Preventing Injection Attacks Report - Tunestore

Preventing Injection Attacks

Shashank Mondrati

ITIS: 4221

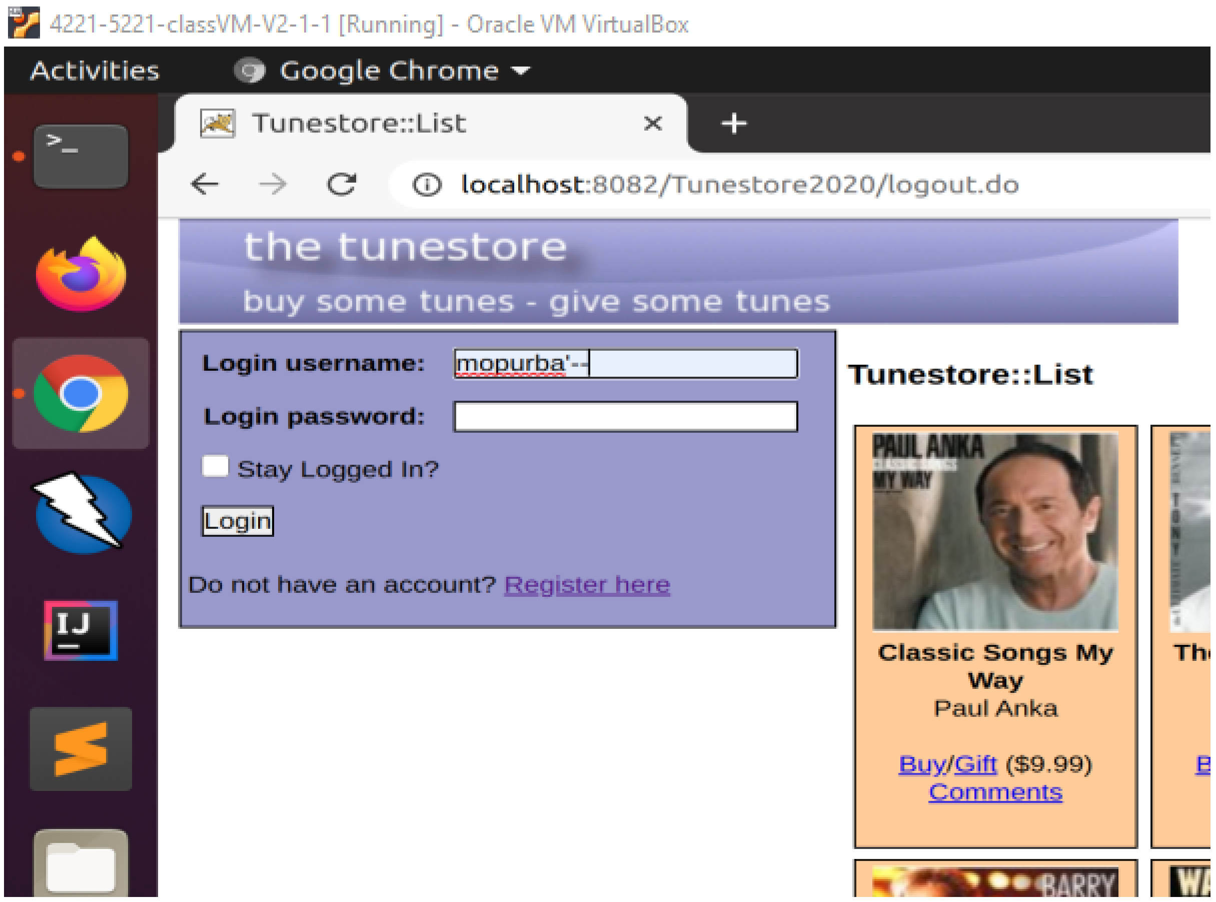
March 2022

Preventing Injection Attacks Report

TABLE OF CONTENTS

1.0 SQL Injection Mitigations

In the current version of the Tunestore application, 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: You can easily login without the password just by doing this (username'--).


4221-5221-classVM-V2-1-1 [Running] - Oracle VM VirtualBox

localhost:8082/Tunestore2020/logout.do

Login username: mopurba'--

Login password:

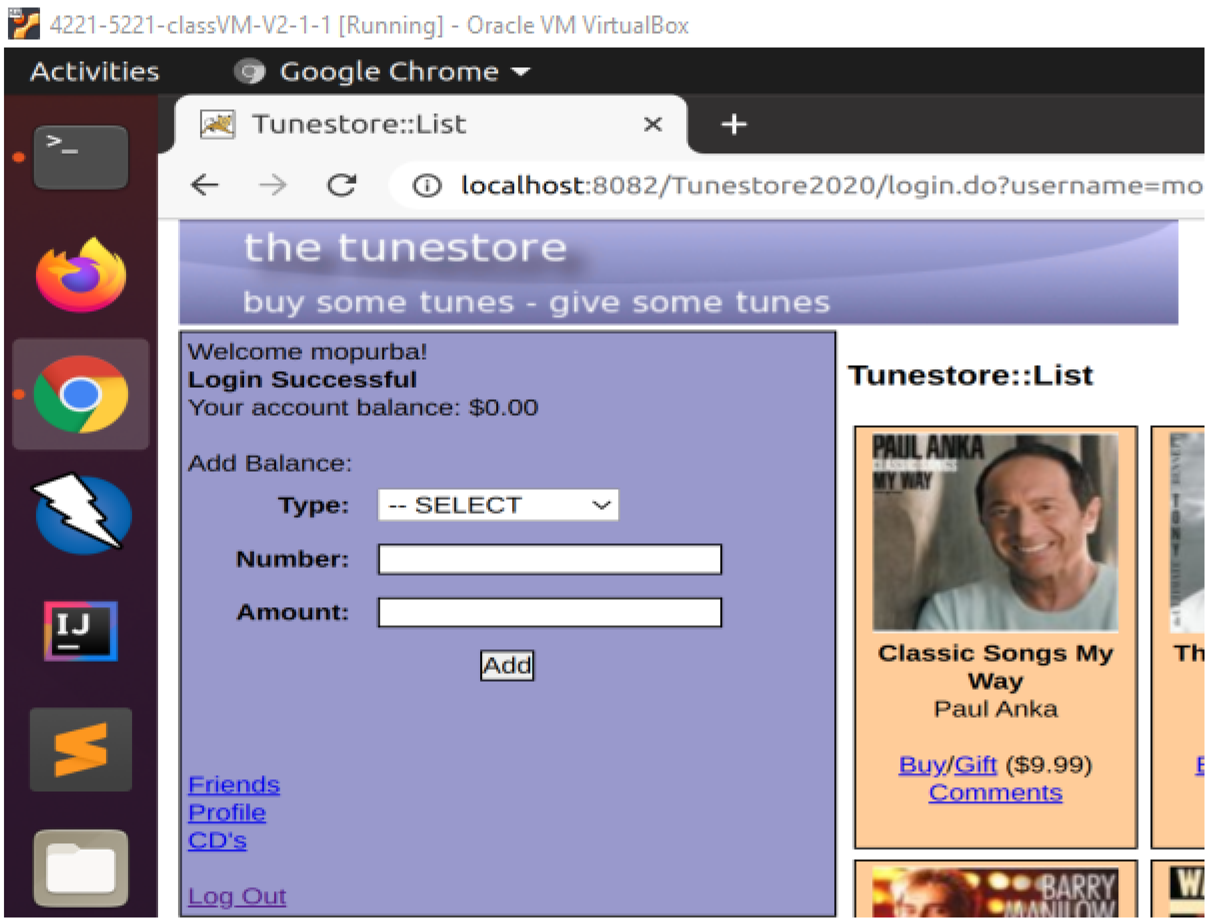
Stay Logged In?

Do not have an account? Register here

Tunestore::List

Classic Songs My Way
Paul Anka
Buy/Gift ($9.99)
Comments

Below is the screenshot where it shows you are logged in,


4221-5221-classVM-V2-1-1 [Running] - Oracle VM VirtualBox

Activities Google Chrome

Tunestore::List

localhost:8082/Tunestore2020/login.do?username=mo

the tunestore

buy some tunes - give some tunes

Welcome mopurba!
Login Successful
Your account balance: $0.00

Add Balance:

Type:	-- SELECT
Number:	
Amount:	

Add

Friends
Profile
CD's

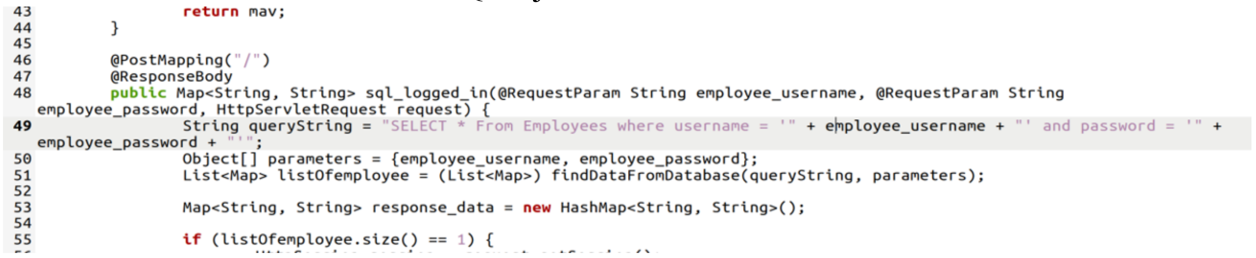
Log Out

Tunestore::List

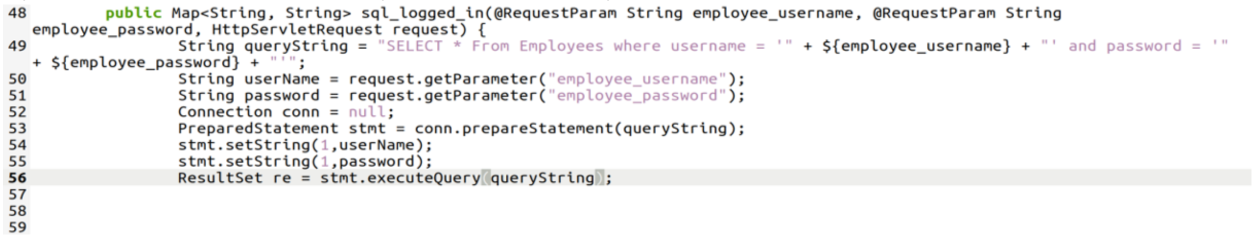
Classic Songs My Way
Paul Anka

Buy/Gift ($9.99)
Comments

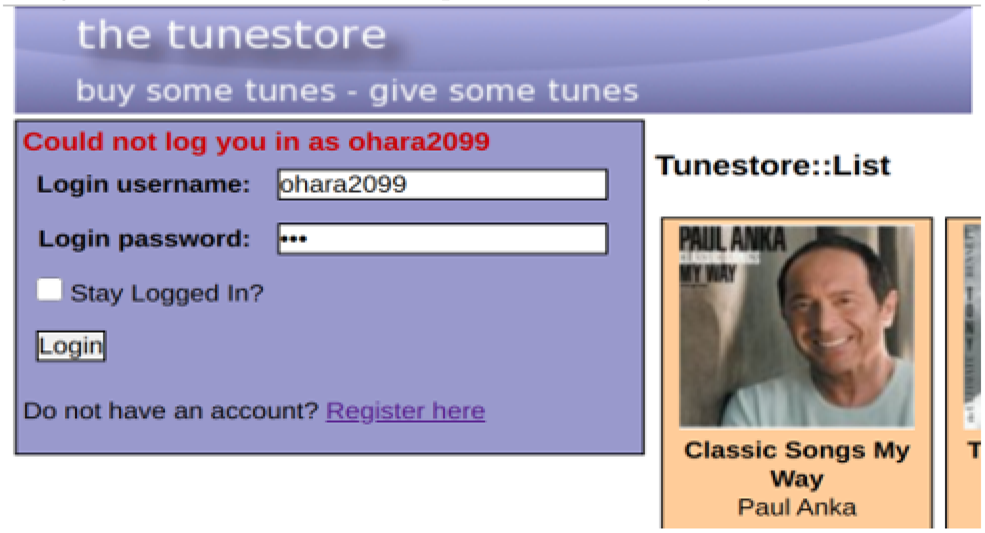
Below is the screenshot where the SQL injection occurs


```
                return mav;
        }

        @PostMapping("/")
        @ResponseBody
        public Map<String, String> sql_logged_in(@RequestParam String employee_username, @RequestParam String
employee_password, HttpServletRequest request) {
                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>();

                if (listOfemployee.size() == 1) {
```

By sanitizing the code, and adding Prepared Statement if would stop SQL injection attacks. Below is the screenshot where the data is sanitized and the clean code is added from preventing injection attacks in the SQLInjectionController.java


```
        public Map<String, String> sql_logged_in(@RequestParam String employee_username, @RequestParam String
employee_password, HttpServletRequest request) {
                String queryString = "SELECT * From Employees where username = '" + ${employee_username} + "' and password = '"
+ ${employee_password} + "'";
                String userName = request.getParameter("employee_username");
                String password = request.getParameter("employee_password");
                Connection conn = null;
                PreparedStatement stmt = conn.prepareStatement(queryString);
                stmt.setString(1,userName);
                stmt.setString(1,password);
                ResultSet re = stmt.executeQuery(queryString);
```

By sanitizing the code, the vulnerability should not work. Below is the screenshot where I tried to log in as a normal user with the password added. As you can see it did not let me log in.



Could not log you in as ohara2099

Login username: ohara2099

Login password: •••

Stay Logged In?

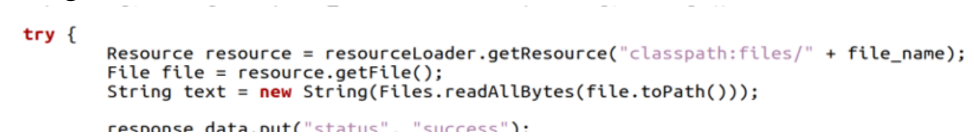
Login

Do not have an account? Register here

2.0 Path Manipulation Mitigation

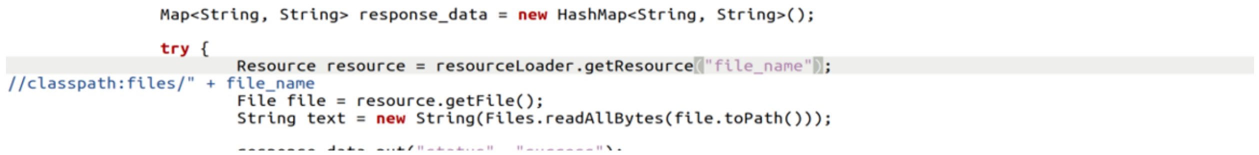
Path manipulation can occur when the attacker has the file name at the address your original file is located. When they have the address they can just change the address with their file, and it will result in changing the path.

To prevent this, we have to make sure the file is a real file name, rather than (xyz/sya/dasl.xml) That is just for representations. Below is the screenshot where the attacker can change the name to their liking.



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

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

Below is a screenshot where the fileName has to be a real file name so it cannot be attacked. If the file name has an address it can be changed by attacker just by altering the address of the file with a malicious code.

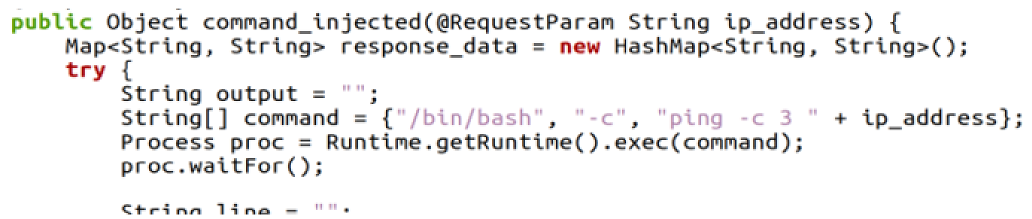


```
            Map<String, String> response_data = new HashMap<String, String>();

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

All these changes were made Path_manipulationController.java

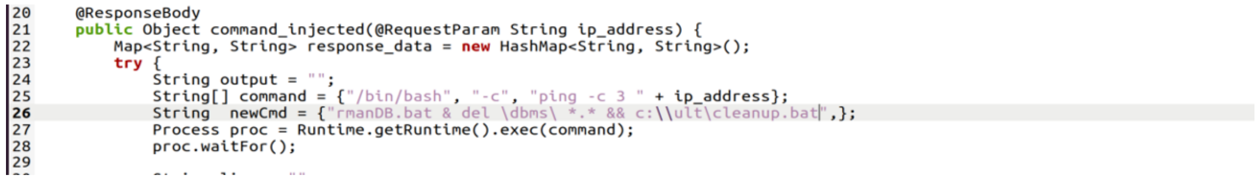
3.0 Command Injection Mitigation

Command injection is a cyber attack that involves executing arbitrary commands on a host operating system (OS). Typically, the threat actor injects the commands by exploiting an application vulnerability, such as insufficient input validation. Below is a screenshot of the code in the try catch block, by altering the commands they can exploit the weaknesses in the OS



```
public Object command_injected(@RequestParam String ip_address) {
    Map<String, String> response_data = new HashMap<String, String>();
    try {
        String output = "";
        String[] command = {"/bin/bash", "-c", "ping -c 3 " + ip_address};
        Process proc = Runtime.getRuntime().exec(command);
        proc.waitFor();

        String line = "";
```

Below is a screenshot of the new modified code where it is protected by sanitizing the code by adding a few other commands. In Line 26 are the new commands if someone tries to hack in, if they do the data will be handled by the bat files.



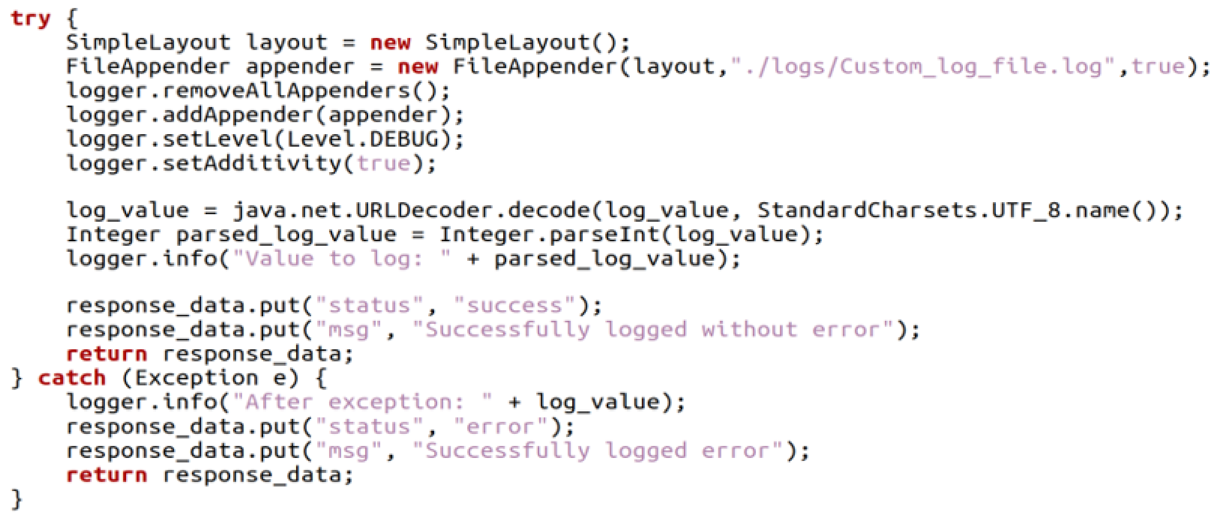
```
20      @ResponseBody
21      public Object command_injected(@RequestParam String ip_address) {
22          Map<String, String> response_data = new HashMap<String, String>();
23          try {
24              String output = "";
25              String[] command = {"/bin/bash", "-c", "ping -c 3 " + ip_address};
26              String  newCmd = {"rmanDB.bat & del \dbms\ *.* && c:\\ult\cleanup.bat",};
27              Process proc = Runtime.getRuntime().exec(command);
28              proc.waitFor();
29
```

4.0 Log Forging Mitigation

Log Forging or injection is an attack where a technique of writing unvalidated user input to log files so that it can allow an attacker to forge log entries or inject malicious content into the logs. It can change the user's log by changing the code in the try catch or the victims code. WE can say the log was supposed to say the answer was 5 in a basic math operation. But when the attacker uses his code to display something else in the code.

Below is the code we can use to prevent log forging attacks. Val is nothing but the value of the log we get from executing the code.
log.info("log faile" + val);

Below is the screenshot where user can be able to change their logs from the Log_injectionController.java

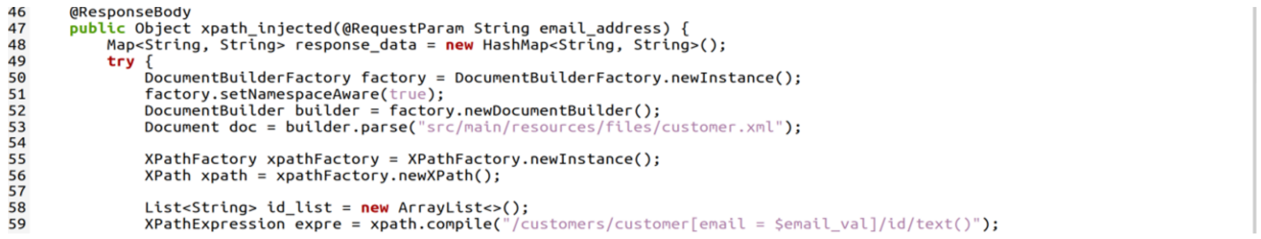


```java
try {
    SimpleLayout layout = new SimpleLayout();
    FileAppender appender = new FileAppender(layout,"./logs/Custom_log_file.log",true);
    logger.removeAllAppenders();
    logger.addAppender(appender);
    logger.setLevel(Level.DEBUG);
    logger.setAdditivity(true);

    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.info("After exception: " + log_value);
    response_data.put("status", "error");
    response_data.put("msg", "Successfully logged error");
    return response_data;
}
```

To prevent this we just need to add or show what the log is to the value, by using the info method in java.

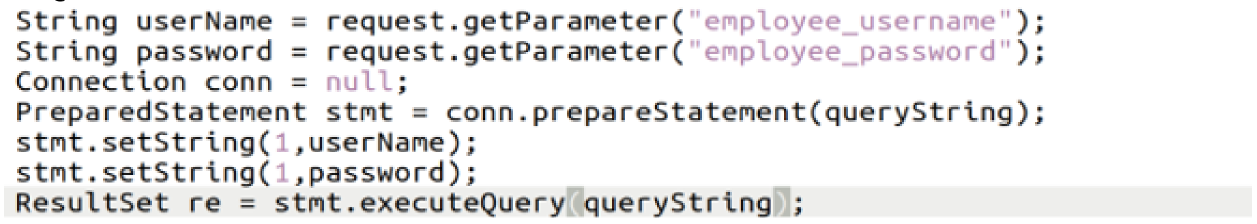
5.0 XPath Injection Mitigation

XPath injection is a type of attack where a malicious input can lead to un-authorised access or exposure of sensitive information such as structure and content of XML document. It occurs when user's input is used in the construction of the query string. Below is the code at line 59 where there is a vulnerability which gives access of user’s email and the email that can be sent from that address.



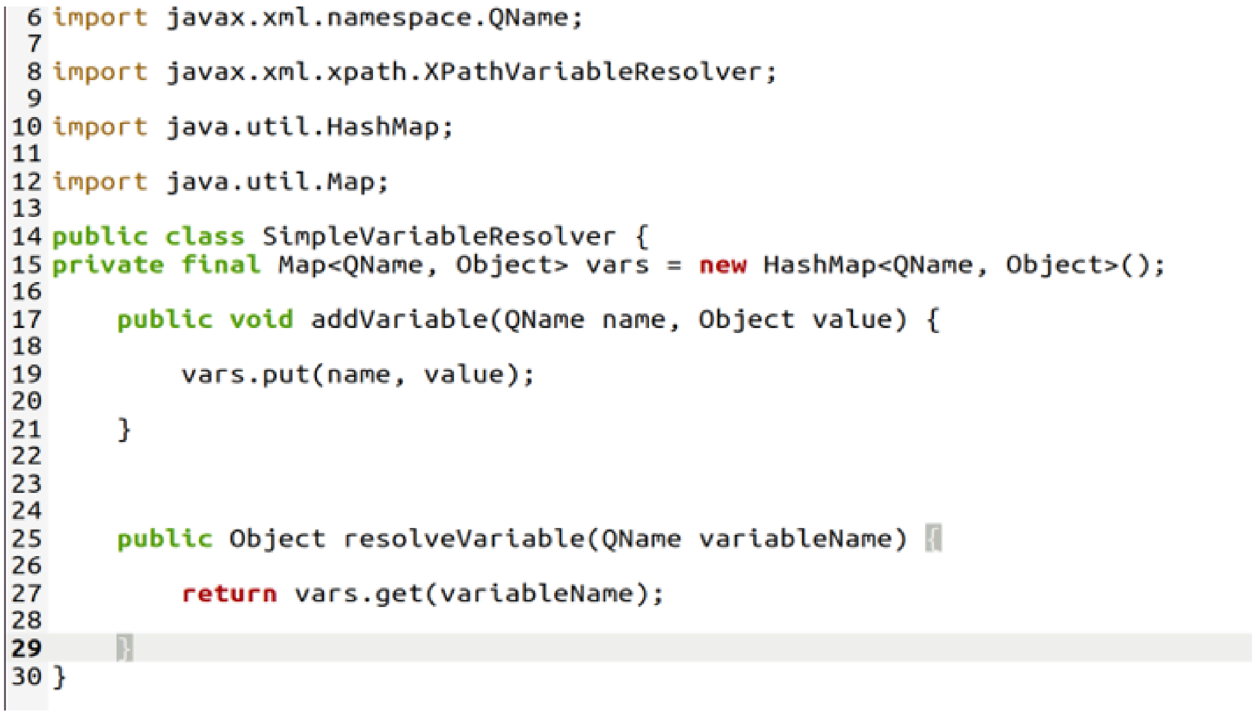
```java
46      @ResponseBody
47      public Object xpath_injected(@RequestParam String email_address) {
48          Map<String, String> response_data = new HashMap<String, String>();
49          try {
50              DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
51              factory.setNamespaceAware(true);
52              DocumentBuilder builder = factory.newDocumentBuilder();
53              Document doc = builder.parse("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 expre = xpath.compile("/customers/customer[email = $email_val]/id/text()");
```

By using Parameterized SQL we can prevent this from happening. Below is a screenshot of a SQL.



```java
String userName = request.getParameter("employee_username");
String password = request.getParameter("employee_password");
Connection conn = null;
PreparedStatement stmt = conn.prepareStatement(queryString);
stmt.setString(1,userName);
stmt.setString(1,password);
ResultSet re = stmt.executeQuery(queryString);
```

With this the code is safe from attackers.



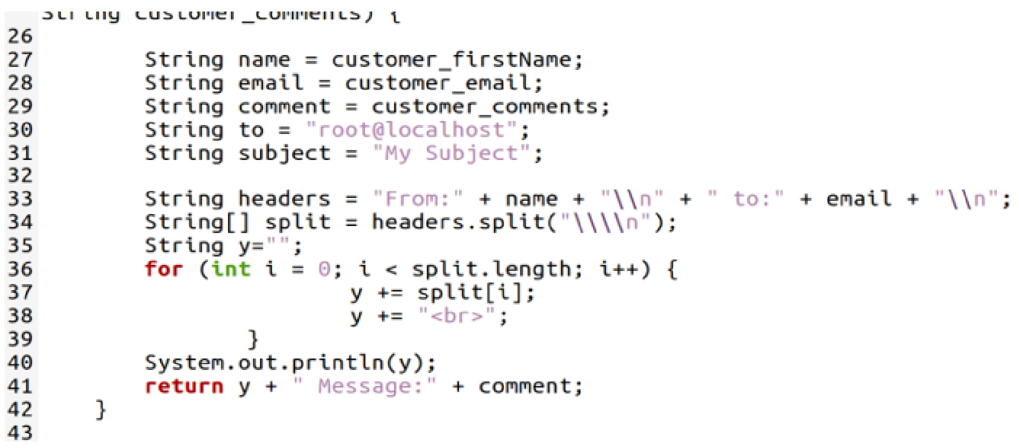
```
import javax.xml.namespace.QName;

import javax.xml.xpath.XPathVariableResolver;

import java.util.HashMap;

import java.util.Map;

public class SimpleVariableResolver {
private final Map<QName, Object> vars = new HashMap<QName, Object>();

    public void addVariable(QName name, Object value) {

        vars.put(name, value);

    }



    public Object resolveVariable(QName variableName) {

        return vars.get(variableName);

    }
}
```

By using the code from above we can assign name to values for like adding name to a object to we can resolve XPath variables when injection or when injected by attackers.

6.0 SMTP Header Injection Mitigation

SMTP header injection vulnerabilities arise when user input is placed into email headers without adequate sanitization, allowing an attacker to inject additional headers with arbitrary values. Below is the screenshot of the application SmtpController.java where the user can enter their name, email or the comment to the address they want to send to.

But when they decide to do it, they mistakenly send themselves the email too. To get rid of this we just need to get rid of the from part in the code. This way it can go from only one user that is the person who sends it.



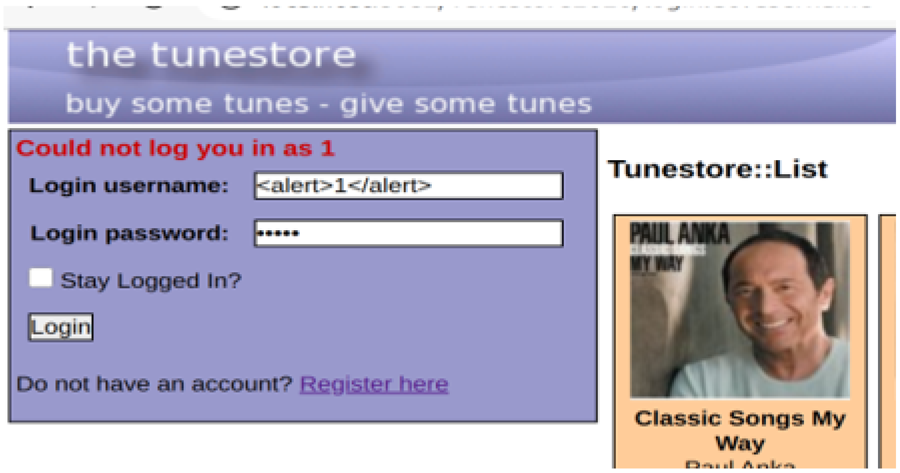
```
    String customer_comments) {

        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;
    }

```

7.0 XSS Mitigation

Overview. Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user.

To prevent XSS, we can use web applications that are single paged that are hardocre react or JS.

Below is the screenshot of an XSS Vulnerability where the attacker can use malicious JS code to steal the victim's credentials.


the tunestore
buy some tunes - give some tunes
Tunestore::List
Classic Songs My Way
Stay Logged In?
Login
Do not have an account? Register here

Could not log you in as 1

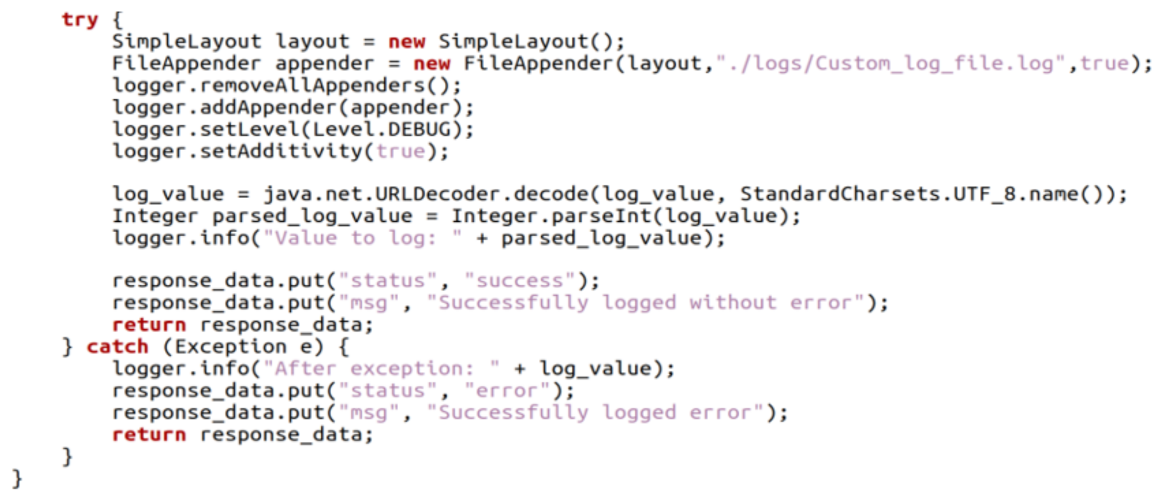
Login username: <alert>1</alert>

Login password: •••••

9.0 Exception Handling

Exception handling ensures that the flow of the program doesn't break when an exception occurs. For example, if a program has bunch of statements and an exception occurs mid way after executing certain statements then the statements after the exception will not execute and the program will terminate abruptly.

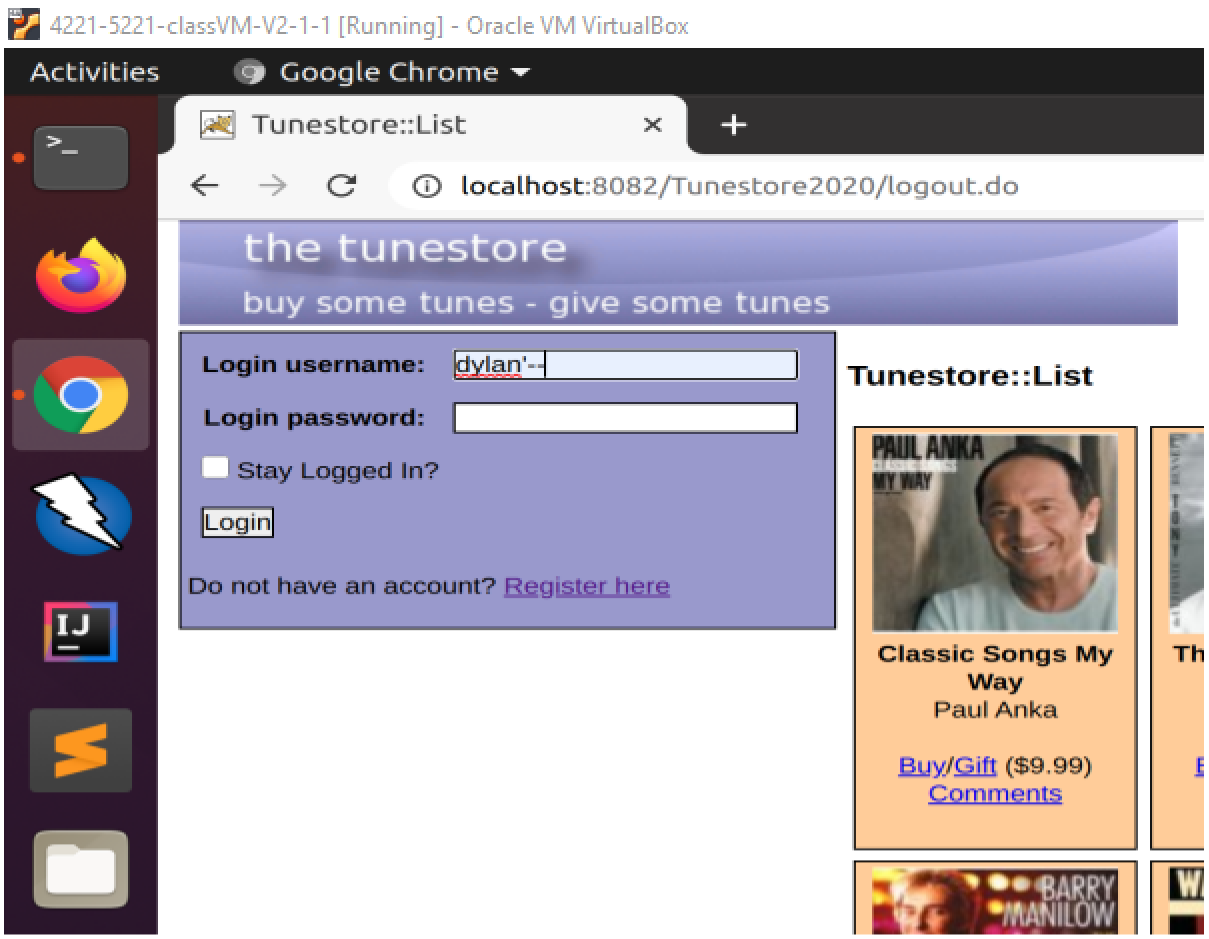
Below is the screenshot we have implemented the try-catch block to try and see if the program catches any exceptions. This try-catch is from Log_controller.java to try control logs, if it gets out of hand the catch block will just throw the exception rather than crashing the entire program.



```
        try {
            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.info("After exception: " + log_value);
            response_data.put("status", "error");
            response_data.put("msg", "Successfully logged error");
            return response_data;
        }
    }
```

1. **Demonstrate that using the incorrect encoding function will not fix vulnerability.**

The way I see this is, when we code something the compiler does its job, but it will not fix the leak in the wall. It can be in many fields. We can use the leak in the login to login easily by using this ('--). Below is the screenshot where the user can login easily.


4221-5221-classVM-V2-1-1 [Running] - Oracle VM VirtualBox
Activities
Google Chrome
Tunestore::List
localhost:8082/Tunestore2020/logout.do
the tunestore
buy some tunes - give some tunes
Tunestore::List
Classic Songs My Way
Paul Anka
Buy/Gift ($9.99)
Comments

Login username: dylan'--

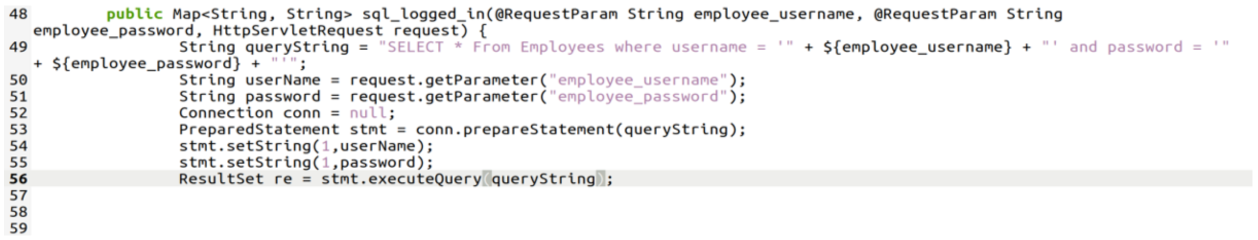
Login password:

Stay Logged In?

Login

Do not have an account? Register here

If there was a mistake in the **Parameterized SQL** statement which is mentioned belows in the screenshot it will not fix the vulnerabilities that allows the attacker to login to get access to almost anyone's account.



```
        public Map<String, String> sql_logged_in(@RequestParam String employee_username, @RequestParam String
employee_password, HttpServletRequest request) {
                String queryString = "SELECT * From Employees where username = '" + ${employee_username} + "' and password = '"
+ ${employee_password} + "'";
                String userName = request.getParameter("employee_username");
                String password = request.getParameter("employee_password");
                Connection conn = null;
                PreparedStatement stmt = conn.prepareStatement(queryString);
                stmt.setString(1,userName);
                stmt.setString(1,password);
                ResultSet re = stmt.executeQuery(queryString);
```