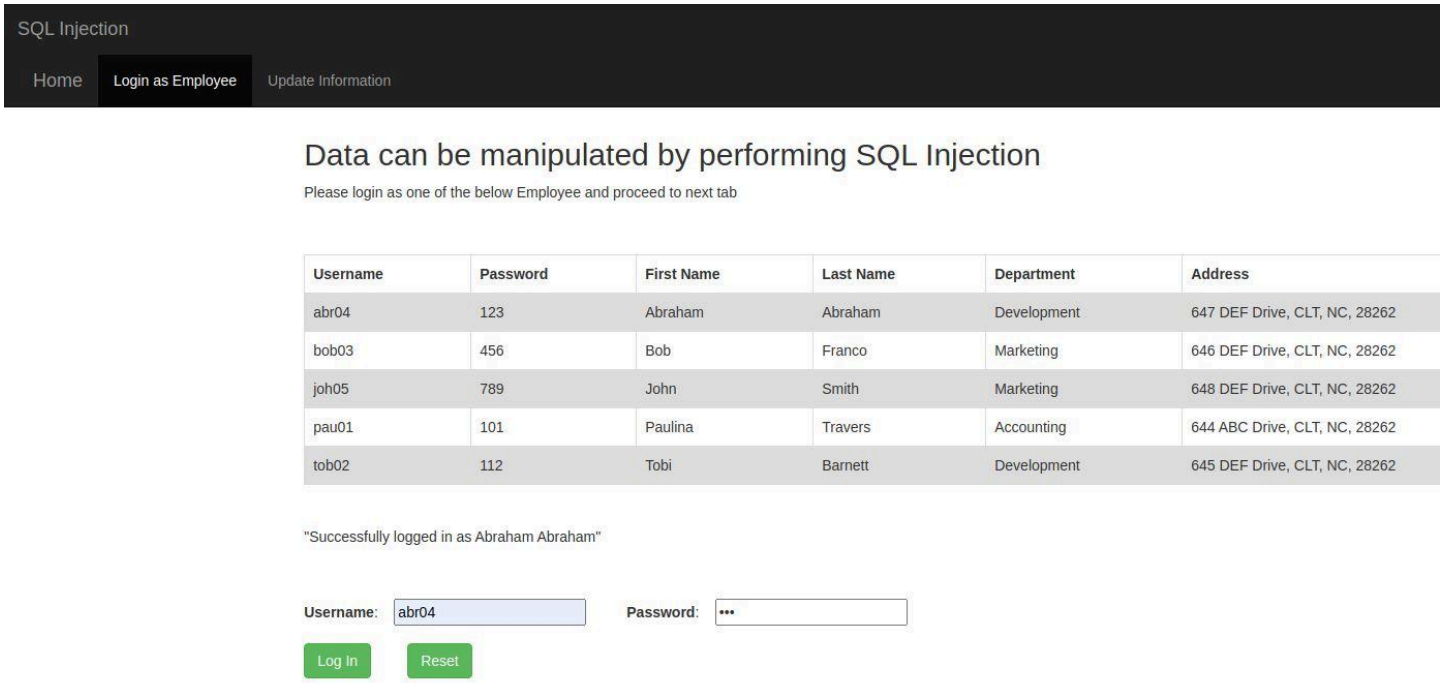
The vulnerability was discovered during an authorized test where, after logging in as one employee, it was possible to update the address of another employee by injecting SQL code in the updated_address field. For example, while logged in as employee_username, an attacker could modify the address of another employee, bob03, by entering the following input:
888 DEF Drive, CLT, NC, 28262' WHERE username = 'bob03'#

This input effectively closed the original SQL statement, appended a new condition targeting the bob03 username, and used # to comment out any remaining SQL code. As a result, the update query executed with the injected SQL segment, changing bob03's address.

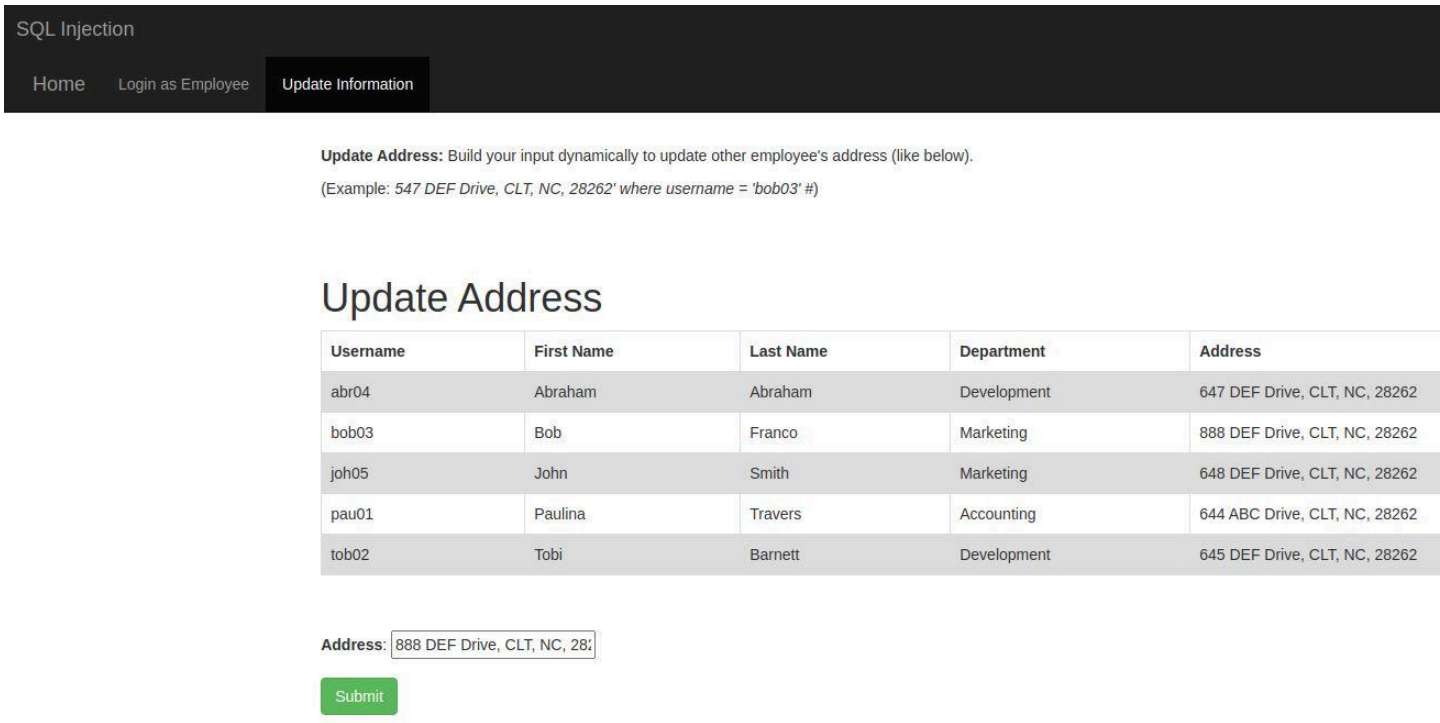
The original query was constructed as follows, directly embedding updated_address without parameterization:
String updateQuery = "UPDATE Employees SET address = '" + updated_address + "' WHERE username = '" + loggedInUser + "'";

This allowed attackers to control the SQL query structure through the updated_address input.

Here is a demonstration of this vulnerability:
Here is the login page where the attacker uses the credentials of one employee:



One the next tab, the attacker cna update the address of the user they logged in as, but if SQL injection is utilized, the attacker change the address of another user in the database:



2.2 SQL Injection - Fix

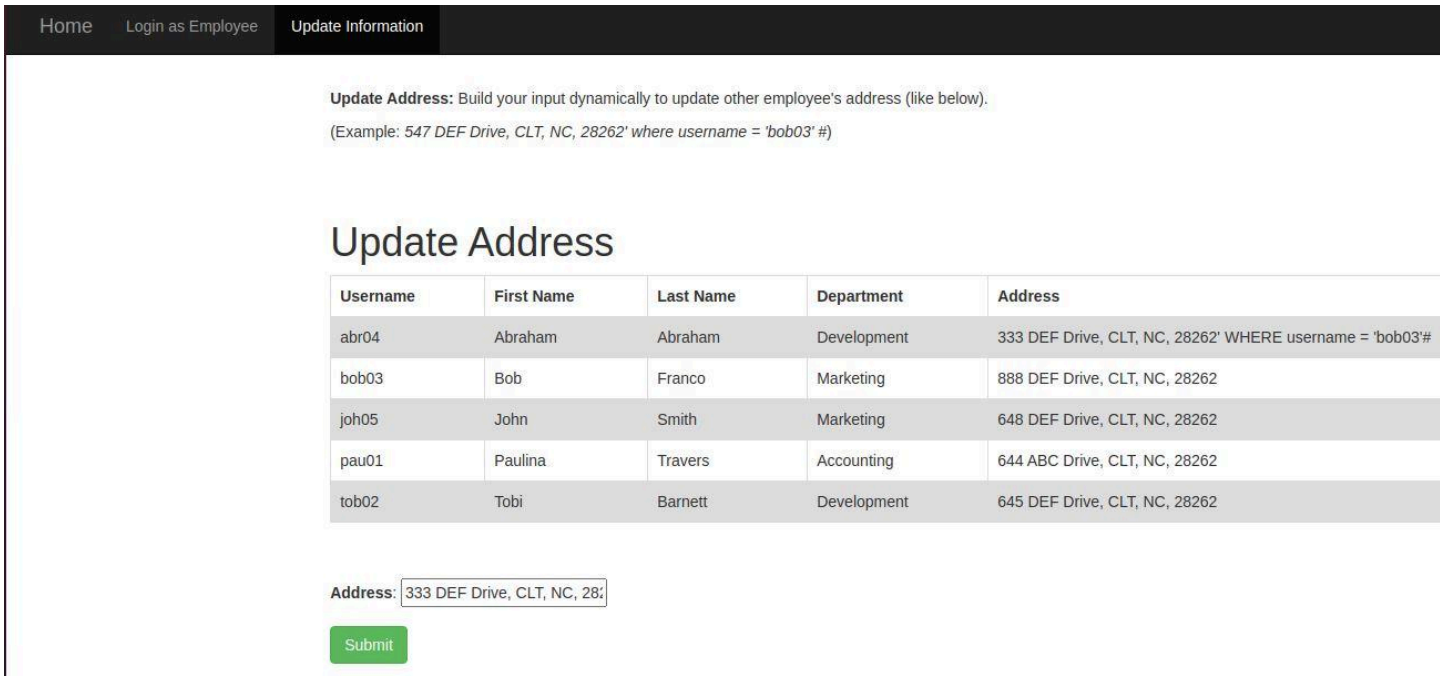
Here is the specific code that is causing the vulnerability:

```
104                 String updateQuery = "UPDATE Employees SET address = '" + updated_address + "' WHERE username = '" + loggedInUser + "'";
105                 // change in 'jdbcTemplate.update' function by passing parameters so that dynamic input will not harm database.
106                 updatedEmpInfo = jdbcTemplate.update(updateQuery);
```

To mitigate this issue, we applied parameterized queries, which separate SQL logic from user inputs by using ? placeholders. The updated code now constructs the query as follows:

```
104                 String updateQuery = "UPDATE Employees SET address = ? WHERE username = ?";
105                 // change in 'jdbcTemplate.update' function by passing parameters so that dynamic input will not harm database.
106                 updatedEmpInfo = jdbcTemplate.update(updateQuery, updated_address, loggedInUser);
```

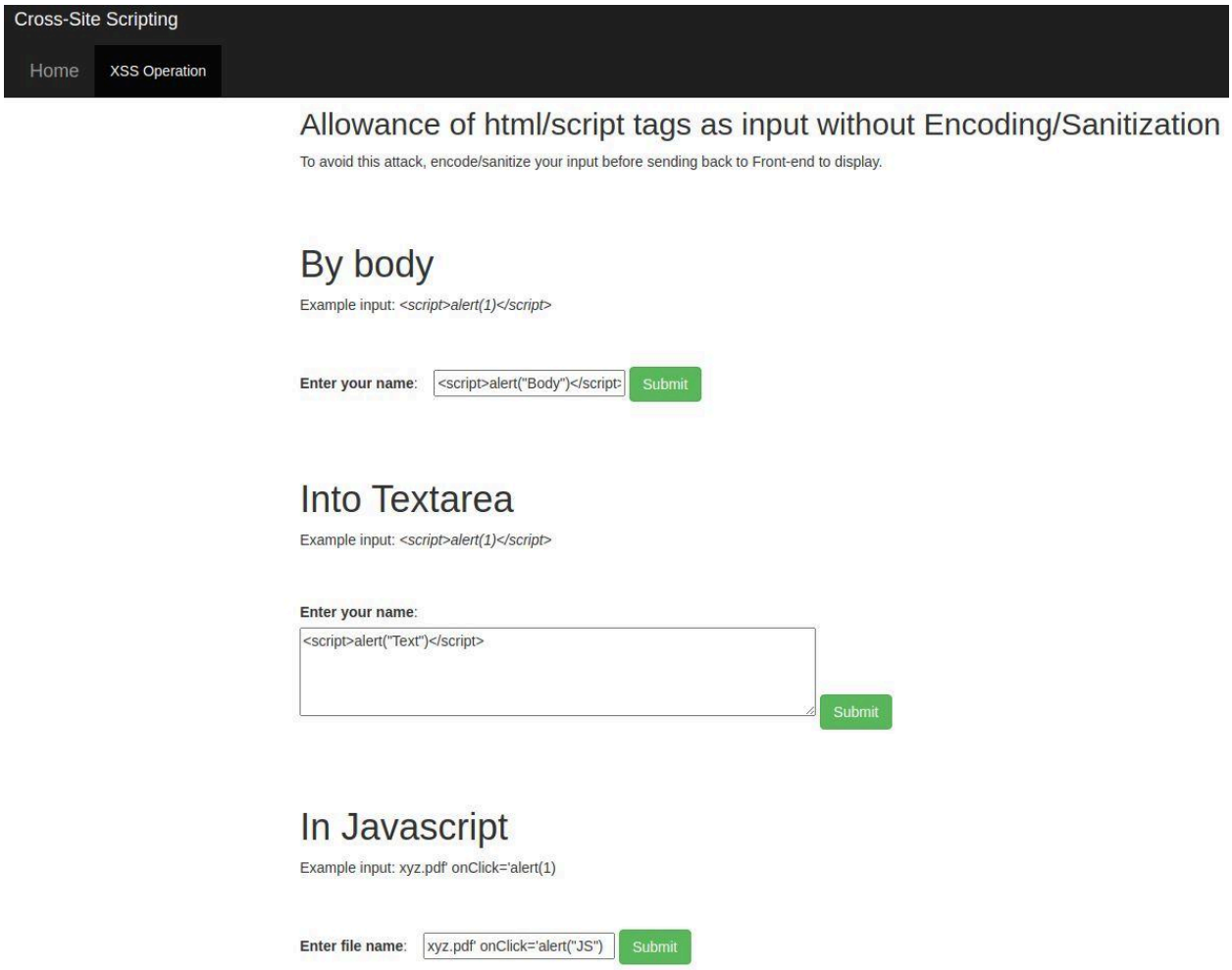
By using parameterized queries, any injected SQL is treated strictly as a literal string, preventing the input from altering the intended SQL logic. As a result, only the logged-in employee's address can be updated:



3.0 XSS

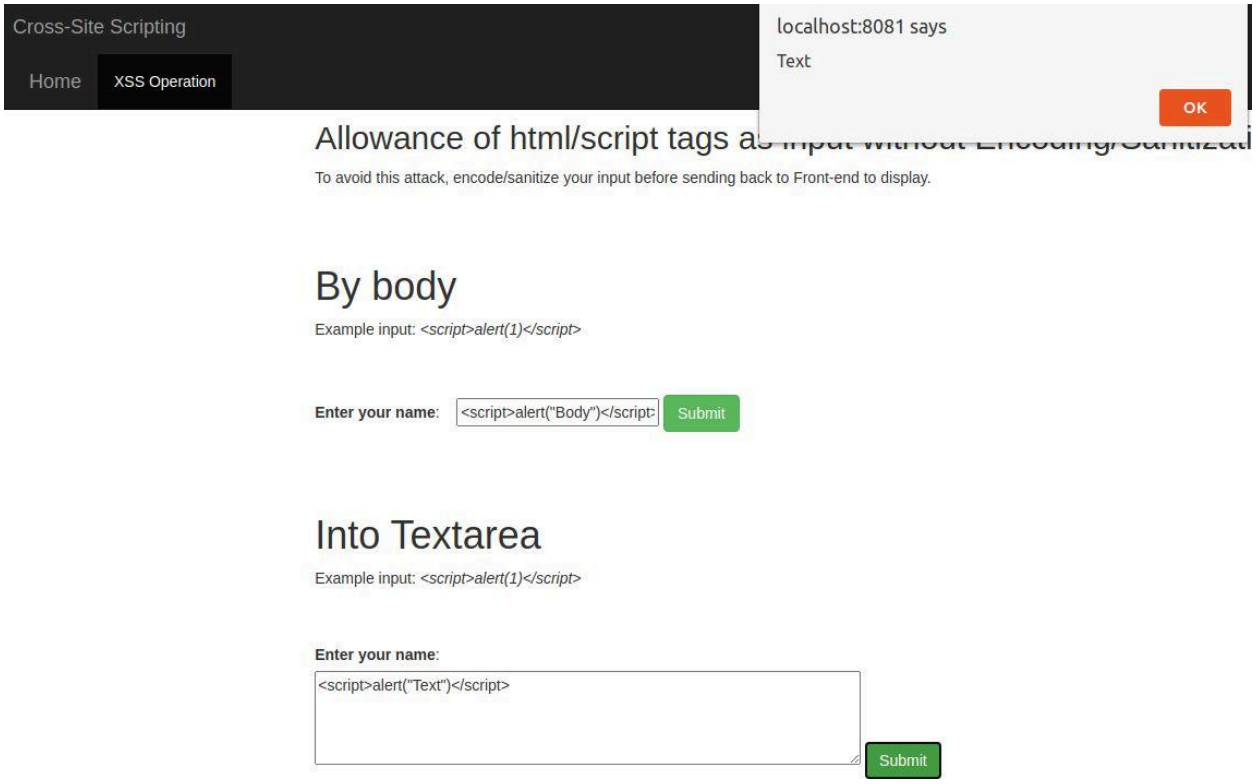
XSS is a vulnerability that allows attackers to inject malicious scripts into web applications. These scripts are then executed in the browser of users who view the infected content. The vulnerability arises when an application does not properly sanitize user input before rendering it on web pages. XSS vulnerabilities can be exploited to steal sensitive information such as session cookies, redirect users to malicious websites, or execute other malicious actions within a victim's browser.

3.1 XSS - Vulnerability



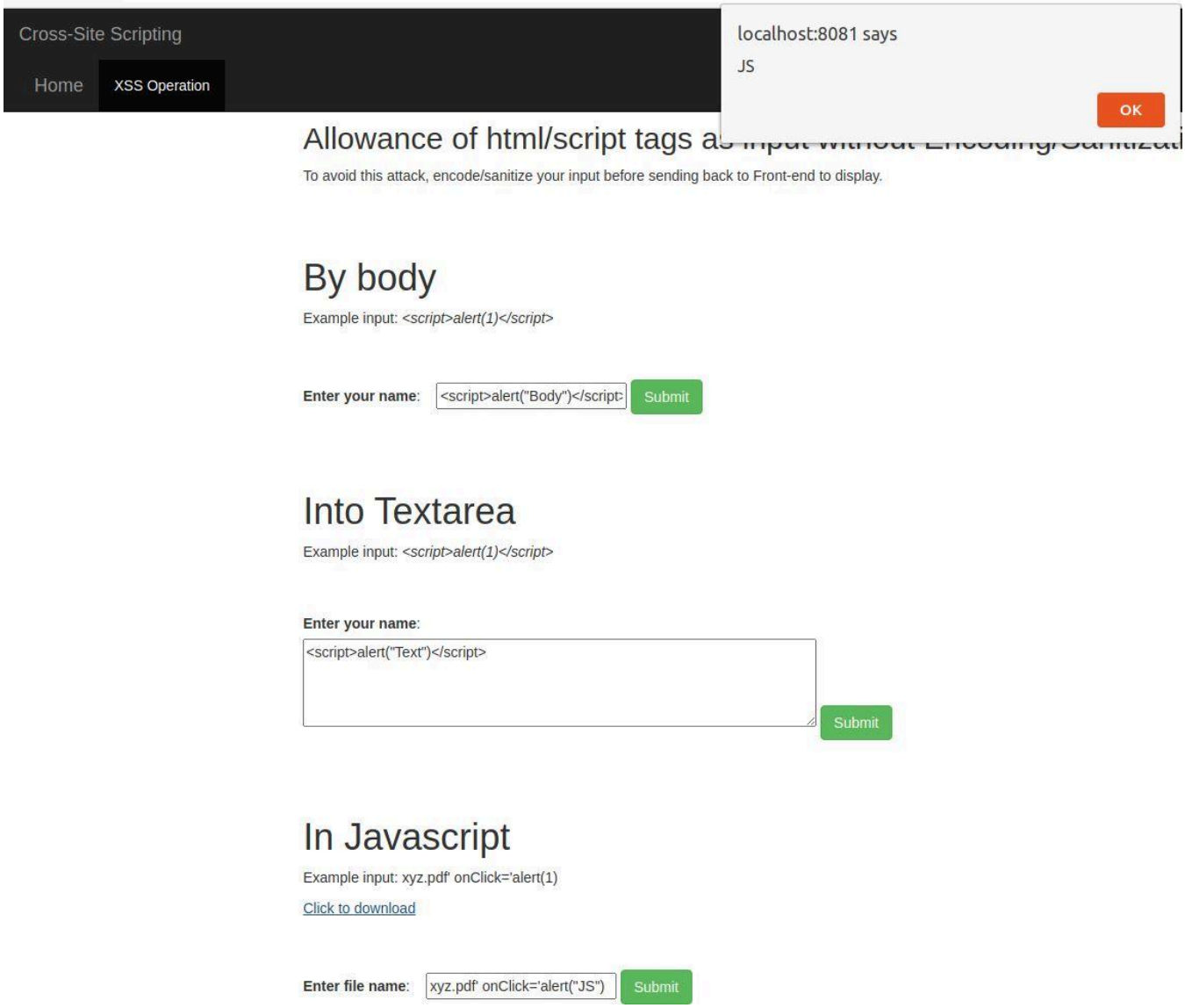
XSS in Body Tag:
Example Input: <script>alert("Body")</script>
Upon submission, this code triggered an alert with the message "Body," indicating that the application rendered the script directly in the browser's body without encoding, allowing it to execute.

XSS in Textarea:
Example Input: <script>alert("Text")</script>
When submitted in the textarea, this code executed an alert with the message "Text." The script was not sanitized or encoded, allowing it to run directly.



XSS in JavaScript:
Example Input: xyz.pdf' onClick='alert("JS")
When submitted as the file name, this input triggered a JavaScript alert with the message "JS" when clicked, demonstrating that the application embedded the JavaScript in the rendered HTML without sanitizing it.

(When clicked):



3.2 XSS - Fix

Here is the current code:

```
17      @GetMapping("/body_xss")
18      @ResponseBody
19      public String body_xss(@RequestParam String body_tagVal) throws Exception {
20          return body_tagVal;
21      }
22
23      @GetMapping("/textarea_xss")
24      @ResponseBody
25      public Object textarea_xss(@RequestParam String textarea_tagVal) throws Exception {
26          return textarea_tagVal;
27      }
28
29      @GetMapping("/js_xss")
30      @ResponseBody
31      public Object js_xss(@RequestParam String js_tagVal) throws Exception {
32          return js_tagVal;
33      }
```

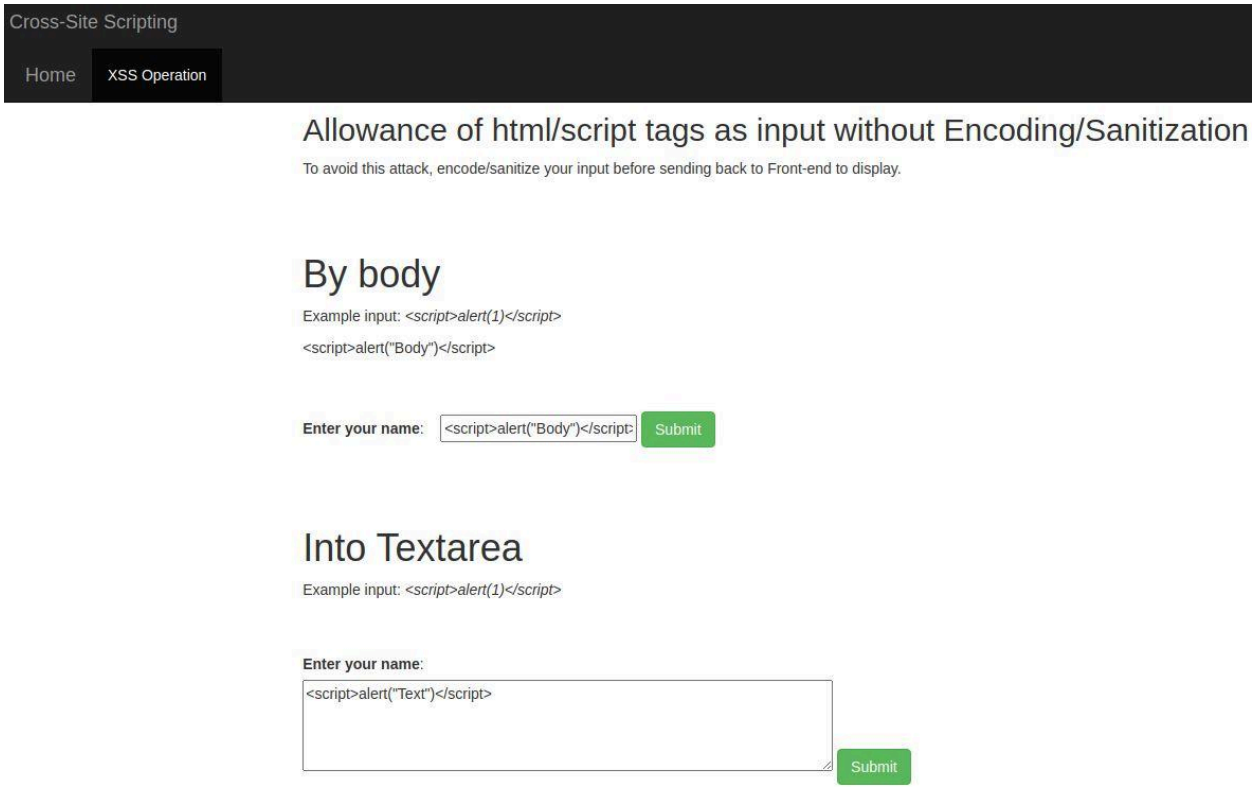
And here is the updated code:

```
17      @GetMapping("/body_xss")
18      @ResponseBody
19      public String body_xss(@RequestParam String body_tagVal) throws Exception {
20          return escapeHtml(body_tagVal);
21      }
22
23      @GetMapping("/textarea_xss")
24      @ResponseBody
25      public Object textarea_xss(@RequestParam String textarea_tagVal) throws Exception {
26          return escapeHtml(textarea_tagVal);
27      }
28
29      @GetMapping("/js_xss")
30      @ResponseBody
31      public Object js_xss(@RequestParam String js_tagVal) throws Exception {
32          String safeString = escapeHtml(js_tagVal);
33          safeString = safeString.replace("\"", "\\\"").replace("'", "\\'");
34          return safeString;
35      }
36
37  }
```

There are two main fixes:
HTML Encoding: The escapeHtml function is used to convert special characters in the input to HTML entities. This ensures that any HTML tags are rendered as text rather than executed as code.

JavaScript Context Handling: In the js_xss method, after escaping HTML, the code replaces single and double quotes with their escaped versions (\' and \"). This prevents the injection of JavaScript code by ensuring that any quotes within the string do not allow the injected script to break out of its intended context.

Screenshots showing the successful fix:
HTML Encoding:



The script is displayed as plain text for the body. For the Textarea nothing is shown and noting pops up.

JavaScript Context Handling:

When the user clicks on the download link, it redirects the user to this page:

3.2 XSS - Incorrect Encoding

To demonstrate the impact of incorrect encoding, I'll modify the code by replacing the correct HTML encoding (escapeHtml) with incorrect URL encoding (URLEncoder.encode). This will show how URL encoding fails to prevent XSS in an HTML context
Here's the updated XssController code with the incorrect encoding function:

```
@GetMapping("/body_xss")
@ResponseBody
public String body_xss(@RequestParam String body_tagVal) throws Exception {
    return URLEncoder.encode(body_tagVal, StandardCharsets.UTF_8.toString());
}

@GetMapping("/textarea_xss")
@ResponseBody
public Object textarea_xss(@RequestParam String textarea_tagVal) throws Exception {
    return URLEncoder.encode(textarea_tagVal, StandardCharsets.UTF_8.toString());
}

@GetMapping("/js_xss")
@ResponseBody
public Object js_xss(@RequestParam String js_tagVal) throws Exception {
    String safeString = URLEncoder.encode(js_tagVal, StandardCharsets.UTF_8.toString());
    safeString = safeString.replace("\"", "\\\"").replace("'", "\\'");
    return safeString;
}
```

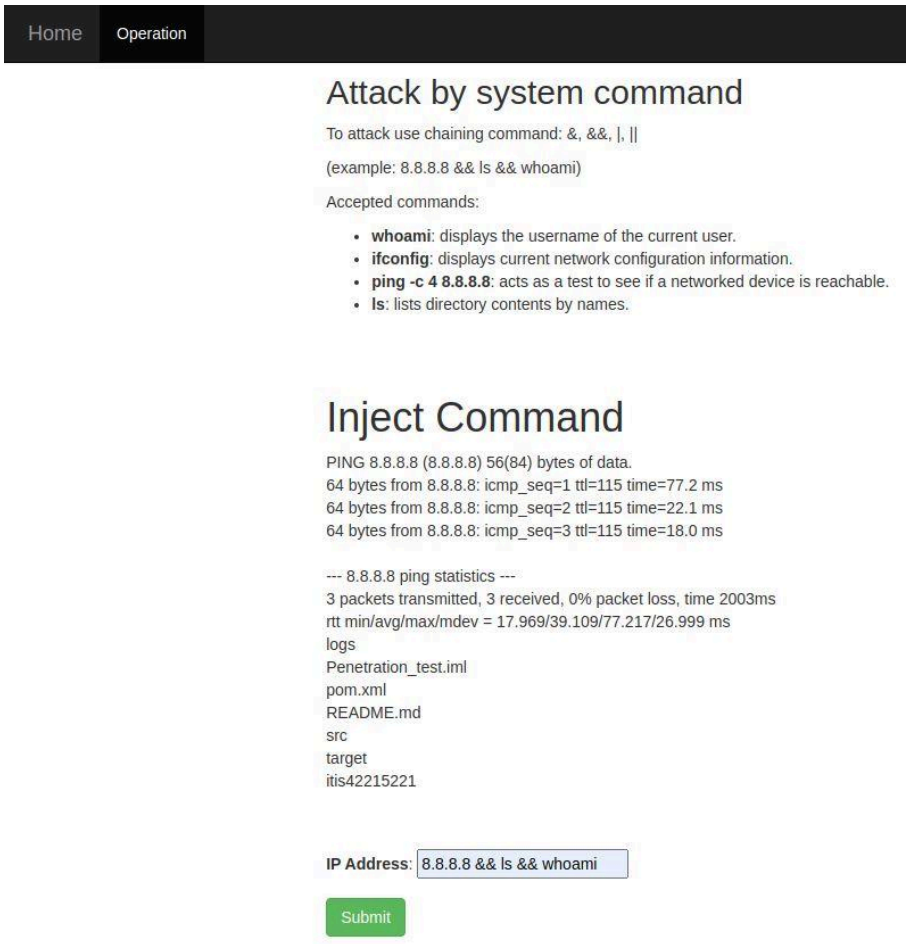
4.0 Command injection

A Command Injection vulnerability allows an attacker to execute unauthorized commands on the server by injecting additional commands into the ip_address parameter. Since user input is directly appended to a system command without sufficient validation, an attacker could enter specially made input to execute multiple chained commands, potentially compromising sensitive server information.

4.1 Command injection - Vulnerability

The vulnerability can be demonstrated by inputting a command that chains additional system commands. For example, entering the following input in the IP Address field:
8.8.8.8 && ls && whoami

This input not only runs the ping command as expected but also executes ls to list files and whoami to reveal the current user. Here is the output:



4.2 Command injection - Fix

The following line of code in the command_injected method adds user input (ip_address) directly to the command string, making it susceptible to injection attacks:

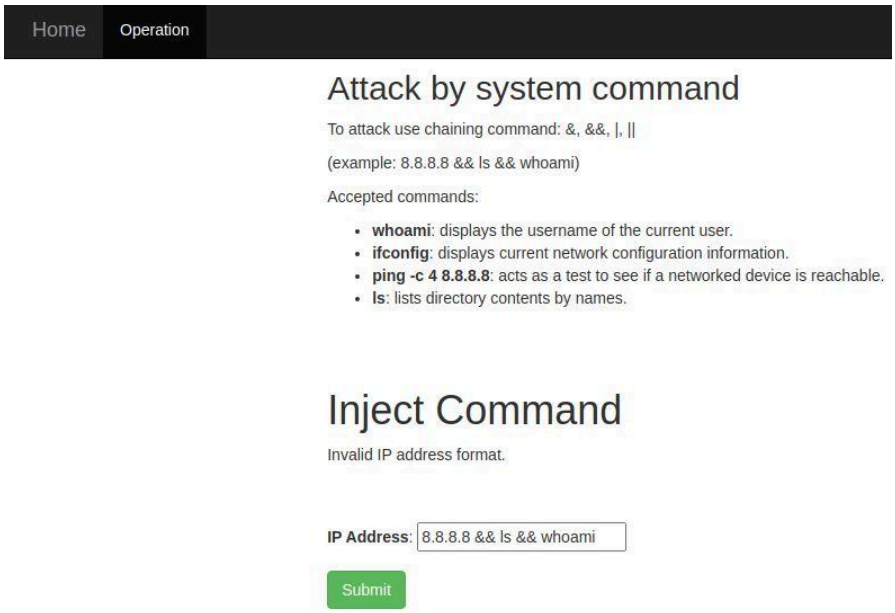
```
23          try {
24              String output = "";
25              String[] command = {"/bin/bash", "-c", "ping -c 3 " + ip_address};
26              Process proc = Runtime.getRuntime().exec(command);
27              proc.waitFor();
```

Here's the updated code that fixes the vulnerability:

```
19      @PostMapping("/output/")
20      @ResponseBody
21      public Object command_injected(@RequestParam String ip_address) {
22          Map<String, String> response_data = new HashMap<>();
23
24          if (!ip_address.matches("^([0-9]{1,3}\\.){3}[0-9]{1,3}$")) {
25              response_data.put("status", "error");
26              response_data.put("msg", "Invalid IP address format.");
27              return response_data;
28          }
29
30          try {
31              String output = "";
32
33              ProcessBuilder processBuilder = new ProcessBuilder("ping", "-c", "3", ip_address);
34              Process proc = processBuilder.start();
35              proc.waitFor();
```

First we use a regular expression to allow only valid IPv4 addresses, preventing special characters and command operators from being processed. We then use ProcessBuilder to separate command arguments, which prevents input from being interpreted as part of a single command string.

After implementing this fix, the application will then reject the command as seen here:



5.0 Path Manipulation

Path traversal vulnerabilities occur when an application does not restrict user input and allows access to unauthorized files outside of the intended directory. In this case, an attacker can enter a path like ../application.properties to read sensitive configuration files.

5.1 Path Manipulation - Vulnerability

The Path_manipulationController is vulnerable to path traversal attacks due to inadequate input validation on the file_name parameter. The vulnerability can be demonstrated by inputting this command:
../application.properties

Here is the result:

5.2 Path Manipulation - Fix

The following code adds file_name directly to the resource path without validation, allowing users to navigate up the directory tree and access files outside the designated path:

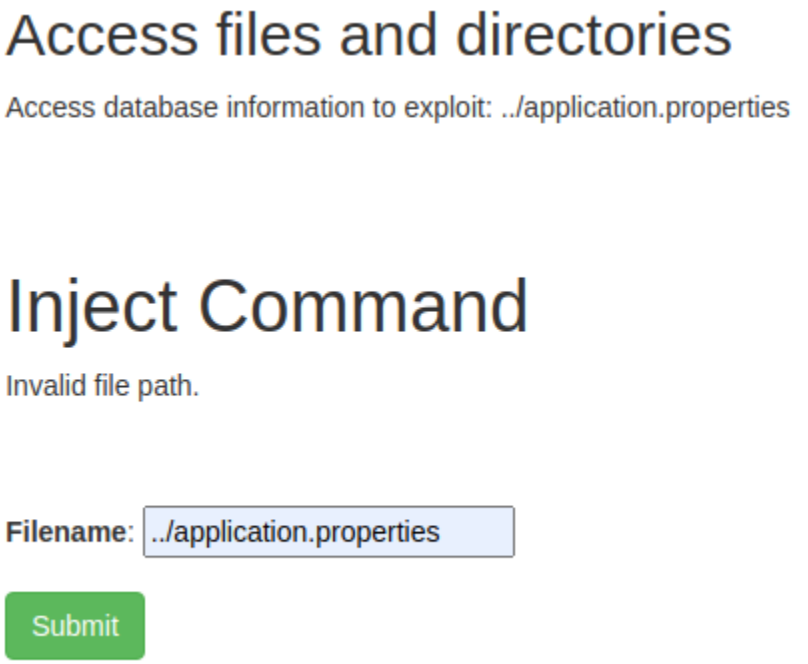
```
35                try {
36                    Resource resource = resourceLoader.getResource("classpath:files/" + file_name);
37                    File file = resource.getFile();
38                    String text = new String(Files.readAllBytes(file.toPath()));
39
```

To fix this, we can sanitize the filename to ensure file_name does not contain special characters such as .. or / that allow traversal outside the designated directory. We will also restrict access to specific directories by using strict path checking to confirm the requested file is within the intended files/directory.

Here is the updated code:

```
    @GetMapping("/viewFile")
    @ResponseBody
    public Map<String, String> view_file(@RequestParam String file_name) throws Exception {
    Map<String, String> response_data = new HashMap<>();

    if (file_name.contains("..") || file_name.contains("/") || file_name.contains("\\")) {
        response_data.put("status", "error");
        response_data.put("msg", "Invalid file path.");
        return response_data;
    }

    try {
        Resource resource = resourceLoader.getResource("classpath:files/" + file_name);
        File file = resource.getFile();

        if (!file.getPath().contains("/files/")) {
            response_data.put("status", "error");
            response_data.put("msg", "Access denied.");
            return response_data;
        }

        String text = new String(Files.readAllBytes(file.toPath()));
        response_data.put("status", "success");
        response_data.put("msg", text);
        return response_data;

    } catch (IOException e) {
        e.printStackTrace();
        response_data.put("status", "error");
        response_data.put("msg", "No output found");
        return response_data;
    }
}
```

Here is the result after inputting the command after the fix is implemented:



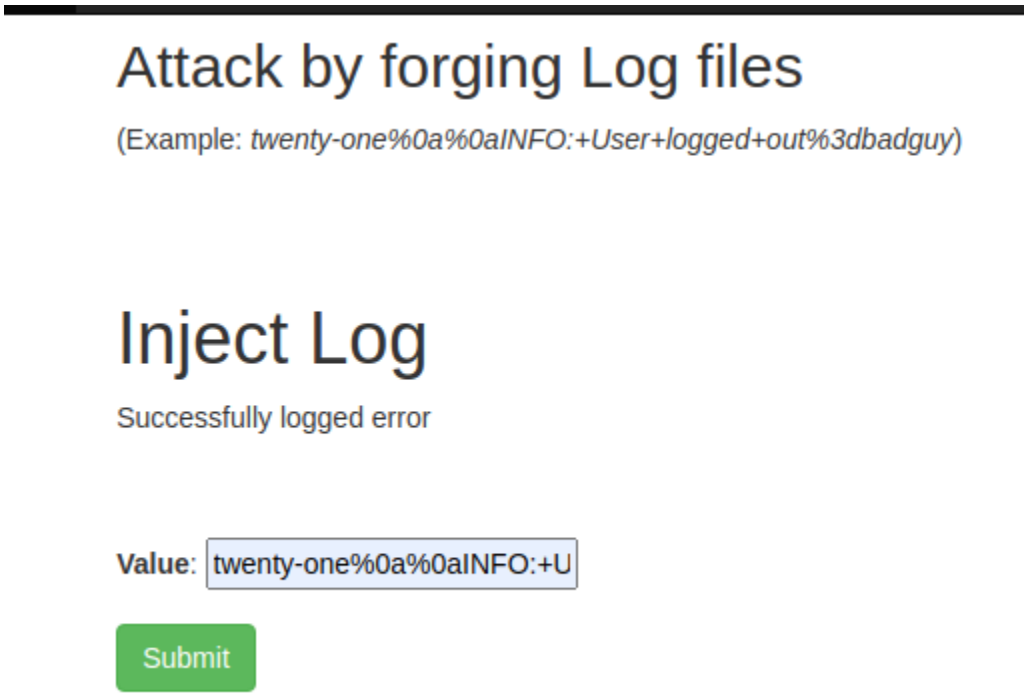
6.0 Log Injection

Log Injection vulnerabilities let an attacker inject arbitrary log messages into log files, potentially overwriting important log information or injecting malicious data that can be interpreted by the system or an administrator. In this case, the application directly writes user-supplied data into log files without proper validation or sanitization, which may allow an attacker to inject special characters or multiple log entries to manipulate the log file.

6.1 Log Injection - Vulnerability

An attacker can provide a specially crafted log value (like the one below) as the input. This input, after being decoded, could inject additional log entries, causing multiple entries or log misinterpretation:
twenty-one%0a%0aINFO:+User+logged+out%3dbadguy

When this is inputted this is the result:



This shows that the log value was successfully injected.

6.2 Log Injection - Fix

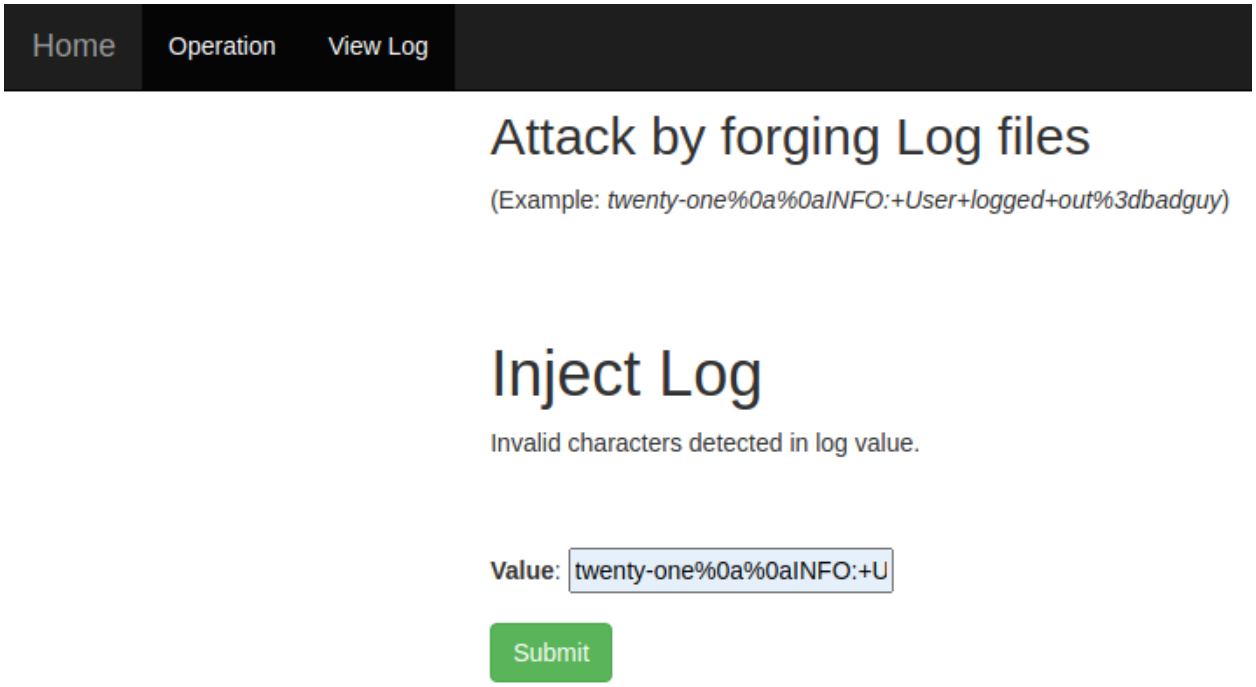
The following line is vulnerable because it allows user input to be directly logged without sanitization or validation, enabling an attacker to inject log entries:

```
43              log_value = java.net.URLDecoder.decode(log_value, StandardCharsets.UTF_8.name());
44              Integer parsed_log_value = Integer.parseInt(log_value);
45              logger.info("Value to log: "                          + parsed_log_value);
```

To fix this, input validation will be utilized to sanitize the log value to ensure it doesn't contain newline or control characters (%0a or %0d) which could be used to forge log entries. The code will also filter special characters before they are written to the log to prevent unwanted formatting or log manipulation.
Here is the fixed code:

```
@PostMapping("/")
@ResponseBody
public Object log_injected(@RequestParam String log_value) {
    Map<String, String> response_data = new HashMap<String, String>();
    Logger logger = LogManager.getLogger(Log_injectionController.class);

    try {
        if (log_value.contains("\n") || log_value.contains("\r") || log_value.contains("%0a") || log_value.contains("%0d")) {
            response_data.put("status", "error");
            response_data.put("msg", "Invalid characters detected in log value.");

            logger.error("Attempted log injection with invalid characters.");
            return response_data;
        }

        SimpleLayout layout = new SimpleLayout();
        FileAppender appender = new FileAppender(layout,"./logs/Custom_log_file.log", true);
        logger.removeAllAppenders();
        logger.addAppender(appender);
        logger.setLevel(Level.DEBUG);
        logger.setAdditivity(true);

        log_value = java.net.URLDecoder.decode(log_value, StandardCharsets.UTF_8.name());
        Integer parsed_log_value = Integer.parseInt(log_value);
        logger.info("Value to log: " + parsed_log_value);

        response_data.put("status", "success");
        response_data.put("msg", "Successfully logged without error");
        return response_data;
    } catch (Exception e) {
        logger.error("Error logging value: " + e.getMessage());
        response_data.put("status", "error");
        response_data.put("msg", "An error occurred while processing the log value.");
        return response_data;
    }
}
```

Here is the result after inputting the log value after the fix is implemented:



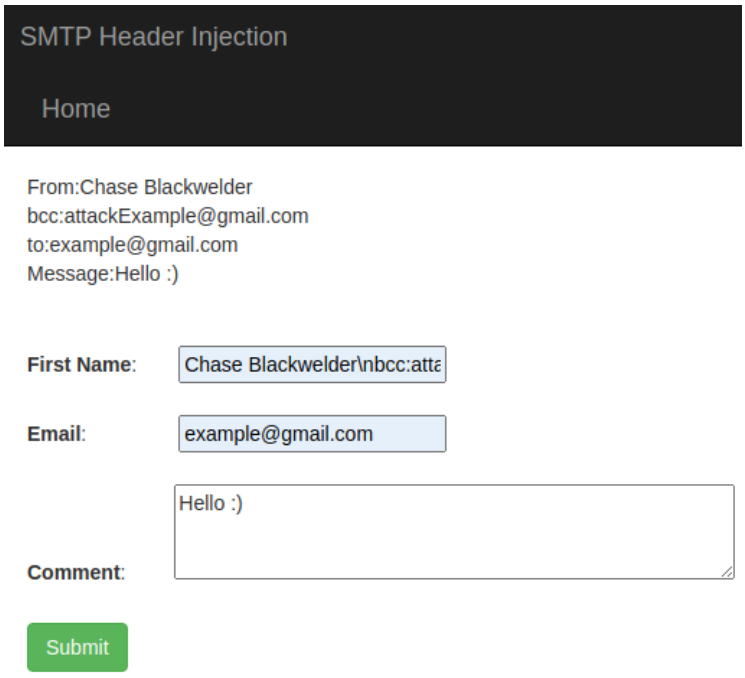
7.0 SMTP

SMTP header injection occurs when an attacker can manipulate email headers by injecting arbitrary values into them. This is typically done by submitting specially crafted input that includes newline characters. This type of vulnerability can lead to unauthorized access to sensitive information, data leakage, or a compromised email system. Proper input validation and sanitization of user inputs are crucial to prevent such attacks.

7.1 SMTP - Vulnerability

In this case, the First Name field contains a newline (\n), allowing the attacker to inject a bcc header. If these inputs are inserted into the fields:
First Name: Chase Blackwelder\nbcc:attackExample@gmail.com
Email: example@gmail.com
Comment: whatever you want, like “Hello :)”

This result is given:



7.2 SMTP - Fix

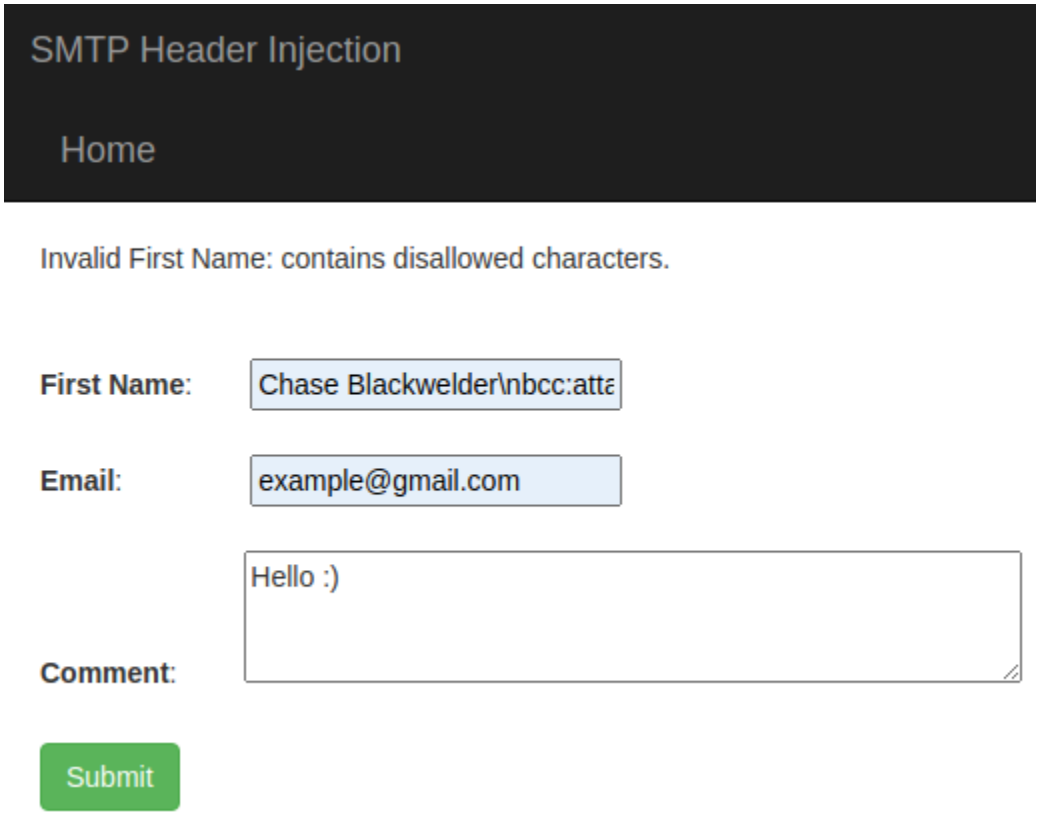
Here, user inputs (customer_firstName, customer_email, and customer_comments) are concatenated into the email headers without validation, allowing attackers to inject new headers, such as bcc, by using newline characters (\n).
Here is the vulnerable code:

```
@GetMapping("/form")
@ResponseBody
public String smtp_header_submit(@RequestParam String customer_firstName,@RequestParam String customer_email,@RequestParam String customer_comments) {

    String name = customer_firstName;
    String email = customer_email;
    String comment = customer_comments;
    String to = "root@localhost";
    String subject = "My Subject";

    String headers = "From:" + name + "\\n" + " to:" + email + "\\n";
    String[] split = headers.split("\\\\n");
    String y="";
    for (int i = 0; i < split.length; i++) {
                    y += split[i];
                    y += "<br>";
            }
    System.out.println(y);
    return y + " Message:" + comment;
}
```

To fix this vulnerability custom validation was added for the inputs using regular expressions. The validation ensures that the First Name and Comment fields contain only safe characters (alphanumeric, spaces, commas, and hyphens), while the Email field is validated for proper format. Invalid inputs are rejected early, preventing any potential injection into the email headers. Here is the fixed code:

```
@GetMapping("/form")
@ResponseBody
public String smtp_header_submit(@RequestParam String customer_firstName, @RequestParam String customer_email, @RequestParam String customer_comments) {

    String name = "";
    String email = "";
    String comment = "";

    if (!isValidSafeString(customer_firstName)) {
        return "Invalid First Name: contains disallowed characters.";
    }
    if (!isValidEmail(customer_email)) {
        return "Invalid Email: format is incorrect.";
    }
    if (!isValidSafeString(customer_comments)) {
        return "Invalid Comment: contains disallowed characters.";
    }

    name = customer_firstName;
    email = customer_email;
    comment = customer_comments;

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

    String headers = "From:" + name + "\n" + "To:" + email + "\n";
    String[] split = headers.split("\n");
    String headerOutput = "";
    for (int i = 0; i < split.length; i++) {
        headerOutput += split[i];
        headerOutput += "<br>";
    }

    return headerOutput + "Message: " + comment;
}

private boolean isValidSafeString(String input) {
    String regex = "^[\\p{Alnum}\\p{Space},-]+$";
    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(input);
    return matcher.matches();
}

private boolean isValidEmail(String email) {
    String regex = "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$";
    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(email);
    return matcher.matches();
}
```

Now when the same input from before is used, this is the result:



8.0 Handling Exceptions

Exceptions will be handled by the SecurityEnhancedAPI class, which validates filenames, safe strings, and email addresses. The goal is to handle these exceptions by logging error messages, encoding the faulty strings, and rejecting invalid inputs. The following steps were done to address the vulnerabilities:

The Path_manipulationController handles requests related to file path manipulation, while the SmtpController handles requests related to email validation, and string validation. The following changes were made to handle exceptions effectively:

Logging and Error Handling: When an exception is thrown by the SecurityEnhancedAPI, it is caught in the controller, and an error message is returned to the user. The error message includes the exception details, ensuring that users are informed of what went wrong.

Encoding Problematic Inputs: The input that caused the exception is encoded to prevent further issues like code injection or malicious input exploitation. This ensures that the system safely handles problematic input without compromising security.

Rejecting Invalid Inputs: If an exception is thrown, the controller rejects the input and returns a response indicating that the validation failed. The invalid input is not processed further, preventing any potential security risks.

8.1 Handling Exceptions - Fix

Here is security/SecurityEnhancedAPI.java:

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

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.io.Resource;
import org.springframework.core.io.ResourceLoader;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.HashMap;
import java.util.Map;

@Controller
@RequestMapping("/path_manipulation")
public class Path_manipulationController {

        @Autowired
        ResourceLoader resourceLoader;

    @GetMapping("/")
    public String path_manipulation_index() {
        return "path_manipulation/index";
    }

    @GetMapping("/viewFile")
    @ResponseBody
    public Map<String, String> view_file(@RequestParam String file_name) throws Exception {
    Map<String, String> response_data = new HashMap<>();

    if (file_name.contains("..") || file_name.contains("/") || file_name.contains("\\")) {
        response_data.put("status", "error");
        response_data.put("msg", "Invalid file path.");
        return response_data;
    }

    try {
        Resource resource = resourceLoader.getResource("classpath:files/" + file_name);
        File file = resource.getFile();

        if (!file.getPath().contains("/files/")) {
            response_data.put("status", "error");
            response_data.put("msg", "Access denied.");
            return response_data;
        }

        String text = new String(Files.readAllBytes(file.toPath()));
        response_data.put("status", "success");
        response_data.put("msg", text);
        return response_data;

    } catch (IOException e) {
        e.printStackTrace();
        response_data.put("status", "error");
        response_data.put("msg", "No output found");
        return response_data;
```

security/exceptions/WrongEmail.java:

```java
package net.uncc.app.security.exceptions;

public class WrongEmail extends Exception {
    public WrongEmail(String message) {
        super(message);
    }
}

```

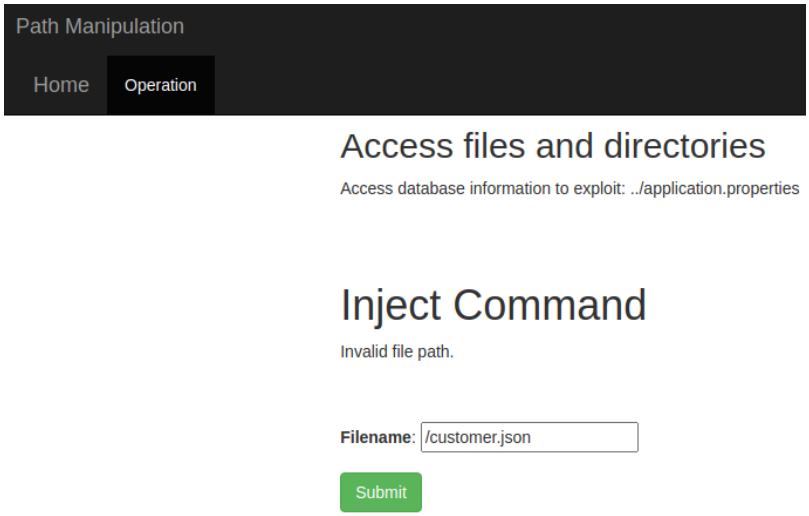
security/exceptions/WrongFileName.java:

```java
package net.uncc.app.security.exceptions;

public class WrongFileName extends Exception {
    public WrongFileName(String message) {
        super(message);
    }
}

```

security/exceptions/WrongSafeString.java:

```java
package net.uncc.app.security.exceptions;

public class WrongSafeString extends Exception {
    public WrongSafeString(String message) {
        super(message);
    }
}

```

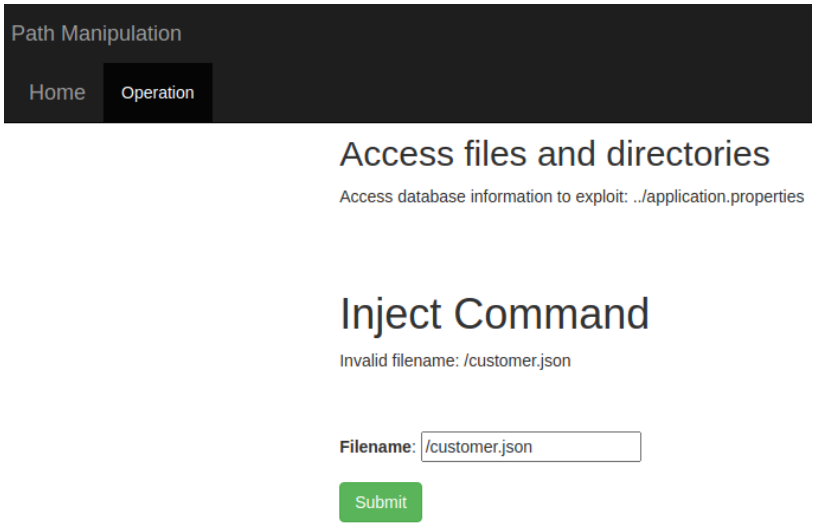
Here is the output given by the old code in the Path_manipulationController.java file:



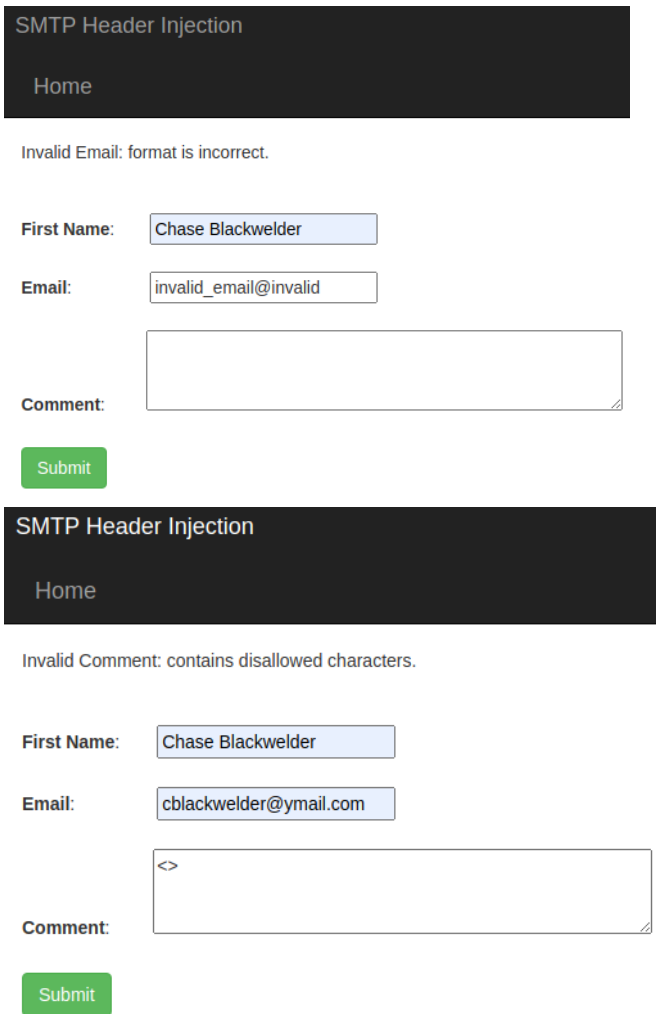
Here is the updated Path_manipulationController.java file which implements the SecurityEnhancedAPI file:

```java
@Controller
@RequestMapping("/path_manipulation")
public class Path_manipulationController {

    @Autowired
    ResourceLoader resourceLoader;

    @Autowired
    SecurityEnhancedAPI securityEnhancedAPI;

    @GetMapping("/")
    public String path_manipulation_index() {
        return "path_manipulation/index";
    }

    @GetMapping("/viewFile")
    @ResponseBody
    public Map<String, String> view_file(@RequestParam String file_name) throws Exception {
        Map<String, String> response_data = new HashMap<>();

        try {
            String validFileName = securityEnhancedAPI.GetFilename(file_name);

            if (file_name.contains("..") || file_name.contains("/") || file_name.contains("\\")) {
                response_data.put("status", "error");
                response_data.put("msg", "Invalid file path.");
                return response_data;
            }

            Resource resource = resourceLoader.getResource("classpath:files/" + validFileName);
            File file = resource.getFile();

            if (!file.getPath().contains("/files/")) {
                response_data.put("status", "error");
                response_data.put("msg", "Access denied.");
                return response_data;
            }

            String text = new String(Files.readAllBytes(file.toPath()));
            response_data.put("status", "success");
            response_data.put("msg", text);
            return response_data;

        } catch (WrongFileName e) {
            response_data.put("status", "error");
            response_data.put("msg", e.getMessage());
            return response_data;
        } catch (IOException e) {
            response_data.put("status", "error");
            response_data.put("msg", "No output found");
            return response_data;
        }
    }
}
```

And here is the output after said implementation:



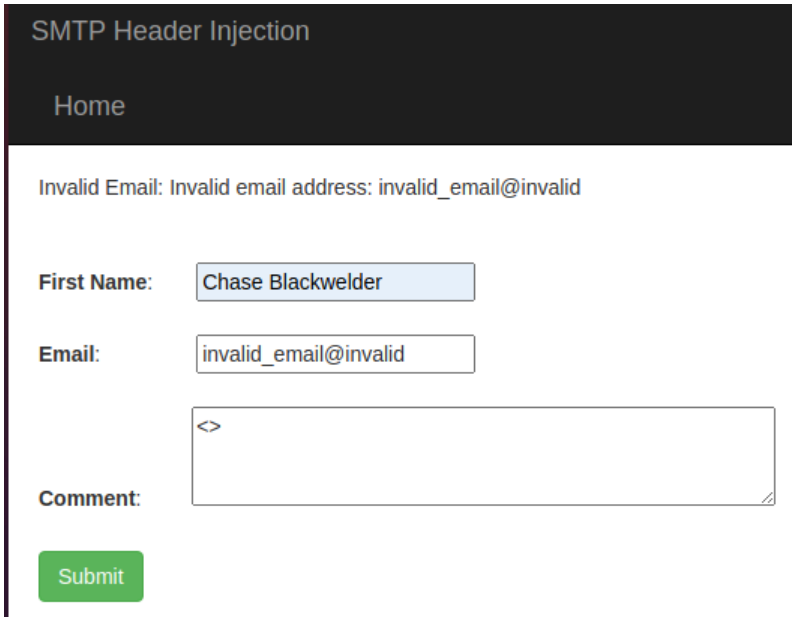
SecurityEnhancedAPI can also be implemented in the SmtpController. Here is the output for both an invalid email and disallowed characters being used as input:



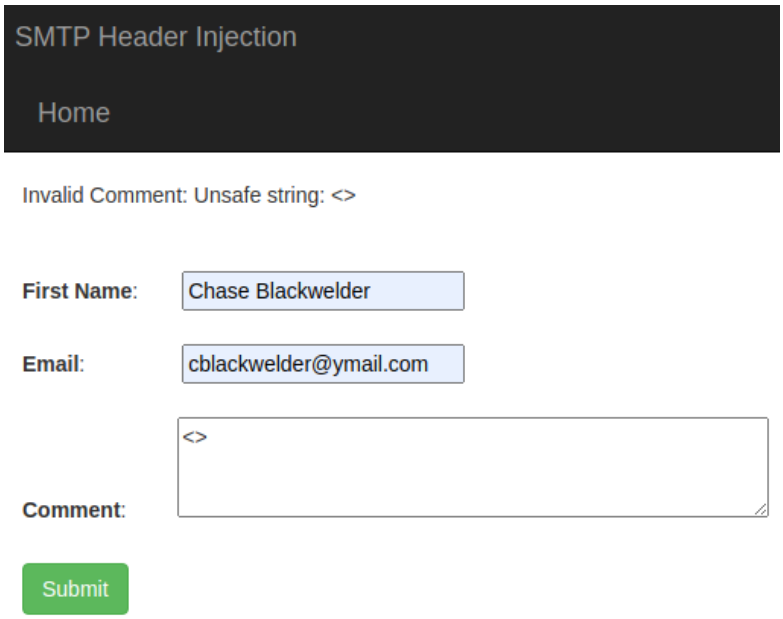
When SecurityEnhancedAPI is implemented, this is what the code will look like:

```
15 @Controller
16 @RequestMapping("/smtp_injection")
17 public class SmtpController {
18
19     @Autowired
20     ResourceLoader resourceLoader;
21
22     @Autowired
23     SecurityEnhancedAPI securityEnhancedAPI;
24
25     @GetMapping("/")
26     public String smtp_header_index() {
27         return "smtp_injection/index";
28     }
29
30     @GetMapping("/form")
31     @ResponseBody
32     public String smtp_header_submit(@RequestParam String customer_firstName, @RequestParam String customer_email, @RequestParam String customer_comments) {
33
34         String name = "";
35         String email = "";
36         String comment = "";
37
38         try {
39             email = securityEnhancedAPI.GetEmail(customer_email);
40         } catch (WrongEmail e) {
41             return "Invalid Email: " + e.getMessage();
42         }
43
44         try {
45             name = securityEnhancedAPI.GetSafeString(customer_firstName);
46         } catch (WrongSafeString e) {
47             return "Invalid First Name: " + e.getMessage();
48         }
49
50         try {
51             comment = securityEnhancedAPI.GetSafeString(customer_comments);
52         } catch (WrongSafeString e) {
53             return "Invalid Comment: " + e.getMessage();
54         }
55
56         String to = "root@localhost";
57         String subject = "My Subject";
58
59         String headers = "From:" + name + "\n" + "To:" + email + "\n";
60         String[] split = headers.split("\n");
61         String headerOutput = "";
62         for (String header : split) {
63             headerOutput += header;
64             headerOutput += "<br>";
65         }
66
67         return headerOutput + "Message: " + comment;
68     }
69
70 }
```

This is the output for when an invalid email is inputted:



And this is the output for when an invalid comment is inputted:



9.0 Xpath Query

XPath Injection is a security vulnerability that allows an attacker to manipulate an XPath query to access or modify data in an XML document in unauthorized ways. This vulnerability occurs when user input is directly embedded in an XPath query without proper sanitization, enabling the attacker to craft malicious input that can alter the query's logic.

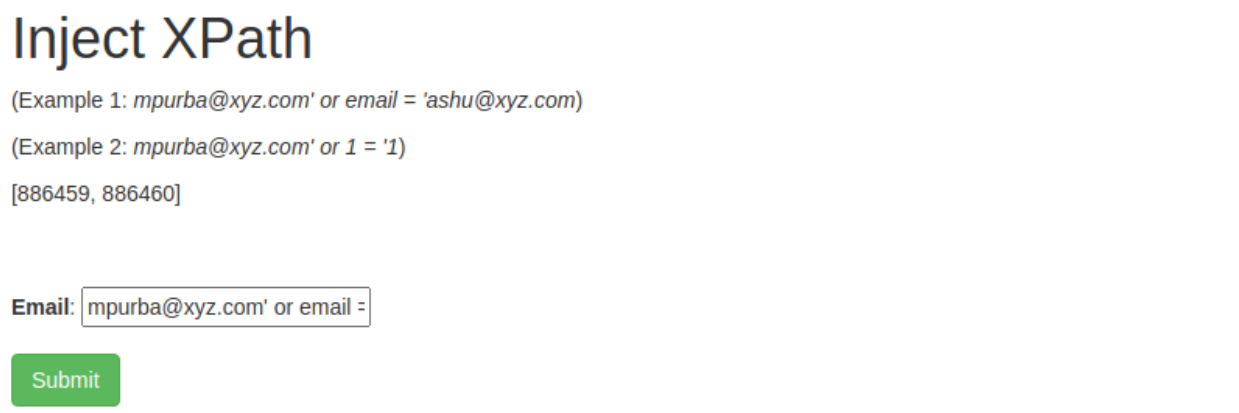
9.1 Xpath Query - Vulnerability

There is an XPath Injection vulnerability in the XPath_injectionController class of the application, which is responsible for processing user inputs related to customer data. Specifically, the email address input was used in an XPath query to fetch customer IDs, but the query construction was vulnerable to manipulation.

User input (the email address) is directly inserted into the XPath query without any sanitization or validation, allowing attackers to manipulate the structure of the query and execute arbitrary XPath expressions Here is the part of the code which contains this vulnerability:

```
List<String> id_list = new ArrayList<>();
XPathExpression expression = xpath.compile("/customers/customer[email = '" + email_address + "']/id/text()");
NodeList nodes = (NodeList) expression.evaluate(doc, XPathConstants.NODESET);
for (int i = 0; i < nodes.getLength(); i++)
```

This is the current output for malicious code input with this code:



9.2 Xpath Query - Fix

This can be fixed through the use of a parameterized XPath expression, where the user input is treated as a variable instead of being directly concatenated into the query string. The SimpleVariableResolver class resolves the user input in a safe way, making it impossible for the attacker to alter the query structure. Here is the SimpleVariableResolver class:

```
1 package net.uncc.app.xpath_injection;
2
3 import javax.xml.namespace.QName;
4 import javax.xml.xpath.XPathVariableResolver;
5 import java.util.HashMap;
6 import java.util.Map;
7
8 public class SimpleVariableResolver implements XPathVariableResolver {
9
10     private final Map<QName, Object> vars = new HashMap<QName, Object>();
11     public void addVariable(QName name, Object value) {
12         vars.put(name, value);
13     }
14
15     public Object resolveVariable(QName variableName) {
16         return vars.get(variableName);
17     }
18
19 }
20
```

And here is how this new class was implemented in the XPath_injectionController.java file:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse("src/main/resources/files/customer.xml");

XPathFactory xpathFactory = XPathFactory.newInstance();
XPath xpath = xpathFactory.newXPath();

SimpleVariableResolver resolver = new SimpleVariableResolver();
resolver.addVariable(new QName(null, "email_val"), email_address);
xpath.setXPathVariableResolver(resolver);

XPathExpression expression = xpath.compile("/customers/customer[email = $email_val]/id/text()");
NodeList nodes = (NodeList) expression.evaluate(doc, XPathConstants.NODESET);
```

This is the output after the new code is implemented and the same input form before is used: