Preventing Injection Attacks Report - Tunestore

Preventing Injection Attacks Report - Tunestore Report

Tristan Allison

ITIS 4221/5221

March 26th, 2022

PREVENTING INJECTION ATTACKS REPORT - TUNESTORE

TABLE OF CONTENTS

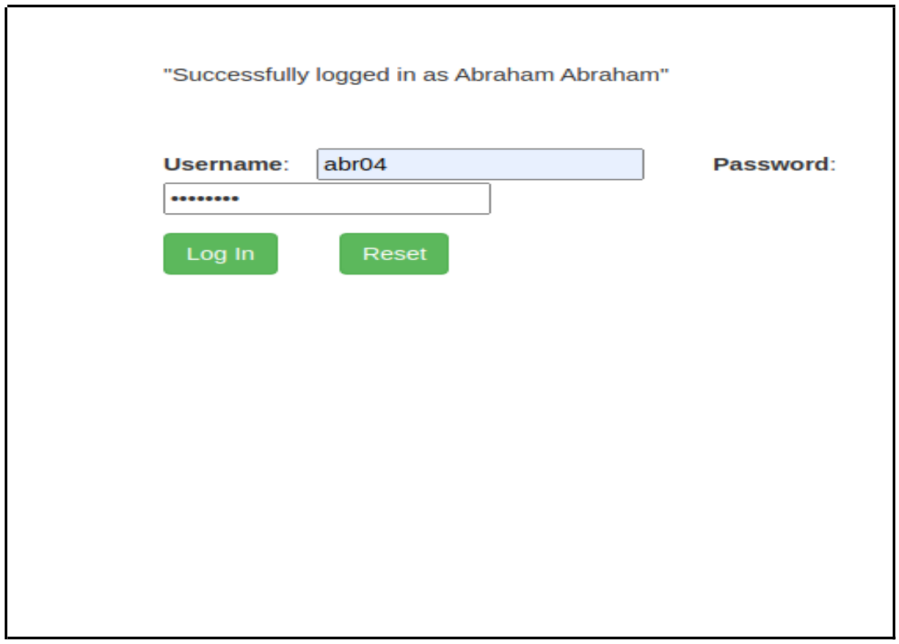
1.0 SQL Injection Mitigations

In the current version of the penetration test, there is an SQL injection vulnerability present in the login functionality of the application. This vulnerability allows an attacker to login as a random user by entering the following in the password field:

‘ OR ‘1’=’1

The screenshots below show this SQL injection vulnerability being maliciously exploited.

‘ OR ‘1’=’1 is entered into the password field:



The following screenshot contains the original code in LoginAction.java that runs when a user attempts to login to Tunestore (vulnerable lines of code highlighted).

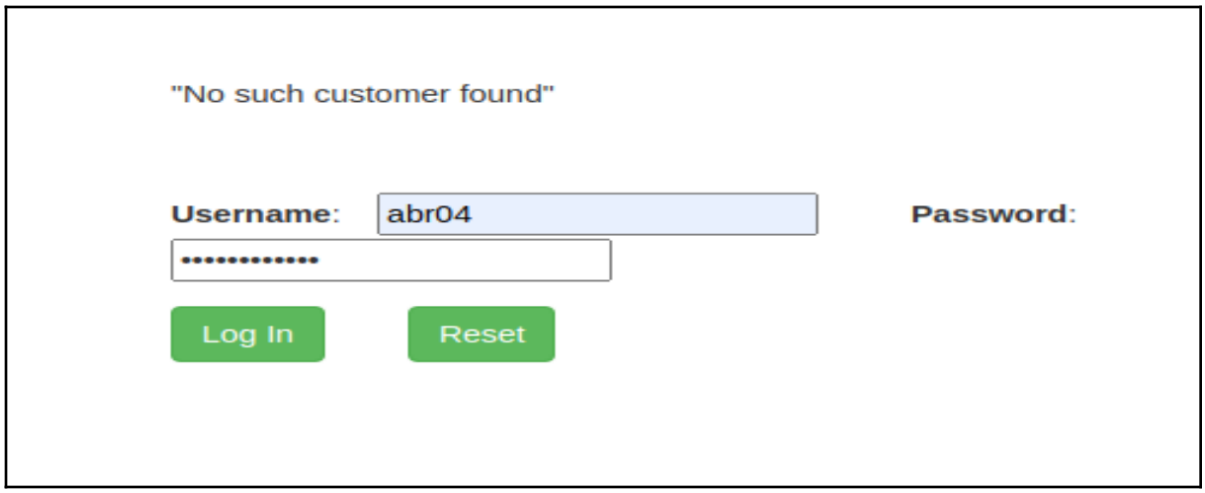
```
@PostMapping("/")
@ResponseBody
public Map<String, String> sql_logged_in(@RequestParam String employee_username, @RequestParam St
    String queryString = "SELECT * From Employees where username = '" + employee_username + "' ar
    Object[] parameters = {employee_username, employee_password};
    List<Map> listOfemployee = (List<Map>) findDataFromDatabase(queryString, parameters);

    Map<String, String> response_data = new HashMap<String, String>();
```

The first step in fixing this SQL injection vulnerability was to add two more lines of code that causes the user's login and password input to move to the position of the *?* placeholders. Now that the query's structure has been fixed, the user can enter any type of input without affecting the structure. The query can now be executed safely.

Below is a screenshot that displays the code after the changes described above were made (updated lines of code highlighted).

```
public Map<String, String> sql_logged_in(@RequestParam String employee_username, @RequestPa
    Connection conn = jdbcTemplate.getDataSource().getConnection();
    String queryString = "SELECT * From Employees where username = ? and password = ?'";
    PreparedStatement stmt = conn.prepareStatement(queryString);
    stmt.setString( parameterIndex: 1,employee_username);
    stmt.setString( parameterIndex: 2,employee_password);
    Object[] parameters = {employee_username, employee_password};
    List<Map> listOfemployee = (List<Map>) stmt.executeQuery();

    Map<String, String> response_data = new HashMap<String, String>();
```

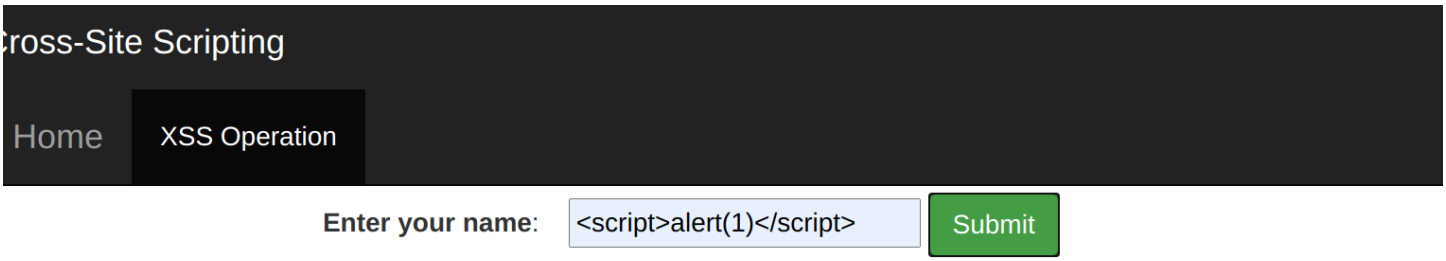
Now, when an attacker attempts to use SQL injection to login as a random user, the following will occur:

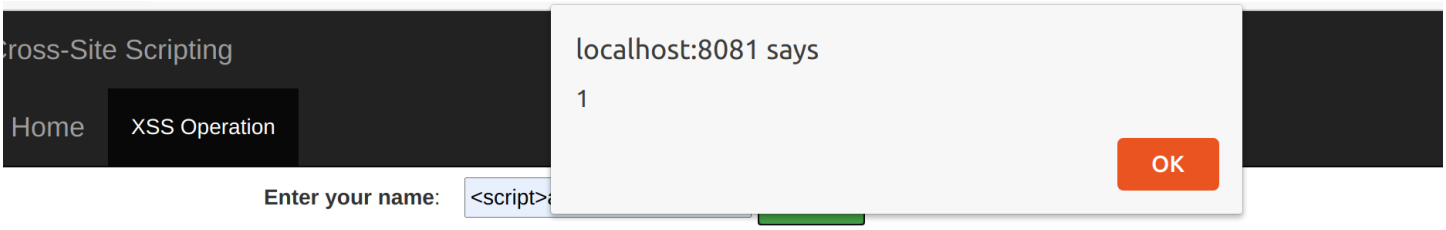


Instead of the attacker being logged in as a random user, the input is now handled correctly.

2.0 XSS Mitigations

- An XSS vulnerability is where javascript is inserted into a webpage to make it do something when the user loads the page. Two different types of XSS vulnerabilities are stored and reflected. The difference between them is that stored is stored on the website itself so whenever anyone goes to the website it loads, but Reflected is where the link that the user uses has been modified.





- *provide a screenshot/excerpt of the source code that contains the vulnerable code*

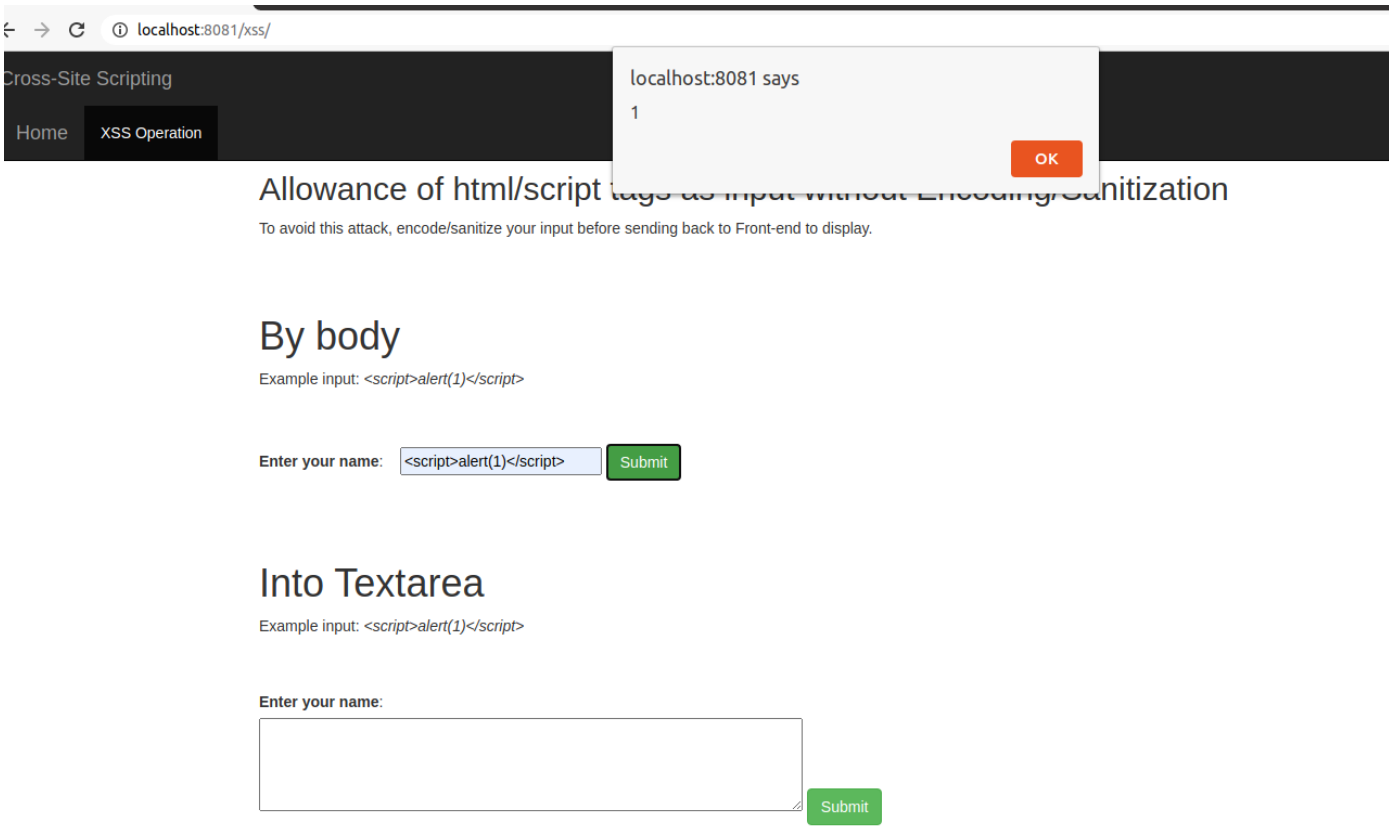
```
    @ResponseBody
    public String body_xss(@RequestParam String body_tagVal) throws Exception {
        return body_tagVal;
    }

    @GetMapping("/textarea_xss")
    @ResponseBody
    public Object textarea_xss(@RequestParam String textarea_tagVal) throws Exception {
        return textarea_tagVal;
    }

    @GetMapping("/js_xss")
    @ResponseBody
    public Object js_xss(@RequestParam String js_tagVal) throws Exception {
        return js_tagVal;
    }

}
```

● *Briefly explain how the original code needs to be updated and provide the new fixed code*
The code needs sanitize so that the input is not just passing what is entered into the html. This can be done by using an escapehtml(userInput). This will keep the code from running if there is an HTML tag in the text area such as the javascript tag to run javascript code.

```
    @GetMapping("/")
    public String xss_index() { return "xss/index"; }

    @GetMapping("/body_xss")
    @ResponseBody
    public String body_xss(@RequestParam String body_tagVal) throws Exception {
        return escapeHtml(body_tagVal);
    }

    @GetMapping("/textarea_xss")
    @ResponseBody
    public Object textarea_xss(@RequestParam String textarea_tagVal) throws Exception {
        return escapeHtml(textarea_tagVal);
    }

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

- *If I had instead used escapeCSV it would not have fixed it as shown below:*

```
    @GetMapping("/")
    public String xss_index() { return "xss/index"; }

    @GetMapping("/body_xss")
    @ResponseBody
    public String body_xss(@RequestParam String body_tagVal) throws Exception {
        return escapeCsv(body_tagVal);
    }

    @GetMapping("/textarea_xss")
    @ResponseBody
    public Object textarea_xss(@RequestParam String textarea_tagVal) throws Exception {
        return escapeHtml(textarea_tagVal);
    }

    @GetMapping("/js_xss")
    @ResponseBody
    public Object js_xss(@RequestParam String js_tagVal) throws Exception {
```



3.0 Path Manipulation Mitigations

● *Path manipulation is where the url can be changed to get into things that you do not have permission to get into such as not being an admin and being able to go into admin only files.*

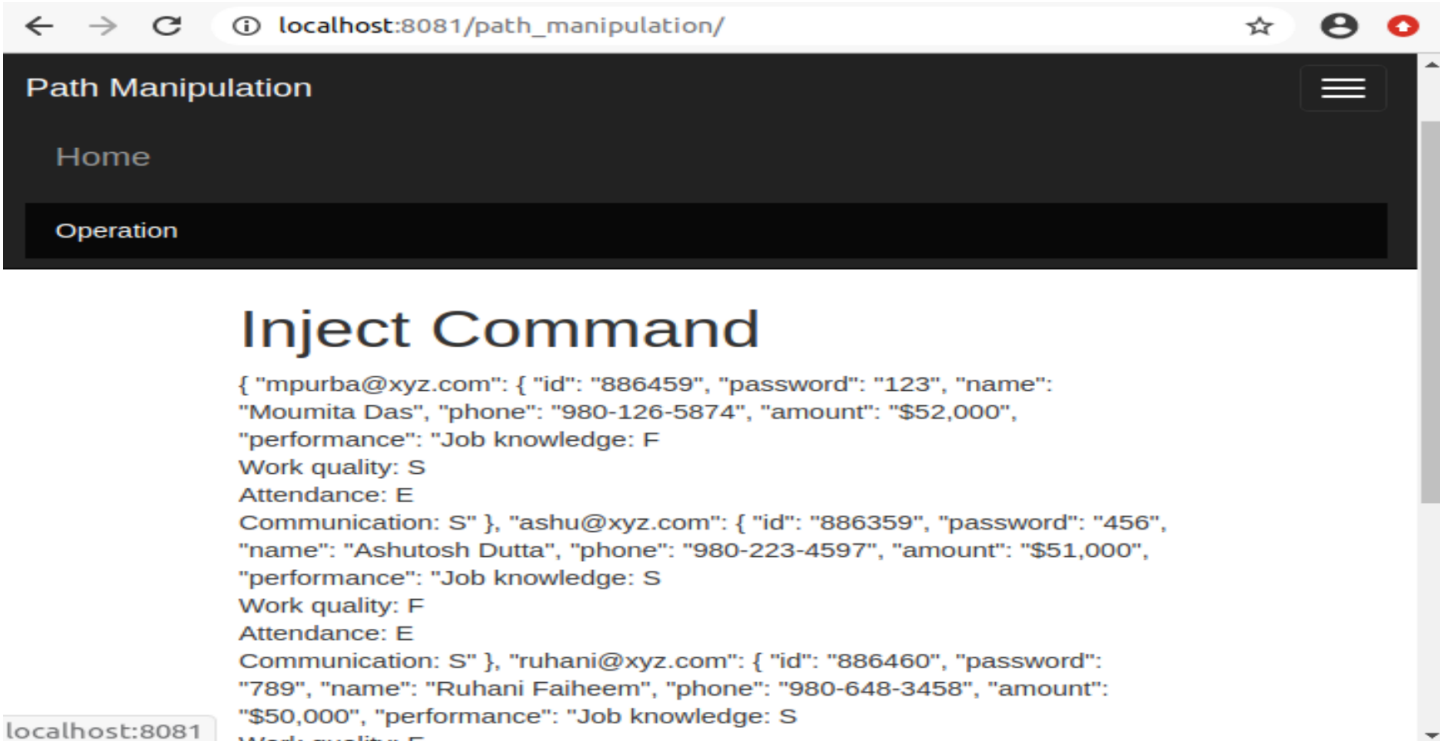
Access database information to exploit: ../application.properties

Inject Command

Filename: customer.json

Submit

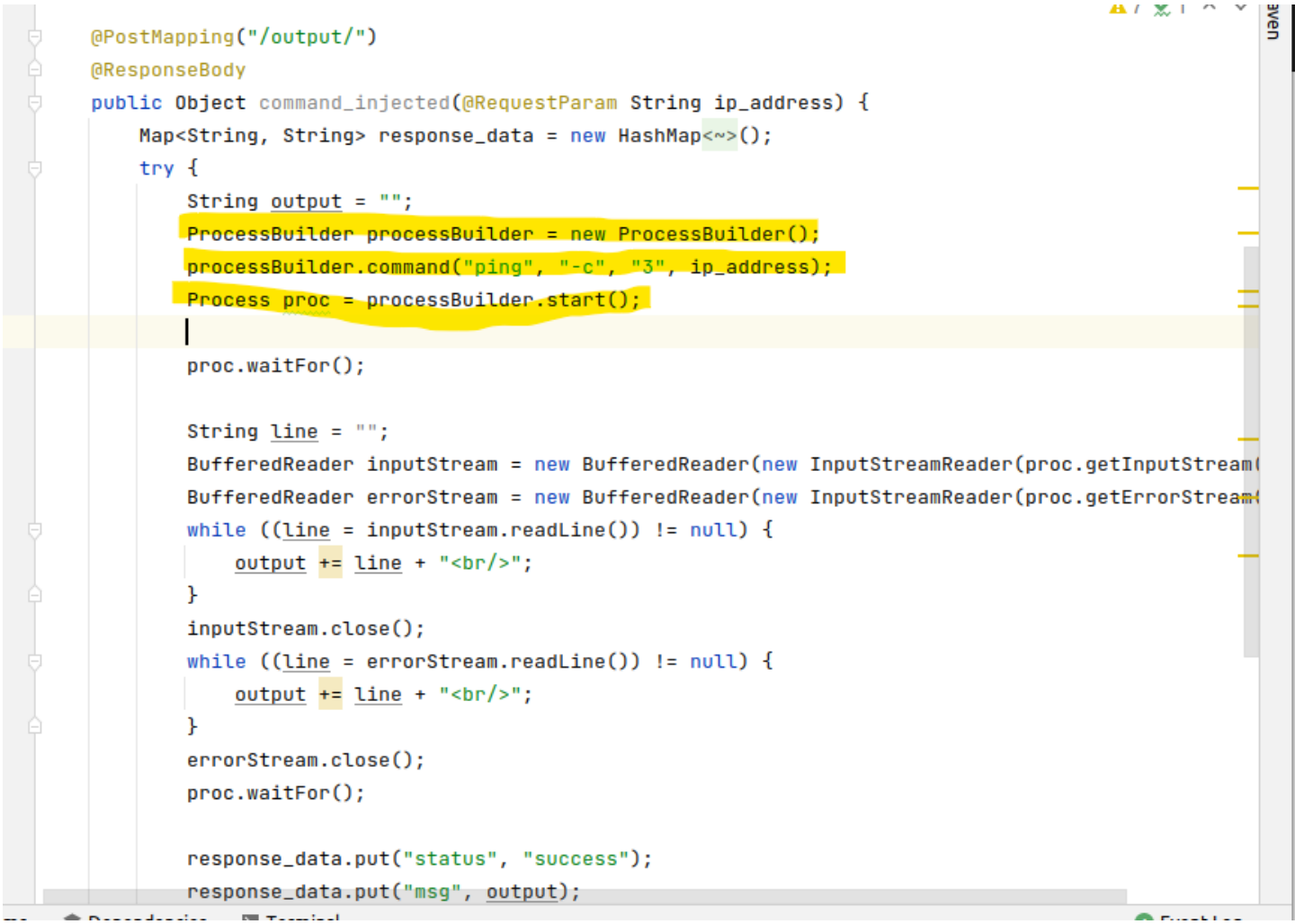
localhost:8081



4.0 Command Injection Mitigations

● *A command Injection Vulnerability is where the user can pass data to the host from the application and the server runs it in a system shell.*

● *Briefly explain how the original code needs to be updated and provide the new fixed code*



```
@PostMapping("/output/")
@ResponseBody
public Object command_injected(@RequestParam String ip_address) {
    Map<String, String> response_data = new HashMap<~>();
    try {
        String output = "";
        ProcessBuilder processBuilder = new ProcessBuilder();
        processBuilder.command("ping", "-c", "3", ip_address);
        Process proc = processBuilder.start();

        proc.waitFor();

        String line = "";
        BufferedReader inputStream = new BufferedReader(new InputStreamReader(proc.getInputStream(
        BufferedReader errorStream = new BufferedReader(new InputStreamReader(proc.getErrorStream(
        while ((line = inputStream.readLine()) != null) {
            output += line + "<br/>";
        }
        inputStream.close();
        while ((line = errorStream.readLine()) != null) {
            output += line + "<br/>";
        }
        errorStream.close();
        proc.waitFor();

        response_data.put("status", "success");
        response_data.put("msg", output);
```

These lines of code need to be changed so that it is not just using the user's input and is instead validating that the user is entering the correct input. This can be done by switching this line of code with: *ProcessBuilder processBuilder = new ProcessBuilder();*

processBuilder.command("ping", "-c", "3", ip_address);

Process proc = processBuilder.start();

- *Rerun the exploitation and show that it is now fixed*

5.0 Log Forging Mitigations

Log Forging section here.

Provide a quick description of the vulnerability along with a screenshot that shows the

vulnerability being maliciously exploited (similar to Tunestore I and II)

● *A log Forging vulnerability is where a user is able to have their unvalidated input written to the log files allowing them to fake log entries or inject malicious code into the logs.*

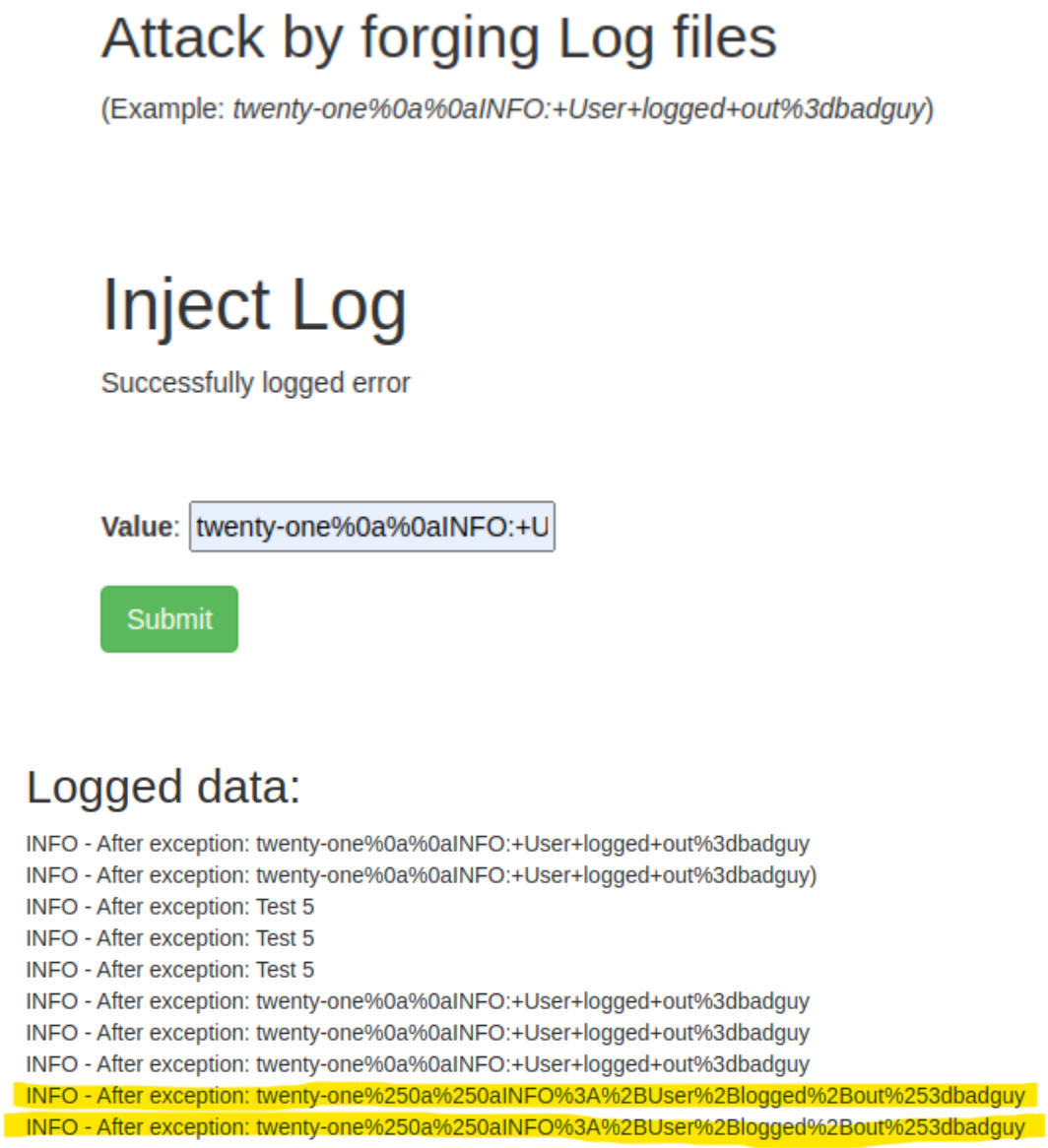
Logged data:

INFO - After exception: twenty-one%0a%0aINFO:+User+logged+out%3dbadguy
INFO - After exception: twenty-one%0a%0aINFO:+User+logged+out%3dbadguy)
INFO - After exception: Test 5
INFO - After exception: Test 5
INFO - After exception: Test 5
INFO - After exception: twenty-one%0a%0aINFO:+User+logged+out%3dbadguy
INFO - After exception: twenty-one%0a%0aINFO:+User+logged+out%3dbadguy
INFO - After exception: twenty-one%0a%0aINFO:+User+logged+out%3dbadguy

● *The code needs to encode the entry to keep the user from entering something and it being logged so when it is encoded it changes what the user entered so that the user's entry does not get logged in the same form it is entered in to prevent a log forgery.*

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

        log_value = java.net.URLEncoder.encode(log_value,StandardCharsets.UTF_8.toString());
        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.info("After exception: " + log_value);
        response_data.put("status", "error");
        response_data.put("msg", "Successfully logged error");
        return response_data;
```

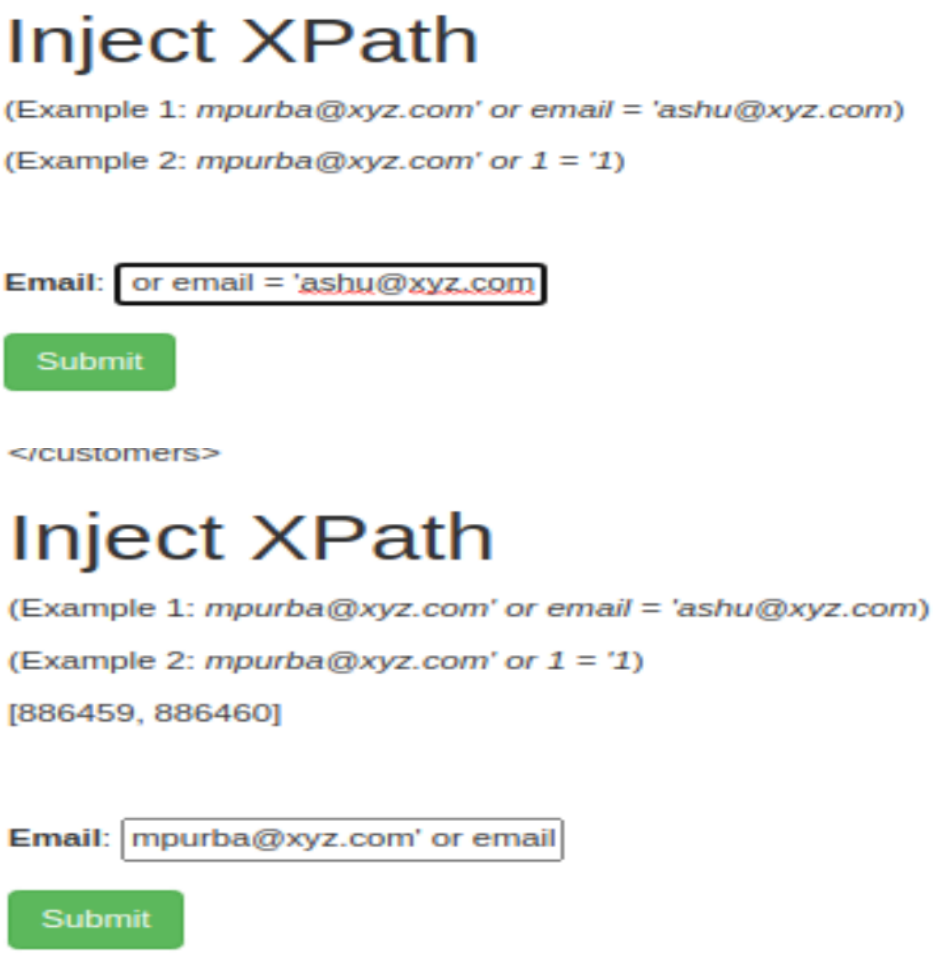
● *Rerun the exploitation and show that it is now fixed*



The highlighted area above shows that the log no longer has the data manipulated despite the input being the same.

6.0 XPath Injection Mitigations

● *XPath injection manipulation is where the user enters information and that information is used to create a path to access things and the user uses it to access sensitive information that they should not have access to.*



```java
@PostMapping("/")
@ResponseBody
public Object xpath_injected(@RequestParam String email_address) {
    Map<String, String> response_data = new HashMap<~>();
    try {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        factory.setNamespaceAware(true);
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document doc = builder.parse( uri: "src/main/resources/files/customer.xml");

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

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

        response_data.put("status", "success");
```

● *Changing the code to : SimpleVariableResolver resolver = new SimpleVariableResolver();*

resolver.addVariable(new QName(null, "email_val"), email_address);

xpath.setXPathVariableResolver(resolver);

XPathExpression expre = xpath.compile("/customers/customer[email = $email_val]/id/text()");

Fixes the errors in the original code.

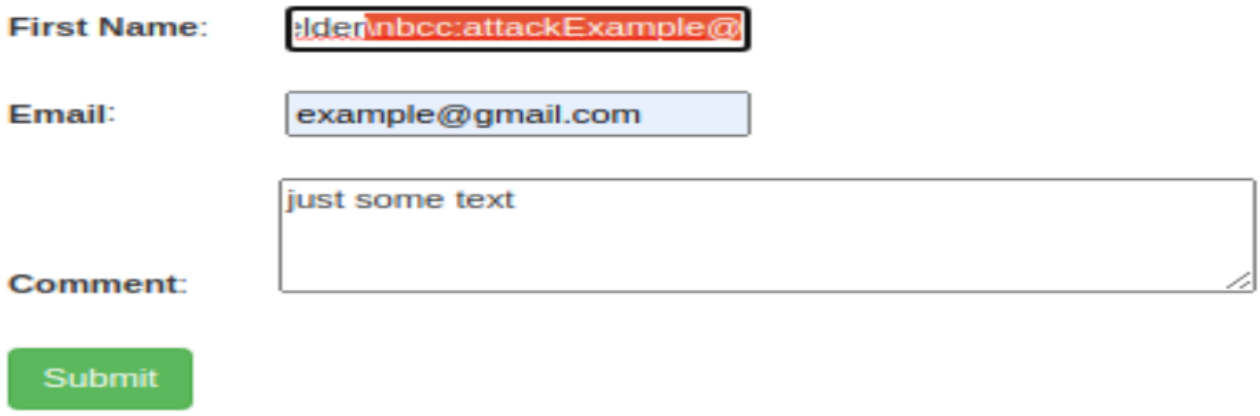
```java
        Document doc = builder.parse( uri: "src/main/resources/files/customer.xml");

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

        List<String> id_list = new ArrayList<>();
        //

        SimpleVariableResolver resolver = new SimpleVariableResolver();
        resolver.addVariable(new QName( namespaceURI: null, localPart: "email_val"), email_address);
        xpath.setXPathVariableResolver(resolver);
        XPathExpression expre = xpath.compile( expression: "/customers/customer[email = $email_val]/id/text()");

        //
        NodeList nodes = (NodeList) expre.evaluate(doc, XPathConstants.NODESET);
        for (int i = 0; i < nodes.getLength(); i++)
            id_list.add(nodes.item(i).getNodeValue());

        response_data.put("status", "success");
        response_data.put("msg", Arrays.toString(id_list.toArray()));
```

● *This code fixes the vulnerability:*

Now when XPath manipulation is tried all the person gets is [] so they do not get any information.

7.0 SMTP Header Injection Mitigations

● *SMTP Header Injection is where malicious code is put into the header of an email, because the data is not sanitized properly. This allows for the contents of the email to be altered as well as attach viruses or send the email to a third party.*



From:Chase Blackwelder
bcc:attackExample@gmail.com
to:example@gmail.com
Message:just some text

First Name: Chase Blackwelder\nbcc:at

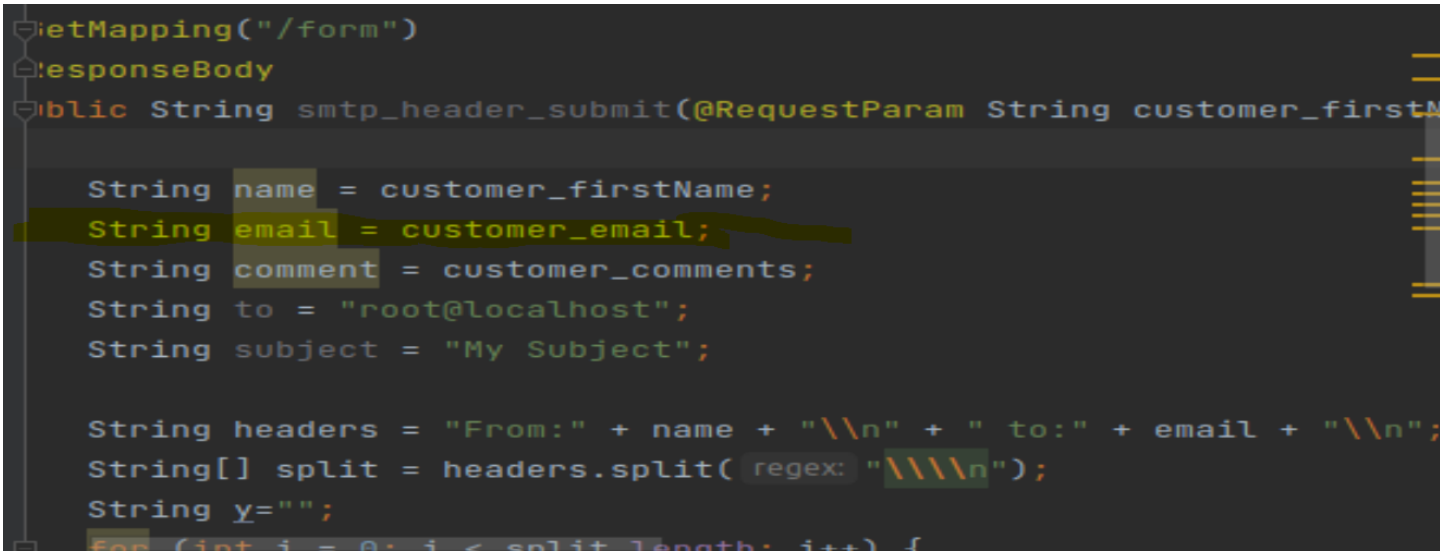
Email: example@gmail.com

Comment: just some text

Submit

● *Briefly explain how the original code needs to be updated and provide the new fixed code*

This can be fixed by not allowing the /n character to be entered into the header of the email.



The error is that it just takes the customer email without doing anything so the user can enter anything as the email. It can be fixed using replace to get rid of any unwanted characters that the person sending

the email could use to make the email malicious. This can be done with noralizeSpace() or a replace, but normalizeSpace has it built in.

```
Path_manipulationController.java | SmtpController.java | Log_injectionController.java
19      @GetMapping("/")
20      public String smtp_header_index() { return "smtp_injection/index"; }
23
24      @GetMapping("/form")
25      @ResponseBody
26      public String smtp_header_submit(@RequestParam String customer_firstName,@RequestParam String customer_email,@RequestParam Stri
27          customer_email=customer_email.replaceAll( regex: "\n", replacement: "");
28          customer_firstName = customer_firstName.replaceAll( regex: "\n", replacement: "");
29          String name = customer_firstName;
30          String email = customer_email;
31
32          String comment = customer_comments;
33          String to = "root@localhost";
34          String subject = "My Subject";
35
36          String headers = "From:" + name + "\\n" + " to:" + email + "\\n";
37          String[] split = headers.split( regex: "\\\\n");
38          String y="";
39          for (int i = 0; i < split.length; i++) {
```

- *Rerun the exploitation and show that it is now fixed*