

# **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

	<u>Section</u>	<u>Page #</u>
1.0 SQL Injection Mitigations		3
2.0 XSS Mitigations		5
3.0 Path Manipulation Mitigations		8
4.0 Command Injection Mitigations	6	
5.0 Log Forging Mitigations	7	
6.0 XPath Injection Mitigations	7	
7.0 SMTP Header Injection Mitigations	7	

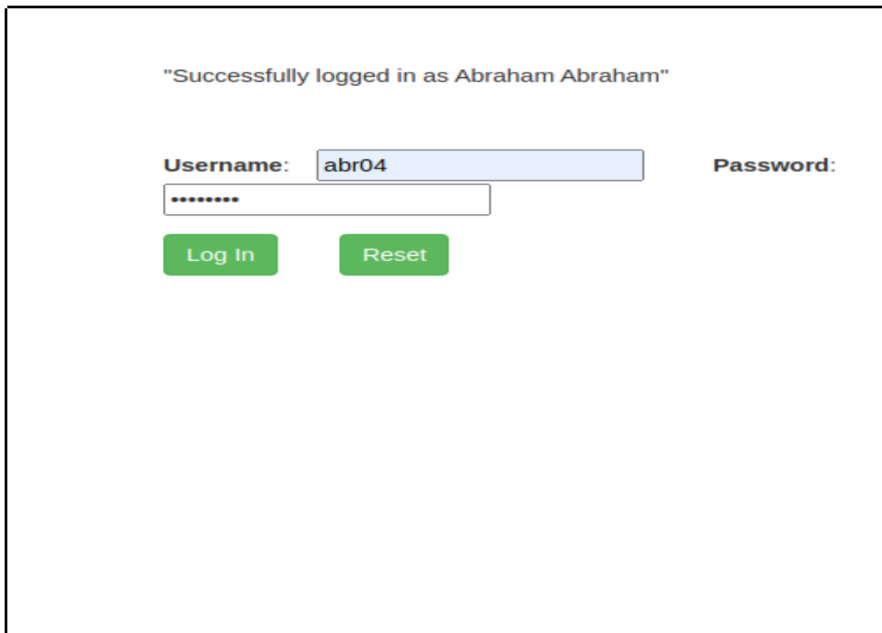
## 1.0 SQL Injection Mitigations

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 screenshot shows a web application interface with a login form. At the top, a message reads: "Successfully logged in as Abraham Abraham". Below this, the form has two input fields: "Username:" with the value "abr04" and "Password:" with a masked password "\*\*\*\*\*". There are two green buttons: "Log In" and "Reset".

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).

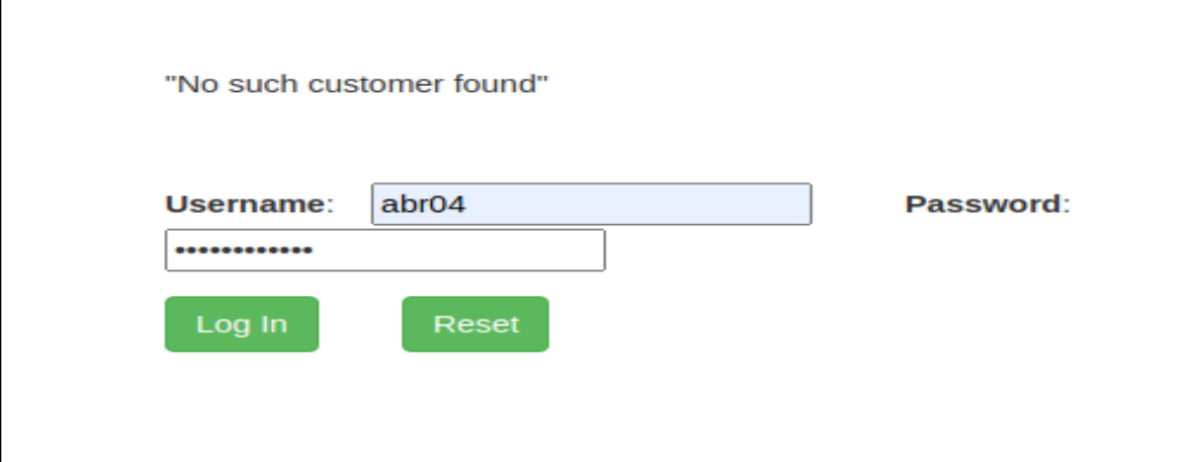
```
@PostMapping("/")
@ResponseBody
public Map<String, String> sql_logged_in(@RequestParam String employee_username, @RequestParam String employee_password) {
    String queryString = "SELECT * From Employees where username = '" + employee_username + "' and password = '" + employee_password + "'";
    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).

```
@ResponseBody  
public Map<String, String> sql_logged_in(@RequestParam String employee_username, @RequestParam  
    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:



The screenshot shows a web application interface with a message at the top that reads "No such customer found". Below this, there is a login form. The "Username:" label is followed by a text input field containing the text "abr04". The "Password:" label is followed by a password input field filled with dots. At the bottom of the form, there are two green buttons: "Log In" and "Reset".

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.

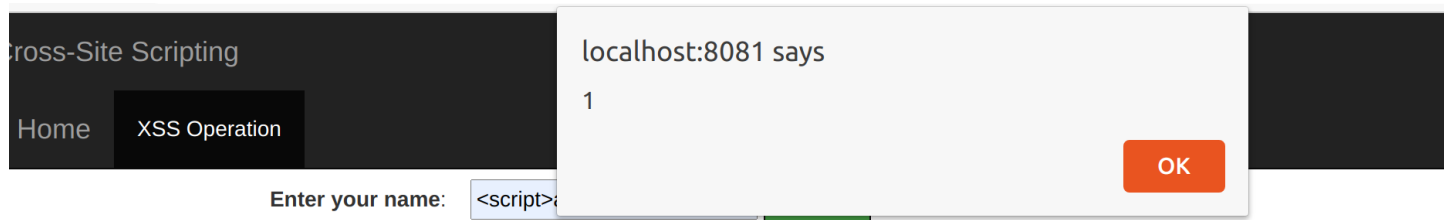


Cross-Site Scripting

Home XSS Operation

Enter your name:

Submit



Cross-Site Scripting

Home XSS Operation

Enter your name:

localhost:8081 says

1

OK

## Into Textarea

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

- *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 to 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.

## By body

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

`<script>alert(1)</script>`

Enter your name:

```

@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:

```

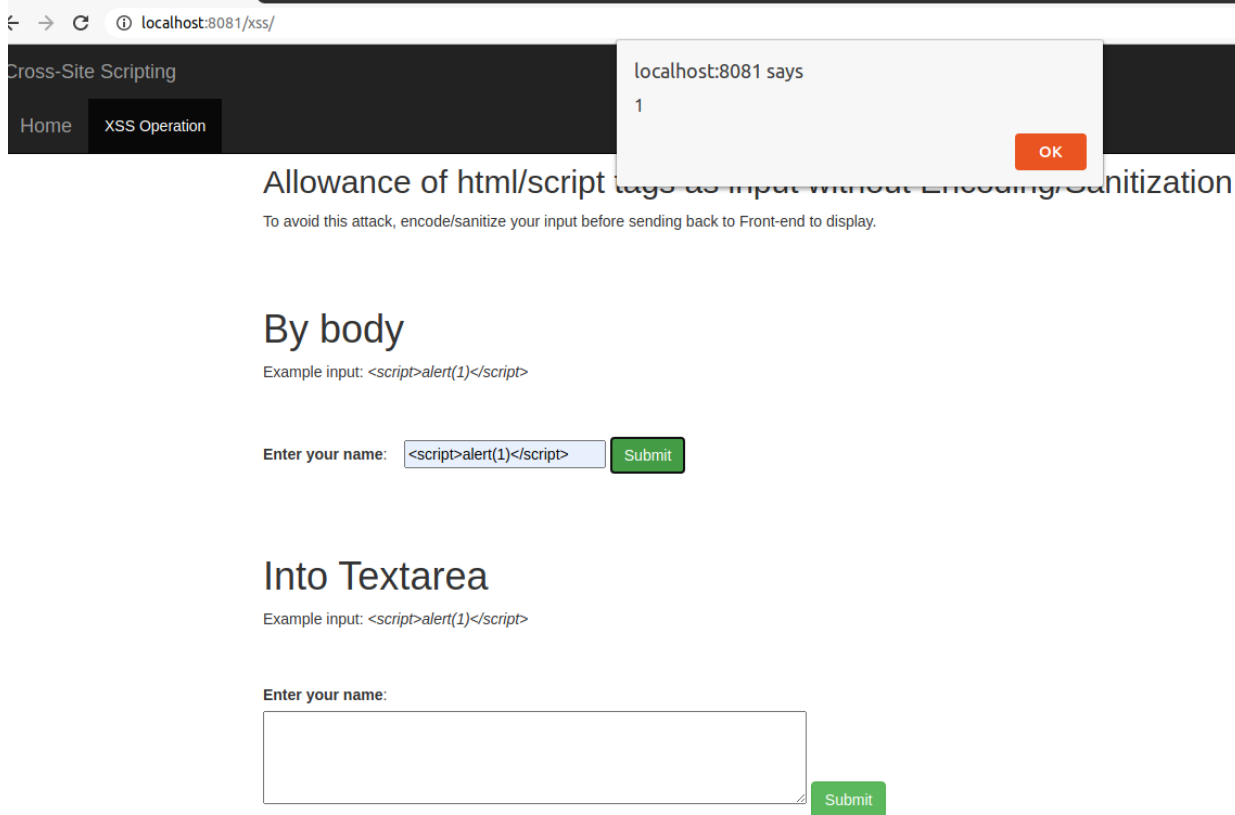
@GetMapping("/")
public String xss_index() { return "xss/index"; }

@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.*



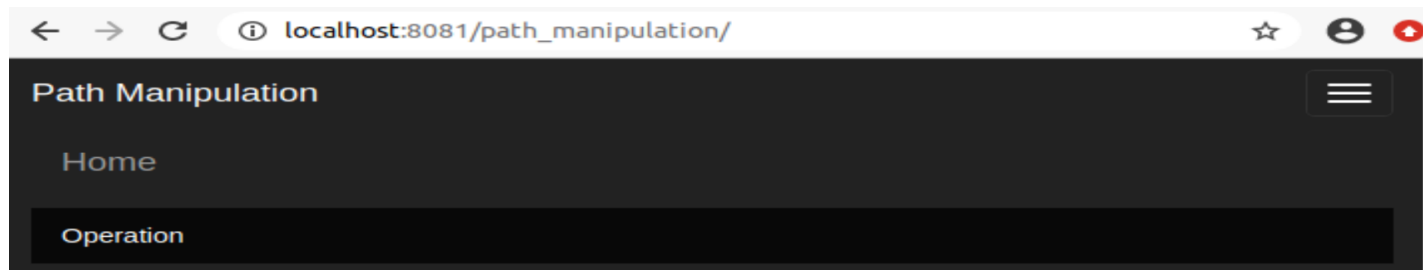
Access database information to exploit: ../application.properties

# Inject Command

Filename:

Submit

localhost:8081



# Inject Command

```
{ "mpurba@xyz.com": { "id": "886459", "password": "123", "name":  
"Moumita Das", "phone": "980-126-5874", "amount": "$52,000",  
"performance": "Job knowledge: F  
Work quality: S  
Attendance: E  
Communication: S" }, "ashu@xyz.com": { "id": "886359", "password": "456",  
"name": "Ashutosh Dutta", "phone": "980-223-4597", "amount": "$51,000",  
"performance": "Job knowledge: S  
Work quality: F  
Attendance: E  
Communication: S" }, "ruhani@xyz.com": { "id": "886460", "password":  
"789", "name": "Ruhani Faiheem", "phone": "980-648-3458", "amount":  
"$50,000", "performance": "Job knowledge: S  
Work quality: F  
Attendance: E  
Communication: S" }
```

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.*

# Inject Command

IP Address:

Submit

localhost:8081/.../command\_in...

# 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=53 time=16.8 ms

64 bytes from 8.8.8.8: icmp\_seq=2 ttl=53 time=18.0 ms

64 bytes from 8.8.8.8: icmp\_seq=3 ttl=53 time=16.5 ms

--- 8.8.8.8 ping statistics ---

3 packets transmitted, 3 received, 0% packet loss, time 2002ms

rtt min/avg/max/mdev = 16.463/17.077/17.956/0.637 ms

logs

Penetration\_test.iml

pom.xml

README.md

src

target

itis42215221

IP Address:

Submit

- *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.*

### Attack by forging Log files

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

### Inject Log

Value:

Submit

---

### Attack by forging Log files

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

### Inject Log

Successfully logged error

Value:

Submit

## 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*

# Attack by forging Log files

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

## Inject Log

Successfully logged error

Value:

Submit

### 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
INFO - After exception: twenty-one%250a%250aINFO%3A%2BUser%2Blogged%2Bout%253dbadguy
INFO - After exception: twenty-one%250a%250aINFO%3A%2BUser%2Blogged%2Bout%253dbadguy
```

*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.*

### Inject XPath

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

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

Email:

Submit

</customers>

### Inject XPath

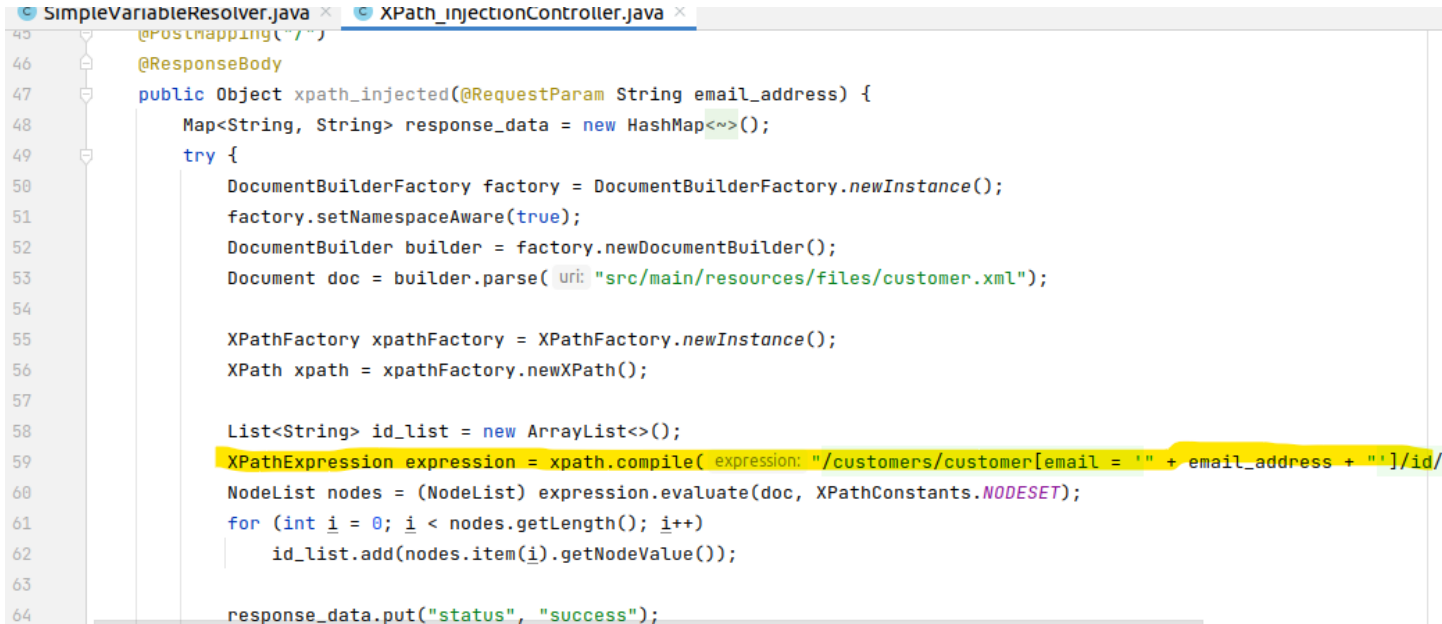
(Example 1: `mpurba@xyz.com'` or `email = 'ashu@xyz.com'`)

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

[886459, 886460]

Email:

Submit



```

45  @PostMapping("/{id}")
46  @ResponseBody
47  public Object xpath_injected(@RequestParam String email_address) {
48      Map<String, String> response_data = new HashMap<>();
49      try {
50          DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
51          factory.setNamespaceAware(true);
52          DocumentBuilder builder = factory.newDocumentBuilder();
53          Document doc = builder.parse(new File("src/main/resources/files/customer.xml"));
54
55          XPathFactory xpathFactory = XPathFactory.newInstance();
56          XPath xpath = xpathFactory.newXPath();
57
58          List<String> id_list = new ArrayList<>();
59          XPathExpression expression = xpath.compile("/customers/customer[email = '" + email_address + "']/id/text()");
60          NodeList nodes = (NodeList) expression.evaluate(doc, XPathConstants.NODESET);
61          for (int i = 0; i < nodes.getLength(); i++)
62              id_list.add(nodes.item(i).getNodeValue());
63
64          response_data.put("status", "success");

```

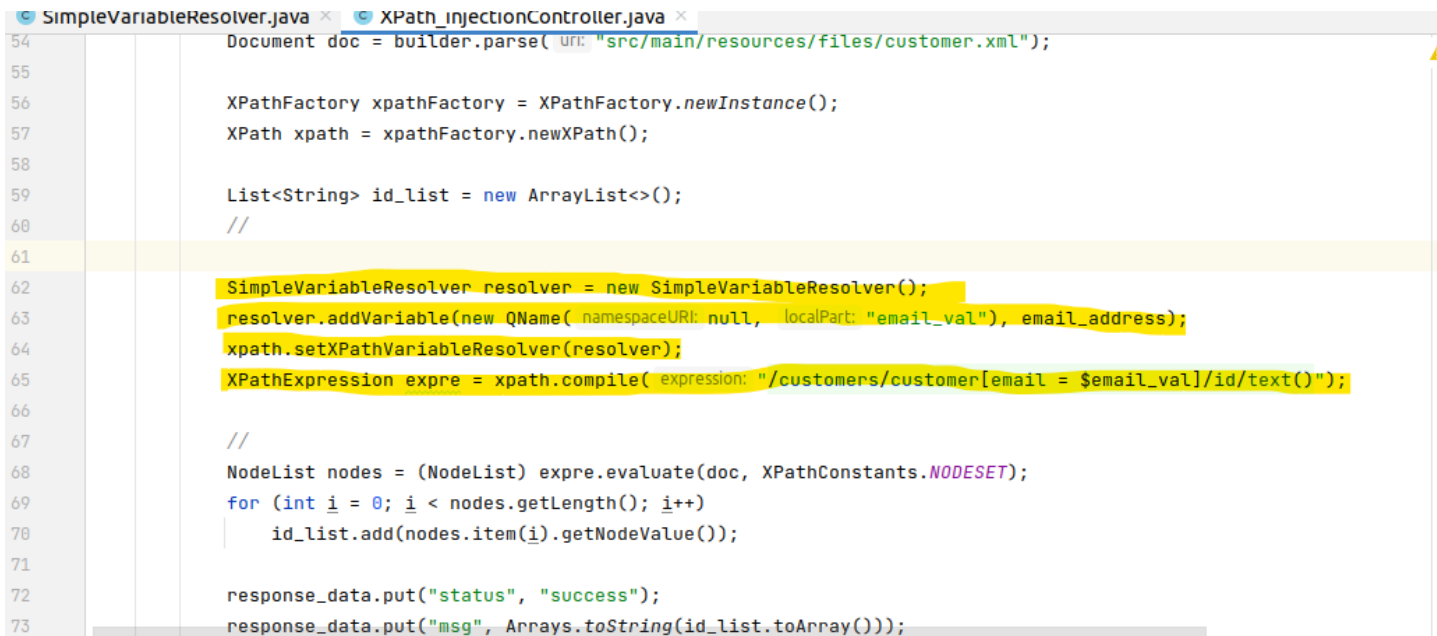
- Changing the code to : `SimpleVariableResolver resolver = new SimpleVariableResolver();`

`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.*



```

54  Document doc = builder.parse(new File("src/main/resources/files/customer.xml"));
55
56  XPathFactory xpathFactory = XPathFactory.newInstance();
57  XPath xpath = xpathFactory.newXPath();
58
59  List<String> id_list = new ArrayList<>();
60  //
61
62  SimpleVariableResolver resolver = new SimpleVariableResolver();
63  resolver.addVariable(new QName(null, "email_val"), email_address);
64  xpath.setXPathVariableResolver(resolver);
65  XPathExpression expre = xpath.compile("/customers/customer[email = $email_val]/id/text()");
66
67  //
68  NodeList nodes = (NodeList) expre.evaluate(doc, XPathConstants.NODESET);
69  for (int i = 0; i < nodes.getLength(); i++)
70      id_list.add(nodes.item(i).getNodeValue());
71
72  response_data.put("status", "success");
73  response_data.put("msg", Arrays.toString(id_list.toArray()));

```



- This code fixes the vulnerability:

</customers>

## Inject XPath

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

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



Email:

Submit

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.

First Name:

Email:

Comment:

Submit

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

just some text

Comment:

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.*

```
@PostMapping("/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(regex: "\\n");
    String y="";
    for (int i = 0; i < split.length; i++) {
```

*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 `normalizeSpace()` or a `replace`, but `normalizeSpace` has it built in.

```
Path_manipulationController.java x SmtplibController.java x Log_injectionController.java x
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 String
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++) {
40     y = y + split[i] + "\n";
}
```

- Rerun the exploitation and show that it is now fixed

SMTP Header Injection

Home

From:Chase Blackwelderbcc:attackExample@gmail.com  
to:mpurba@xyz.com  
Message:123

First Name:

Email:

Comment:

