# Project 3

By: Mikiyas Solomon
Class: ITIS 4221
Date: May 8, 2023

# VULNERABILITY ASSESSMENT AND SYSTEMS ASSURANCE REPORT

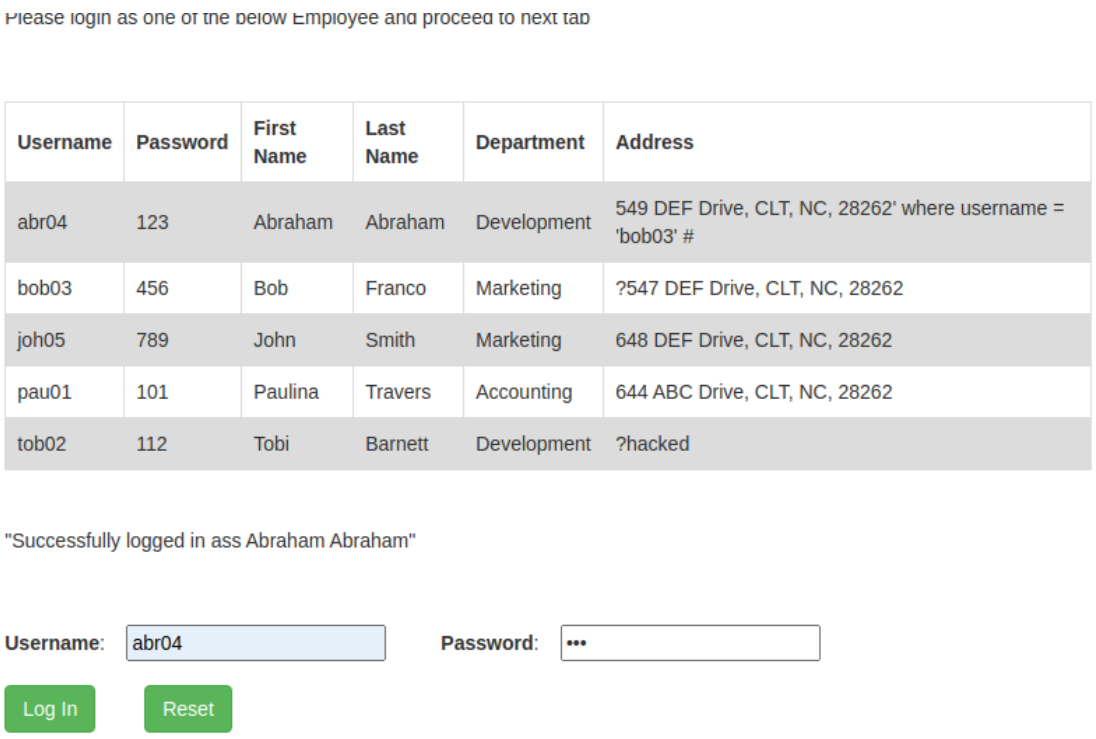
TABLE OF CONTENTS

## 1.0 Purpose

The purpose of this security assessment is to identify Vulnerabilities that ZAP found and the vulnerabilities in the source code.

## 2.0 SQL Injection

- Logging into the SQL injection site, once you login as a employee you are able to update not only your address but other user's address as well by doing an SQL injection



# Update Address

| Username | First Name | Last Name | Department | Address |
|---|---|---|---|---|
| abr04 | Abraham | Abraham | Development | You just got hacked' where username = 'pau01' # |
| bob03 | Bob | Franco | Marketing | ?547 DEF Drive, CLT, NC, 28262 |
| joh05 | John | Smith | Marketing | 648 DEF Drive, CLT, NC, 28262 |
| pau01 | Paulina | Travers | Accounting | ?You just got hacked |
| tob02 | Tobi | Barnett | Development | ?hacked |

**Address**: You just got hacked' where u

Submit

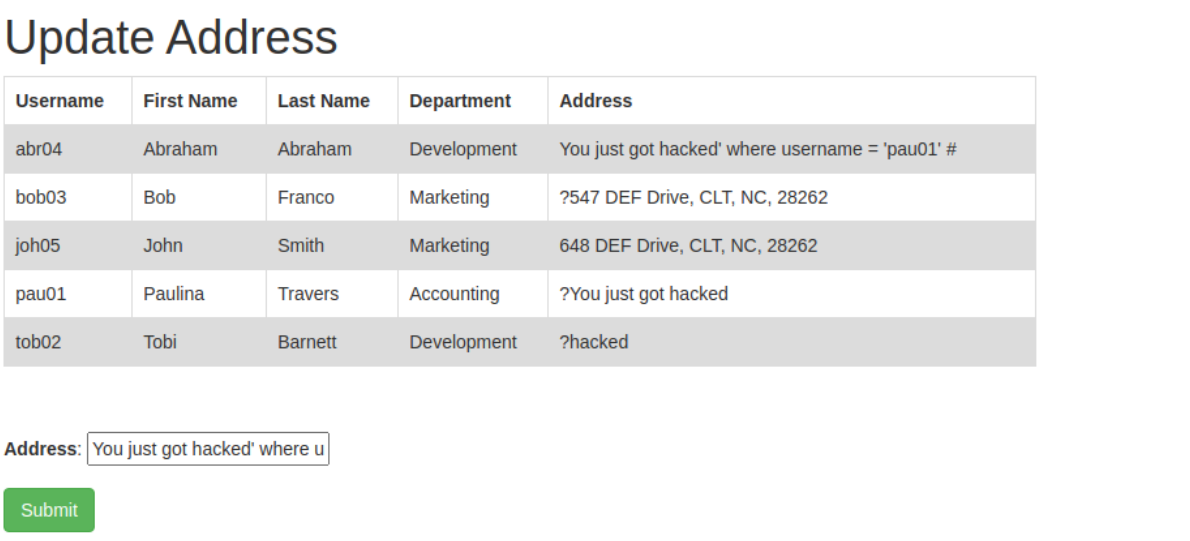
- By inserting this SQL injection “You just got hacked' where username = 'pau01' #” you are able to change someone's address that's not the person logged in as shown in the image above. The image below shows the vulnerable part of this code where any user can update any address and it doesn’t specify that the logged-in user can only change their own address

```
try {
    int updatedEmpInfo = 0;
    // change 'updateQuery' with by applying '?' instead of direct parameter.
    String updateQuery = "UPDATE Employees SET address = '?" + updated_address + "' WHERE username = '?" + loggedInUser + "'?";
    // change in 'jdbcTemplate.update' function by passing parameters so that dynamic input will not harm database.
    updatedEmpInfo = jdbcTemplate.update(updateQuery);
    if (updatedEmpInfo == 0) {
        return "{\"msg\":\"No rows updated.\"}";
    }
    // change to display only logged-in employee's data
    String selectQuery = "SELECT * From Employees";
    List<Map> employeeList = (List<Map>) findDataFromDatabase(selectQuery,  param: null);
    return new ObjectMapper().writeValueAsString(employeeList);
} catch (Exception e){
    e.printStackTrace();
    return "{\"msg\":\"No rows updated.\"}";
}
```

- The updated code below fixes the vulnerability by making the Update employee address into a direct parameter and making sure that only a logged-in user can edit their own address

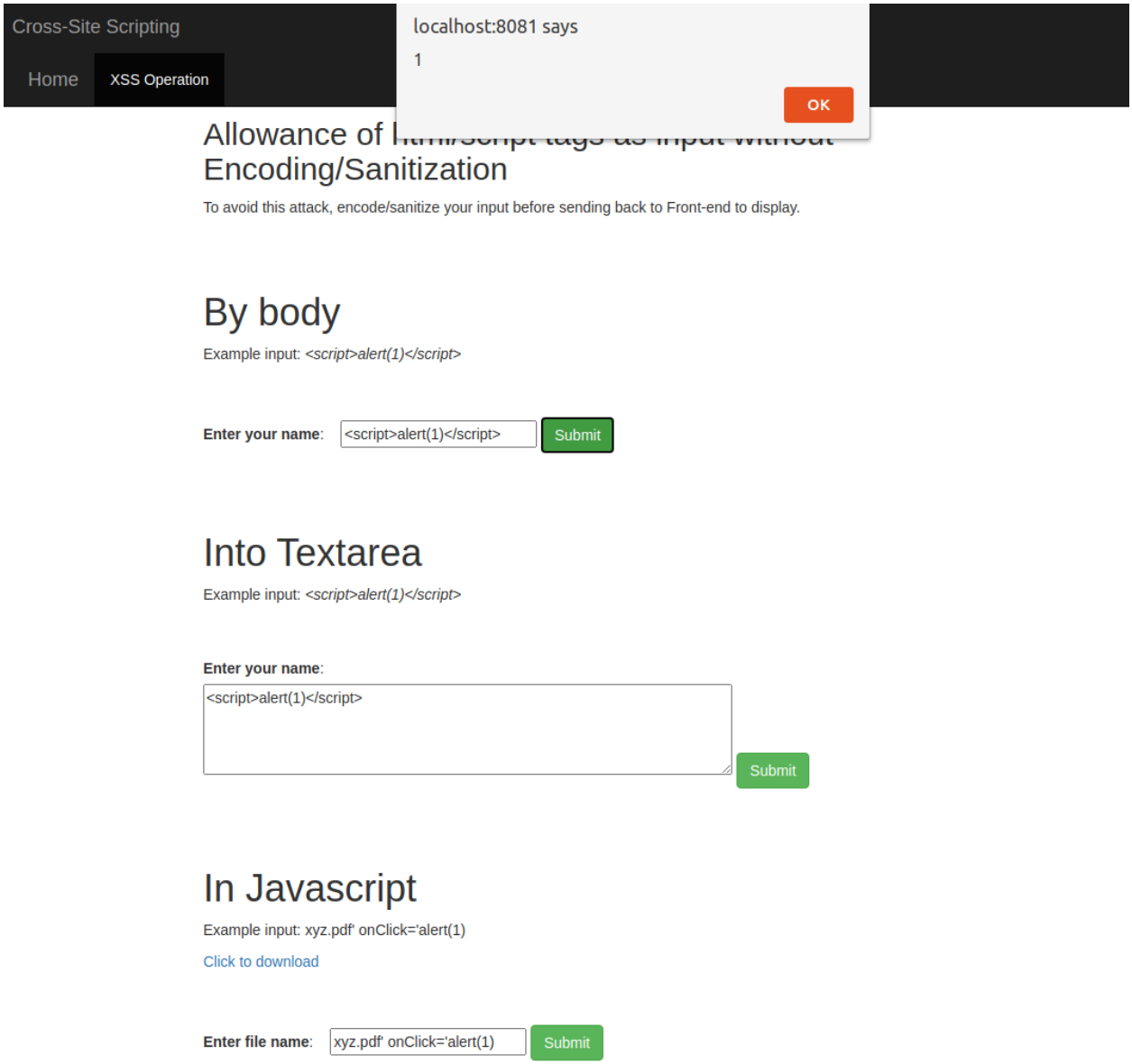
```
try {
    int updatedEmpInfo = 0;
    // change 'updateQuery' with by applying '?' instead of direct parameter.
    String updateQuery = "UPDATE Employees SET address = ? WHERE username = ?";
    // change in 'jdbcTemplate.update' function by passing parameters so that dynamic input will not harm database.
    updatedEmpInfo = jdbcTemplate.update(updateQuery,updated_address,loggedInUser);
    if (updatedEmpInfo == 0) {
        return "{\"msg\":\"No rows updated.\"}";
    }
    // change to display only logged-in employee's data
    String selectQuery = "SELECT * From Employees";
    List<Map> employeeList = (List<Map>) findDataFromDatabase(selectQuery, loggedInUser);
    return new ObjectMapper().writeValueAsString(employeeList);
} catch (Exception e){
    e.printStackTrace();
    return "{\"msg\":\"No rows updated.\"}";
}

---------- Update address(2nd tab) ----------
```

- Now input the same SQL injection no longer updates other user's addresses and only updates the logged-in users



## 2.1 XSS Prevention

- This site is easily susceptible to cross-site scripting as each input lacks input validation which makes it really easy for an attacker to do what he wants as shown in the pictures below



- Looking into the source code of the vulnerability it's clear that there has been absolutely no attempt at input validation

```
public class XssController {

    no usages
    @GetMapping("/")
    public String xss_index() { return "xss/index"; }

    no usages
    @GetMapping("/body_xss")
    @ResponseBody
    public String body_xss(@RequestParam String body_tagVal) throws Exception {
        return body_tagVal;
    }

    no usages
    @GetMapping("/textarea_xss")
    @ResponseBody
    public Object textarea_xss(@RequestParam String textarea_tagVal) throws Exception {
        return textarea_tagVal;
    }

    no usages
    @GetMapping("/js_xss")
    @ResponseBody
    public Object js_xss(@RequestParam String js_tagVal) throws Exception {
        return js_tagVal;
    }

}
```

- There is a really simple fix for some input validation and that's to match if statements to “^[\\w\\s\\-_]*$” this way the text box will only contain alphanumerical numbers and if it doesn’t it will throw an error.

```java
public class XssController {

    no usages
    @GetMapping("/")
    public String xss_index() { return "xss/index"; }

    no usages
    @GetMapping("/body_xss")
    @ResponseBody
    public String body_xss(@RequestParam String body_tagVal) throws Exception {
        if (!body_tagVal.matches( regex: "^[\\w\\s\\-_]*$")) {
            throw new IllegalArgumentException("Invalid input");
        }
        return body_tagVal;
    }

    no usages
    @GetMapping("/textarea_xss")
    @ResponseBody
    public Object textarea_xss(@RequestParam String textarea_tagVal) throws Exception {
        if (!textarea_tagVal.matches( regex: "^[\\w\\s\\-_]*$")) {
            throw new IllegalArgumentException("Invalid input");
        }
        return textarea_tagVal;
    }

    no usages
    @GetMapping("/js_xss")
    @ResponseBody
    public Object js_xss(@RequestParam String js_tagVal) throws Exception {
        if (!js_tagVal.matches( regex: "^[\\w\\s\\-_]*$")) {
            throw new IllegalArgumentException("Invalid input");
        }
        return js_tagVal;
    }

}
```

- Now trying to implement the same attack fails as all user input has been validated

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

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

# By body

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

Error happend !!

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

# Into Textarea

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

**Enter your name**:

<script>alert(1)</script>

Submit

# In Javascript

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

Error happend !!

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

## 2.2 Command Injection

- The command injection allows you to ping an ip and a console will send a response back based on that ping the problem with it is that there is no input validation on this so an attacker can include malicious commands and gain access to unauthorized files.

# Attack by system command

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

(example: 8.8.8.8 && ls && whoami)

Accepted commands:

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

# Inject Command

PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=128 time=18.5 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=128 time=19.9 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=128 time=20.1 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 18.514/19.509/20.069/0.705 ms
logs
pom.xml
README.md
src
target
itis42215221

**IP Address**: 8.8.8.8 && ls && whoami

Submit

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

- The code above shows the vulnerable part of the code and the code below fixes that by using ProcessBuilder we can sanitize user input

```
Map<String, String> response_data = new HashMap<String, String>();
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.getInput
    BufferedReader errorStream = new BufferedReader(new InputStreamReader(proc.getError
    while ((line = inputStream.readLine()) != null) {
        output += line + "<br/>";
```

# Attack by system command

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

(example: 8.8.8.8 && ls && whoami)

Accepted commands:

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

# Inject Command

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

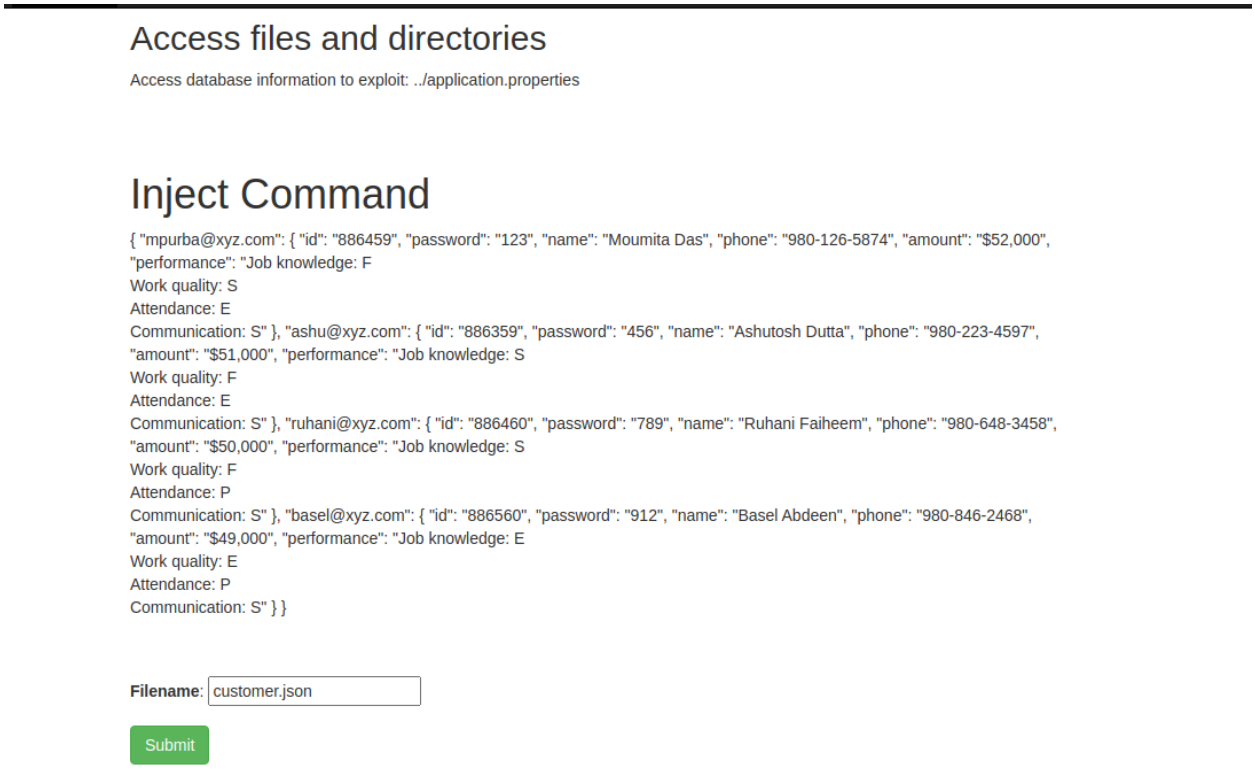
**IP Address**: 8.8.8.8 && ls && whoami

Submit

## 2.3 Path Manipulation

- The site below takes advantage of the lack of input validation to directly access sever files without authorization

---




- The code below shows the vulnerable part of it as all the code does is get the file for the user without verifying what the user wants and if it matches with anything in the server

```java
public Map<String, String> view_file(@RequestParam String file_name) throws Exception {
    Map<String, String> response_data = new HashMap<~>();

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

- In order to mitigate this vulnerability I made a string of allowed characters and if the input matches with anything not in the list then an error is returned

```
String allowedChars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-_";

if (!file_name.matches( regex: "[" + allowedChars + "]+")) {
    response_data.put("status", "error");
    response_data.put("msg", "Invalid input");
    return response_data;
}

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

# Inject Command

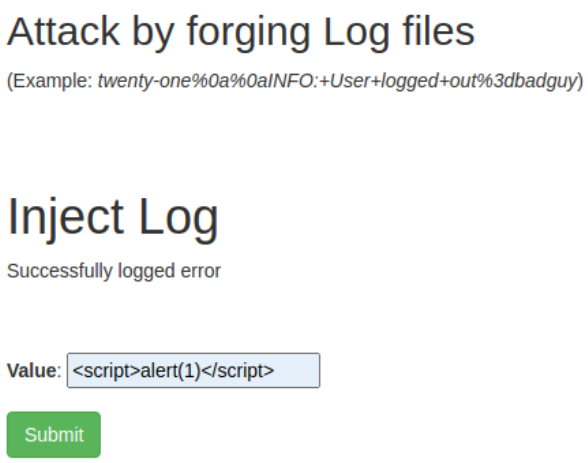
Invalid input

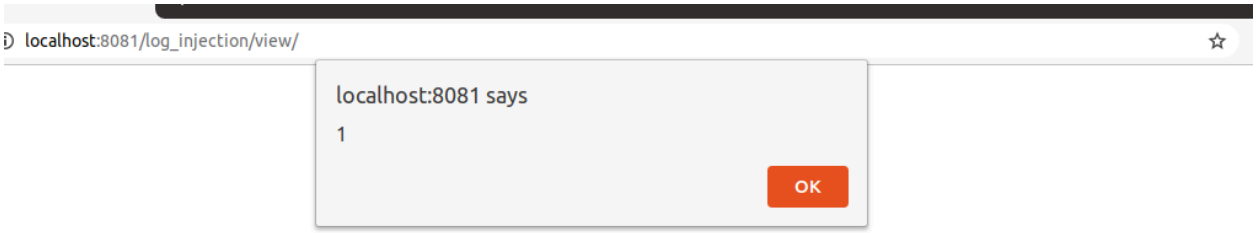
**Filename**: customer.json

Submit

## 2.4 Log Forgery/Injection

- Going to the Log injection page I immediately noticed that this page too is vulnerable as there is no input sanitation so anything you type can attack this site. As an example, I did a simple script into an alert and it went through



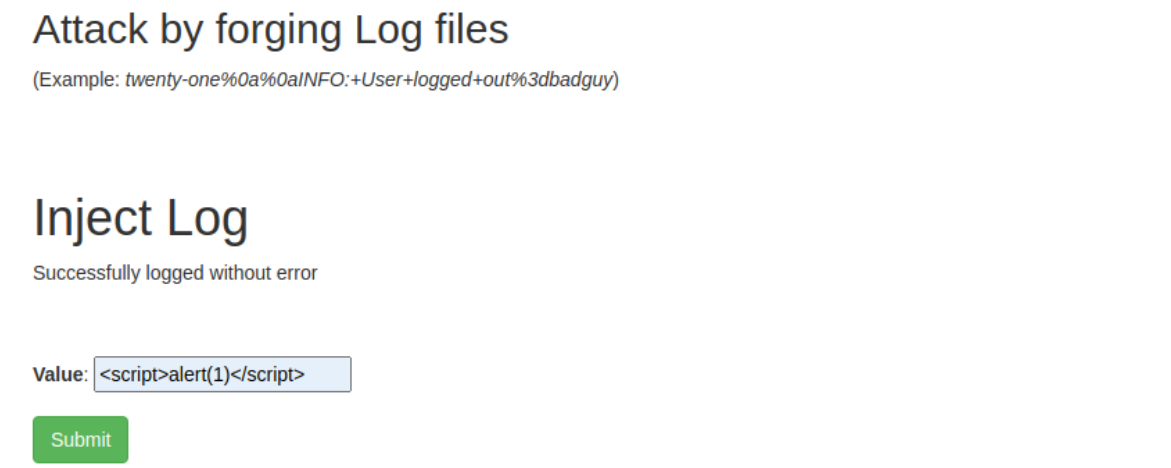



- After further examination of the source code, I found the vulnerable part of the code in the image below

```
try {
    SimpleLayout layout = new SimpleLayout();
    FileAppender appender = new FileAppender(layout, filename: "./logs/Custom_log_file.log", append
    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");
```

- All that was left to do was sanitize the user input so certain characters won't go through and the vulnerability was patched

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

    String sanitized_log_value = log_value.replaceAll( regex: "[^A-Za-z0-9]",  replacement: "");

    logger.info("Value to log: {}" + sanitized_log_value);
```



# Logged data:

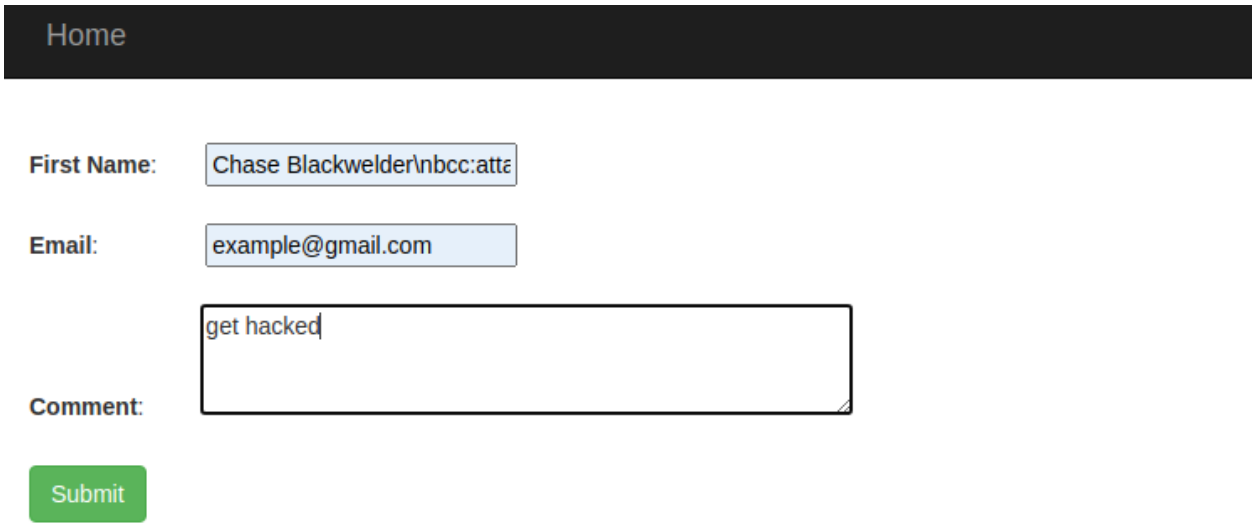
INFO - After exception: wenty-one

INFO: User logged out=badguy
INFO - After exception: mpurba@xyz.com
INFO - After exception: mpurba@xyz.com
INFO - After exception: mpurba@xyz.com
INFO - After exception: mpurba@xyz.com' or 1 = '1
INFO - After exception: mpurba@xyz.com' or 1 = '1
INFO - After exception: twenty-one

INFO - Value to log: {}scriptalert1script
INFO - Value to log: {}f
INFO - Value to log: {}twentyone0a0aINFOUserloggedout3dbadguy
INFO - Value to log: {}scriptalert1script

## 2.5 SMTP

- When looking at the SMTP header page nothing seemed wrong at first but when I attempted to do a header injection it successfully worked and I was able to add a “bcc:” to the email even when no input allowed me to do so



From:Chase Blackwelder
bcc:attackExample@gmail.com
to:example@gmail.com
Message:get hacked

**First Name**: Chase Blackwelder\nbcc:atta

**Email**: example@gmail.com

**Comment**: get hacked

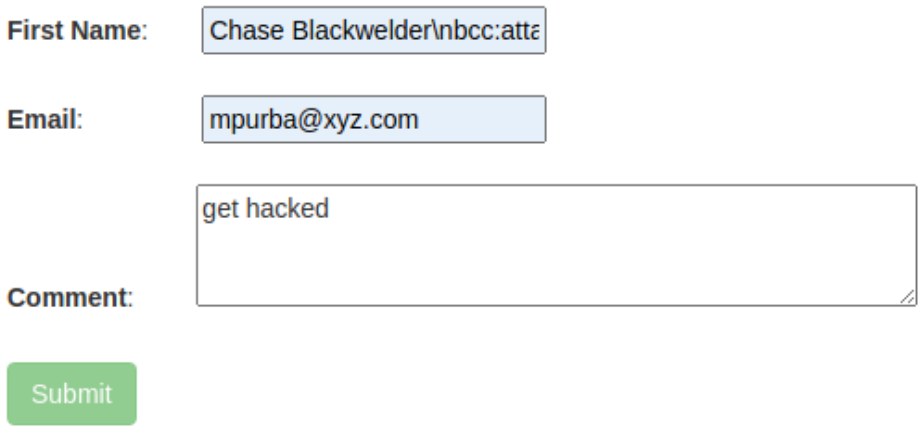
Submit

- The vulnerable part of this is a missing class to further sanitize user inputs to prevent header injection. The code below is the class I added with three methods to prevent this

```
public class SecurityEnhancedAPI {

    no usages
    public String getFilename(String filename) throws FileNotFoundException {
        String regex = "^[\\w,\\s-]+\\.[A-Za-z]{3,10}$";
        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(filename);
        if (matcher.matches()) {
            return filename;
        } else {
            throw new FileNotFoundException("Invalid filename");
        }
    }

    2 usages
    public String getSafeString(String str) throws FileNotFoundException {
        String regex = "^[.\\p{Alnum}\\p{Space}]{0,1024}$";
        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(str);
        if (matcher.matches()) {
            return str;
        } else {
            throw new FileNotFoundException("Invalid safe string");
        }
    }
```

```
public String getEmail(String email) throws FileNotFoundException {
    String regex = "[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}";
    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(email);
    if (matcher.matches()) {
        return email;
    } else {
        throw new FileNotFoundException("Invalid email address");
    }
}
```

- Now when the attack is typed nothing gets returned



## 2.5 XPATH query

- The vulnerability present takes advantage of the lack of data sanitization and allows the user to maliciously manipulate the Xpath and gain access to unauthorized information in the XML file which in this case is a user's id.

# XML data

```
<customers>
  <customer>
    <email>mpurba@xyz.com</email>
    <id>886459</id>
    <password>123</password>
    <name>Moumita Das</name>
    <phone>980-126-5874</phone>
    <amount>30000</amount>
  </customer>
  <customer>
    <email>ashu@xyz.com</email>
    <id>886460</id>
    <password>456</password>
    <name>Ashutosh Dutta</name>
    <phone>980-223-4597</phone>
    <amount>40000</amount>
  </customer>
  <customer>
    <email>ruhani@xyz.com</email>
    <id>886461</id>
    <password>789</password>
    <name>Ruhani Faiheem</name>
    <phone>980-648-3458</phone>
    <amount>50000</amount>
  </customer>
  <customer>
    <email>basel@xyz.com</email>
    <id>886462</id>
    <password>912</password>
    <name>Basel Abdeen</name>
    <phone>980-846-2468</phone>
    <amount>60000</amount>
  </customer>
</customers>
```

# Inject XPath

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

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

[886459, 886460]

**Email**: mpurba@xyz.com' or email =

Submit

- The vulnerability lies in line 59 of the source code as there is no safeguarding of the XPath compilation


- By Making a new class called SimpleVariableResolver I was able to successfully mitigate the vulnerability. This new class makes it so the SimpleVariableResolver class sets the email address as a variable first before it compiles the Xpath expression this was the email address isn't directly in the Xpath expression and as such cannot be modified by an attacker.

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



import javax.xml.namespace.QName;

import javax.xml.xpath.XPathVariableResolver;

import java.util.HashMap;

import java.util.Map;



2 usages
public class SimpleVariableResolver implements XPathVariableResolver {

    2 usages
    private final Map<QName, Object> vars = new HashMap<QName, Object>();

    1 usage
    public void addVariable(QName name, Object value) {

        vars.put(name, value);

    }



    1 usage
    public Object resolveVariable(QName variableName) {

        return vars.get(variableName);

    }

}
```

- The image below is the modifications made to the source code to accommodate the new class

```
SimpleVariableResolver resolver = new SimpleVariableResolver();

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

xpath.setXPathVariableResolver(resolver);

List<String> id_list = new ArrayList<>();
XPathExpression expression = xpath.compile( expression: "/customers/customer[email = $email_val]/id/text()");
```

- As you can see in the image below the attack fails

# XML data


```
<customers>
   <customer>
      <email>mpurba@xyz.com</email>
      <id>886459</id>
      <password>123</password>
      <name>Moumita Das</name>
      <phone>980-126-5874</phone>
      <amount>30000</amount>
   </customer>
   <customer>
      <email>ashu@xyz.com</email>
      <id>886460</id>
      <password>456</password>
      <name>Ashutosh Dutta</name>
      <phone>980-223-4597</phone>
      <amount>40000</amount>
   </customer>
   <customer>
      <email>ruhani@xyz.com</email>
      <id>886461</id>
      <password>789</password>
      <name>Ruhani Faiheem</name>
      <phone>980-648-3458</phone>
      <amount>50000</amount>
   </customer>
   <customer>
      <email>basel@xyz.com</email>
      <id>886462</id>
      <password>912</password>
      <name>Basel Abdeen</name>
      <phone>980-846-2468</phone>
      <amount>60000</amount>
   </customer>
</customers>
```


# Inject XPath

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

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

[]

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

Submit