Assignment Report

EE0052

Name: Thomas Flynn

ID: 16117743

Supervisor: Dr. Sean McGrath

Course: Information & Network Security MEng

Year: 2017

Department: Electronic & Computer Engineering

1 OWASP Vulnerabilities

1.1 A1: Injection

1.1.1 About Injection

Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, particularly in legacy code. They are often found in SQL, LDAP, Xpath, or NoSQL queries; OS commands; XML parsers, SMTP Headers, program arguments, etc. Injection flaws are easy to discover when examining code, but frequently hard to discover via testing. Scanners and fuzzers can help attackers find injection flaws.

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. For SQL calls, this means using bind variables in all prepared statements and stored procedures, and avoiding dynamic queries.**[1]**

1.1.2 Unsafe Example

The following (Java) example is UNSAFE, and would allow an attacker to inject code into the query that would be executed by the database. The unvalidated “customerName” parameter that is simply appended to the query allows an attacker to inject any SQL code they want. Unfortunately, this method for accessing databases is all too common.**[1]**

```
String query = "SELECT account_balance FROM user_data WHERE user_name = "
        + request.getParameter("customerName");
try
{
        Statement statement = connection.createStatement( … );
        ResultSet results = statement.executeQuery( query );
 }
```

1.1.3 Primary Defenses

Defense Option 1: Prepared Statements (with Parameterized Queries)

The use of prepared statements with variable binding (aka parameterized queries) is how all developers should first be taught how to write database queries. They are simple to write, and easier to understand than dynamic queries. Parameterized queries force the developer to first define all the SQL code, and then pass in each parameter to the query later. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied.**[1]**

Safe Java Prepared Statement Example

The following code example uses a PreparedStatement, Java's implementation of a parameterized query, to execute the same database query.**[1]**

```
String custname = request.getParameter("customerName");
String query = "SELECT account_balance FROM user_data WHERE user_name = ? ";
PreparedStatement pstmt = connection.prepareStatement( query );
pstmt.setString( 1, custname);
ResultSet results = pstmt.executeQuery( );
```

Defense Option 2: Stored Procedures

Stored procedures are not always safe from SQL injection. However, certain standard stored procedure programming constructs have the same effect as the use of parameterized queries when implemented safely* which is the norm for most stored procedure languages. They require the developer to just build SQL statements with parameters which are automatically parameterized unless the developer does something largely out of the norm. The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is defined and stored in the database itself, and then called from the application. Both of these techniques have the same effectiveness in preventing SQL injection so your organization should choose which approach makes the most sense for you.**[1]**

Safe Java Stored Procedure Example

The following code example uses a CallableStatement, Java's implementation of the stored procedure interface, to execute the same database query. The "sp_getAccountBalance" stored procedure would have to be predefined in the database and implement the same functionality as the query defined above. **[1]**

```
String custname = request.getParameter("customerName");
try
{
        CallableStatement cs = connection.prepareCall("{call sp_getAccountBalance(?)}");
        cs.setString(1, custname);
        ResultSet results = cs.executeQuery();
        // … result set handling
} catch (SQLException se)
{
        // … logging and error handling
}
```

1.2 A3: Cross Site Scripting (XSS)

1.2.1 About 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.

An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page.**[2]**

1.2.2 XSS Prevention Rules

RULE #0 - Never Insert Untrusted Data Except in Allowed Locations

The first rule is to **deny all** - don't put untrusted data into your HTML document unless it is within one of the slots defined in Rule #1 through Rule #5. The reason for Rule #0 is that there are so many strange contexts within HTML that the list of escaping rules gets very complicated. We can't think of any good reason to put untrusted data in these contexts. This includes "nested contexts" like a URL inside a javascript -- the encoding rules for those locations are tricky and dangerous. If you insist on putting untrusted data into nested contexts, please do a lot of cross-browser testing and let us know what you find out.

Most importantly, never accept actual JavaScript code from an untrusted source and then run it. For example, a parameter named "callback" that contains a JavaScript code snippet. No amount of escaping can fix that.**[2]**

<script>**...NEVER PUT UNTRUSTED DATA HERE...**</script> directly in a script

<!--**...NEVER PUT UNTRUSTED DATA HERE...**--> inside an HTML comment

<div **...NEVER PUT UNTRUSTED DATA HERE...**=test /> in an attribute name

<**NEVER PUT UNTRUSTED DATA HERE...** href="/test" /> in a tag name

<style>**...NEVER PUT UNTRUSTED DATA HERE...**</style> directly in CSS

RULE #1 - HTML Escape Before Inserting Untrusted Data into HTML Element Content

Rule #1 is for when you want to put untrusted data directly into the HTML body somewhere. This includes inside normal tags like div, p, b, td, etc. Most web frameworks have a method for HTML escaping for the characters detailed below. However, this is **absolutely not sufficient for other HTML contexts.** You need to implement the other rules detailed here as well.

<body>**...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...**</body>

...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...

Escape the following characters with HTML entity encoding to prevent switching into any execution context, such as script, style, or event handlers. Using hex entities is recommended in the spec. In addition to the 5 characters significant in XML (&, <, >, ", '), the forward slash is included as it helps to end an HTML entity.**[2]**

& --> &

< --> <

> --> >

" --> "

' --> '

/ --> / forward slash is included as it helps end an HTML entity

RULE #2 - Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes

Rule #2 is for putting untrusted data into typical attribute values like width, name, value, etc. This should not be used for complex attributes like href, src, style, or any of the event handlers like onmouseover. It is extremely important that event handler attributes should follow Rule #3 for HTML JavaScript Data Values.**[2]**

<div attr=**...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...**>content inside UNquoted attribute

content inside single quoted attribute

content inside double quoted attribute

RULE #3 - JavaScript Escape Before Inserting Untrusted Data into JavaScript Data Values

Rule #3 concerns dynamically generated JavaScript code - both script blocks and event-handler attributes. The only safe place to put untrusted data into this code is inside a quoted "data value." Including untrusted data inside any other JavaScript context is quite dangerous, as it is extremely easy to switch into an execution context with characters including (but not limited to) semi-colon, equals, space, plus, and many more, so use with caution.**[2]**

<script>alert('**...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...**')</script>

inside a quoted string

<script>x='**...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...**'</script>

one side of a quoted expression

<div onmouseover="x='**...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...**'" inside quoted event handler

1.2.3 XSS Prevention techniques

Validators

Validation is your chance to verify that the input is as expected. This means that the input should be validated both from a domain view and from a security view. If input represents a string that in your domain should be from 10 to 254 characters long, the same string should be validated also to prevent SQL Injection and XSS attacks. Since many security checks are done using dictionary algorithms it's useful to have a validator that implements those checks and extends it to implement your domain validator. For example: **[2]**

```
public class TenToHundredsValidator extends SecureValidator()
{
    public void validate(…)
    {
        super.validate(...); // Do security checks
        domainvalidate(....);
    }
}
```

Escape output text

<h:outputText/> and <h:outputLabel/>

by default **outputText** has the *escape* attribute set to True. By using this tag to display outputs, you are able to mitigate majority of the XSS vulnerability.**[2]**

1.3 A7: Missing Function Level Access Control

1.3.1 About Missing Function Level Access Control

Web applications typically only show functionality that a user has the need for and rights to use in the UI on screen. For example, if someone works in a hospital, but doesn't have the rights to view health records, then this functionality is not presented to them. But what if a link to see such records was known to them? Or what if the function to show the health records was in the web page code delivered to their browser, but just hidden in the UI. It's possible that a user could spoof the URL to invoke the health records display function if they know how to do that, or they could view the HTML and JavaScript code of a page to see how to call the function. If they do this and they can get access to parts of the application that they shouldn't then this is a case of Missing Function Level Access Controls.**[3]**

1.3.2 Protect against Missing Function Level Access Control vulnerabilities

- It is highly recommended to always apply a deny-by-default rule. Disallow access to all functions in the app by default, then allow access only to those users and other parts of the application that need it.**[3]**
- Even when access to functions within a web application are allowed, each request needs to be verified at the time of the access. Check that the requests are from valid authorised users.**[3]**
- Use access control lists and role based authentication to enforce the above. Use the principle of least privilege. To emphasise – give access to functions only if required. Don't try and give general access and then remove access rights from users who shouldn't have it.**[3]**
- Don't try to rely on security by obscurity by just hiding buttons and links to functionality within the UI. Someone will stumble upon a way to access the hidden functions. Plus, people who are determined to attack your web application will ignore the UI and send requests directly in order to try and get responses from the backend application and database engines.**[3]**
- Check every URL, button, and other way to access functions in your web application using an account with low privileges. See if these accounts can get access to functions they shouldn't. There are tools that can help with this task. They compare scans of your web applications when authenticated as an Admin user, with scans when authenticated as standard users. They then highlight parts of the application that are available to standard users when they shouldn't be.**[3]**
- Assume attackers **will** learn where "hidden" directories and "random" filenames are, so do not store these files in the web root, even if they are not directly linked.**[3]**

1.4 A8: Cross Site Request Forgery (CSRF)

1.4.1 About CSRF

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. The impact of a successful CSRF attack is limited to the capabilities exposed by the vulnerable application. For example, this attack could result in a transfer of funds, changing a password, or purchasing an item in the user's context. In effect, CSRF attacks are used by an attacker to make a target system perform a function via the target's browser without knowledge of the target user, at least until the unauthorized transaction has been committed.

Impacts of successful CSRF exploits vary greatly based on the privileges of each victim. When targeting a normal user, a successful CSRF attack can compromise end-user data and their associated functions. If the targeted end user is an administrator account, a CSRF attack can compromise the entire web application. 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). Utilizing social engineering, an attacker can embed malicious HTML or JavaScript code into an email or website to request a specific 'task URL'. The task then executes with or without the user's knowledge, either directly or by utilizing a Cross-Site Scripting flaw (ex: Samy MySpace Worm).**[4]**

1.4.2 CSRF Example Attack Scenario

The application allows a user to submit a state changing request that does not include anything secret. For example:

> **http://example.com/app/transferFunds?amount=1500&destinationAccount=4673243243**

So, the attacker constructs a request that will transfer money from the victim's account to the attacker's account, and then embeds this attack in an image request or iframe stored on various sites under the attacker's control:

> ****

If the victim visits any of the attacker's sites while already authenticated to example.com, these forged requests will automatically include the user's session info, authorizing the attacker's request.**[5]**

1.4.3 CSRF Prevention

Identify Source Origin

To identify the source origin, it is recommended to use one of these two standard headers that almost all requests include one or both of:

- Origin Header
- Referer Header

Checking the Origin Header

If the Origin header is present, verify its value matches the target origin. The Origin HTTP Header standard was introduced as a method of defending against CSRF and other Cross-Domain attacks. Unlike the Referer, the Origin header will be present in HTTP requests that originate from an HTTPS URL. If the Origin header is present, then it should be checked to make sure it matches the target origin.

This defense technique is specifically proposed in section 5.0 of Robust Defenses for Cross-Site Request Forgery. This paper proposes the creation of the Origin header and its use as a CSRF defense mechanism.**[5]**

Identifying the Target Origin

If you are behind a proxy, there are a number of options to consider:

1. Configure your application to simply know its target origin
2. Use the Host header value
3. Use the X-Forwarded-Host header value

Its your application, so clearly you can figure out its target origin and set that value in some server configuration entry. This would be the most secure approach as its defined server side so is a trusted value. However, this can be problematic to maintain if your application is deployed in many different places, e.g., dev, test, QA, production, and possibly multiple production instances. Setting the correct value for each of these situations can be difficult, but if you can do it, that's great.**[5]**

Verifying the Two Origins Match

Once you've identified the source origin (from either the Origin or Referer header), and you've determined the target origin, however you choose to do so, then you can simply compare the two values and if they don't match you know you have a cross-origin request.**[5]**

2 Applied Prevention Techniques

The primary technique for defending against Injection is the use of a prepared statement for executing sql queries.

2.1 A1 Injection

2.1.1 LoginDAO Class

This class is used to validate a username and password. Below is example code where prepared statements have been used in the class in order to prevent injection exploit.

```
public static int validate(String user, String password)
{
    Connection con = null;
    PreparedStatement ps = null;

    try {
            con = DataConnect.getConnection();//obtain connection to database

            //prepared statement to defend against A1: Injection
            ps = con.prepareStatement("SELECT USERNAME, PW, ISADMIN FROM APP.USERS WHERE USERNAME = ? AND PW = ?");
            ps.setString(1, user);//prepared statement to defend against A1: Injection
            ps.setString(2, password);//prepared statement to defend against A1: Injection

            ResultSet rs = ps.executeQuery();//store values from sql query in rs
```

```
public static void changeProfile(Login user)
{
    Connection con = null;
    PreparedStatement ps = null;

    try
    {
        con = DataConnect.getConnection();

        ps = con.prepareStatement("UPDATE APP.USERS SET MSG = ? WHERE USERNAME = ? AND PW = ?");

        ps.setString(1, user.getMsg());
        ps.setString(2, user.getUser());
        ps.setString(3, user.getPwd());

        ps.execute();
```

2.1.2 Product Bean Class

This class is used by both the customer and the admin when interacting with products. Below is example code where prepared statements have been used in the class in order to prevent injection exploit.

```
public String addNewProduct(Login user){

    Connection con = null;
    PreparedStatement ps = null;

    if(user.getIsAdmin() ==true)// protect against A7 Missing Function Level Access Control
    {
        try{
            List<Product> productList = getProductList();

            for (Product p: productList)
            {
                con = DataConnect.getConnection();
                con.setAutoCommit(true);
                //A1: Injection prevention using prepared statement
                ps = con.prepareStatement("INSERT INTO APP.PRODUCTS (ID, PRODUCT, QUANTITY, PRICE) VALUES (?,?,?,?)");
```

```
public String removeProduct(Product removed, Login user)
{
    Connection con = null;
    PreparedStatement ps = null;

    if(user.getIsAdmin() ==true)// protect against A7 Missing Function Level Access Control
    {
        try
        {
            con = DataConnect.getConnection();
            con.setAutoCommit(true);
            //A1: Injection prevention using prepared statement
            ps = con.prepareStatement("DELETE FROM APP.PRODUCTS WHERE ID =?");

            int id = removed.getId();
            ps.setInt(1, id);
            ps.execute();
            con.commit();

            logger.info("Product : " + removed.getProduct() + " removed from inventory");
```

```
public String searchByID() throws SQLException
{

    if(idParam.equalsIgnoreCase("-1") == false)
    {
        if(ds==null)
                throw new SQLException("Can't get data source");

        //get database connection
        Connection con = dc.getConnection();

        if(con==null)
                throw new SQLException("Can't get database connection");

        PreparedStatement ps
                = con.prepareStatement(
                  "select * from app.products where id = ?");

        ps.setInt(1, Integer.parseInt(idParam));
```

```
public String searchByName() throws SQLException
{
    System.out.println("TESTING  searchByName() ");

    if(idParam.equalsIgnoreCase("-1") == false)
    {
        if(ds==null)
                throw new SQLException("Can't get data source");

        //get database connection
        Connection con = dc.getConnection();

        if(con==null)
                throw new SQLException("Can't get database connection");

        //A1: Injection prevention using prepared statement
        PreparedStatement ps
                = con.prepareStatement(
                  "select * from app.products where product = ?");

        //get customer data from database
        //ps.setInt(1, Integer.parseInt(idParam));
        ps.setString(1, nameParam);

        ResultSet result =  ps.executeQuery();
```

```
public String updateQuantity(Login user)
{

    Connection con = null;
    PreparedStatement ps = null;

    if(user.getIsAdmin() ==true)// protect against A7 Missing Function Level Access Control
    {
        try
        {
            List<Product> productList = getProductList();

            for (Product p: productList)
            {
                con = DataConnect.getConnection();
                con.setAutoCommit(true);
                //A1: Injection prevention using prepared statement
                ps = con.prepareStatement("UPDATE APP.PRODUCTS SET QUANTITY = ? WHERE ID = ?");
```

2.1.3 ShoppingCart class

This class is used by the customer to add products from the shop to their shopping cart. Below is example code where prepared statements have been used in the class in order to prevent injection exploit.

```
public String checkout()
{
    Connection con = null;
    PreparedStatement ps = null;

    try
    {
        con = DataConnect.getConnection();
        con.setAutoCommit(false);

        for(Item item : cart)
        {
            //A1: Injection prevention using prepared statement
            ps = con.prepareStatement("select * from app.products where ID = ?");
            ps.setInt(1, item.getP().getId());
            ResultSet result =  ps.executeQuery();
            int inStoreQuantity = -1;

            System.out.println("item.getP().getId() = " + item.getP().getId());
            System.out.println("item.getQuantity() = " + item.getQuantity());
            System.out.println("inStoreQuantity = " + inStoreQuantity);

            if(result.next())
                inStoreQuantity =  result.getInt("QUANTITY");

            System.out.println("inStoreQuantity = " + inStoreQuantity);

            if(inStoreQuantity >= item.getQuantity())
            {
                try
                {
                    return "checkout";
```

```
public String confirmOrder(String customer){
    logger.info("User : confirmed order ");

    Connection con = null;
    PreparedStatement ps = null;

    try
    {
        con = DataConnect.getConnection();
        con.setAutoCommit(true);
        //A1: Injection prevention using prepared statement
        ps = con.prepareStatement("INSERT INTO APP.ORDERS (CUSTOMER, COST, ORDERID) VALUES (?,?,?)");

        ps.setString(1, customer);
        double totalPrice = getTotal();
        ps.setDouble(2, totalPrice);
        int orderID = ThreadLocalRandom.current().nextInt(1, 10000 + 1);
        ps.setInt(3, orderID);
        ps.execute();
        con.commit();
```

2.2 A3 XSS

Below is example code where implicit escape is used in order to prevent potential XSS exploits.

2.2.1 shop.xhtml

```
<h1>Shop</h1>

<h:form>
        <p>You are logged in as shop user #{login.user}</p>  <!--Implicit escape-->
        <h:commandLink action="#{login.logout}" value="logout"></h:commandLink>
</h:form>
```

```
<h:column>
        <f:facet name="header">
            Product ID
        </f:facet>
        <p>  #{p.id} </p><!--Implicit escape-->

</h:column>

<h:column>
        <f:facet name="header">
            product
        </f:facet>
        <p>  #{p.product} </p><!--Implicit escape-->

</h:column>

        <h:column>
        <f:facet name="header">
            quantity
        </f:facet>
        <p>  #{p.quantity} </p><!--Implicit escape-->

</h:column>

<h:column>
        <f:facet name="header">
            price
        </f:facet>
        <p>   #{p.price} </p><!--Implicit escape-->

</h:column>
```

2.2.2 admin.xhtml

```
<h:form>
        <p>You are logged in as admin #{login.user}</p><!--Implicit escape-->
        <h:commandLink action="#{login.logout}" value="logout"></h:commandLink>
</h:form>

<h:form>
        <h:dataTable value="#{product.getProductList()}" var="p"
            styleClass="order-table"
            headerClass="order-table-header"
            rowClasses="order-table-odd-row,order-table-even-row"
            >

        <h:column>
                <f:facet name="header">
                        Product ID
                </f:facet>
             <p> #{p.id} </p><!--Implicit escape-->

        </h:column>

        <h:column>
                <f:facet name="header">
                        product
                </f:facet>
                <p>  #{p.product} </p><!--Implicit escape-->

        </h:column>

                <h:column>
                <f:facet name="header">
                        quantity
                </f:facet>
                <p>   #{p.quantity} </p><!--Implicit escape-->

        </h:column>
```

2.2.3 profile.xhtml

```
<h:dataTable value="#{profileBean.getProfileList()}" var="p"
        styleClass="order-table"
        headerClass="order-table-header"
        rowClasses="order-table-odd-row,order-table-even-row"
        >

    <h:column>
        <f:facet name="header">
            Profile ID
        </f:facet>
        <p>   #{p.id}</p><!--Implicit escape-->
    </h:column>

    <h:column>
        <f:facet name="header">
            Username
        </f:facet>
        <p> #{p.name} </p><!--Implicit escape-->
    </h:column>

    <h:column>
        <f:facet name="header">
                Message
        </f:facet>
        <p>  #{p.msg} </p><!--Implicit escape-->
    </h:column>

</h:dataTable>

 <h:dataTable value="#{profileBean.result}" var="p"
        styleClass="order-table"
        headerClass="order-table-header"
        rowClasses="order-table-odd-row,order-table-even-row"
        >

    <h:column>
        <f:facet name="header">
                Profile ID
        </f:facet>
        <p>   #{p.id}</p><!--Implicit escape-->
    </h:column>

        <h:column>
        <f:facet name="header">
                Username
        </f:facet>
        <p>  #{p.name}</p><!--Implicit escape-->
    </h:column>

    <h:column>
        <f:facet name="header">
                Message
                </f:facet>
        <p> #{p.msg}</p><!--Implicit escape-->
    </h:column>

 </h:dataTable>
```

2.2.4 XSS Filter class

```
@WebFilter(filterName = "XssFilter", urlPatterns = { "*.xhtml" })
public class XSSFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }

    @Override
    public void destroy() {
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        chain.doFilter(new XSSRequestWrapper((HttpServletRequest) request), response);
    }
```

2.2.4 XSSRequestWrapper class

This class strips out the most common XSS attacks by stripping them out of the htpp header.

```
public class XSSRequestWrapper extends HttpServletRequestWrapper {

    public XSSRequestWrapper(HttpServletRequest servletRequest) {
        super(servletRequest);
    }

    @Override
    public String[] getParameterValues(String parameter) {
        String[] values = super.getParameterValues(parameter);

        if (values == null) {
            return null;
        }

        int count = values.length;
        String[] encodedValues = new String[count];
        for (int i = 0; i < count; i++) {
            encodedValues[i] = stripXSS(values[i]);
        }

        return encodedValues;
    }

    @Override
    public String getParameter(String parameter) {
        String value = super.getParameter(parameter);

        return stripXSS(value);
    }

    @Override
    public String getHeader(String name) {
        String value = super.getHeader(name);
        return stripXSS(value);
    }
```

```
private String stripXSS(String value) {
    if (value != null) {
        System.out.println("StripXSS() called\n Header = " + value);
        // NOTE: It's highly recommended to use the ESAPI library and uncomment the following line to
        // avoid encoded attacks.
        // value = ESAPI.encoder().canonicalize(value);

        // Avoid null characters
        value = value.replaceAll("", "");

        // Avoid anything between script tags
        Pattern scriptPattern = Pattern.compile("<script>(.*?)</script>", Pattern.CASE_INSENSITIVE);
        value = scriptPattern.matcher(value).replaceAll("");

        // Avoid anything in a src='...' type of expression
        scriptPattern = Pattern.compile("src[\r\n]*=[\r\n]*\\\'(.*?)\\\'", Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
        value = scriptPattern.matcher(value).replaceAll("");

        scriptPattern = Pattern.compile("src[\r\n]*=[\r\n]*\\\"(.*?)\\\"", Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
        value = scriptPattern.matcher(value).replaceAll("");

        // Remove any lonesome </script> tag
        scriptPattern = Pattern.compile("</script>", Pattern.CASE_INSENSITIVE);
        value = scriptPattern.matcher(value).replaceAll("");

        // Remove any lonesome <script ...> tag
        scriptPattern = Pattern.compile("<script(.*?)>", Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
        value = scriptPattern.matcher(value).replaceAll("");

        // Avoid eval(...) expressions
        scriptPattern = Pattern.compile("eval\\((.*?)\\)", Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
        value = scriptPattern.matcher(value).replaceAll("");

        // Avoid expression(...) expressions
        scriptPattern = Pattern.compile("expression\\((.*?)\\)", Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
        value = scriptPattern.matcher(value).replaceAll("");

        // Avoid javascript:... expressions
        scriptPattern = Pattern.compile("javascript:", Pattern.CASE_INSENSITIVE);
        value = scriptPattern.matcher(value).replaceAll("");

        // Avoid vbscript:... expressions
        scriptPattern = Pattern.compile("vbscript:", Pattern.CASE_INSENSITIVE);
        value = scriptPattern.matcher(value).replaceAll("");
```

2.3 A7: Missing Function Level Access Control

Below is example code where consideration of Missing function level access control was considered. The prevention technique uses the Login session bean to determine if the user has admin privileges or not. When the user logs in, the login bean's field "isAdmin" is set to true only if it the corresponding field "ISADMIN" in the database is true. Only when this field is true, can the user access privileged functions such as "addNewProduct" and "removeProduct".

```
public String addNewProduct(Login user){

    Connection con = null;
    PreparedStatement ps = null;

    if(user.getIsAdmin() ==true)// protect against A7 Missing Function Level Access Control
    {
        try{
            List<Product> productList = getProductList();

            for (Product p: productList)
            {
                con = DataConnect.getConnection();
                con.setAutoCommit(true);
                //A1: Injection prevention using prepared statement
                ps = con.prepareStatement("INSERT INTO APP.PRODUCTS (ID, PRODUCT, QUANTITY, PRICE) VALUES (?,?,?,?)");
```

```
public String removeProduct(Product removed, Login user)
{
    Connection con = null;
    PreparedStatement ps = null;

    if(user.getIsAdmin() ==true)// protect against A7 Missing Function Level Access Control
    {
        try
        {
            con = DataConnect.getConnection();
            con.setAutoCommit(true);
            //A1: Injection prevention using prepared statement
            ps = con.prepareStatement("DELETE FROM APP.PRODUCTS WHERE ID =?");

            int id = removed.getId();
            ps.setInt(1, id);
            ps.execute();
            con.commit();

            logger.info("Product : " + removed.getProduct() + " removed from inventory");
```

```
public String updateQuantity(Login user)
{

    Connection con = null;
    PreparedStatement ps = null;

    if(user.getIsAdmin() ==true)// protect against A7 Missing Function Level Access Control
    {
        try
        {
            List<Product> productList = getProductList();

            for (Product p: productList)
            {
                con = DataConnect.getConnection();
                con.setAutoCommit(true);
                //A1: Injection prevention using prepared statement
                ps = con.prepareStatement("UPDATE APP.PRODUCTS SET QUANTITY = ? WHERE ID = ?");
```

2.4 A8: Cross Site Request Forgery (CSRF)

JSF 2.x has already builtin CSRF prevention in flavor of `javax.faces.ViewState` hidden field in the form when using server side state saving. In JSF 1.x this value was namely pretty weak and too easy predictable. In JSF 2.0 this has been improved by using a long and strong autogenerated value instead of a rather predictable sequence value and thus making it a robust CSRF prevention.

Only when you're using stateless views as in `<f:view transient="true">`, or there's somewhere a XSS attack hole in the application, then you've a potential CSRF attack hole.

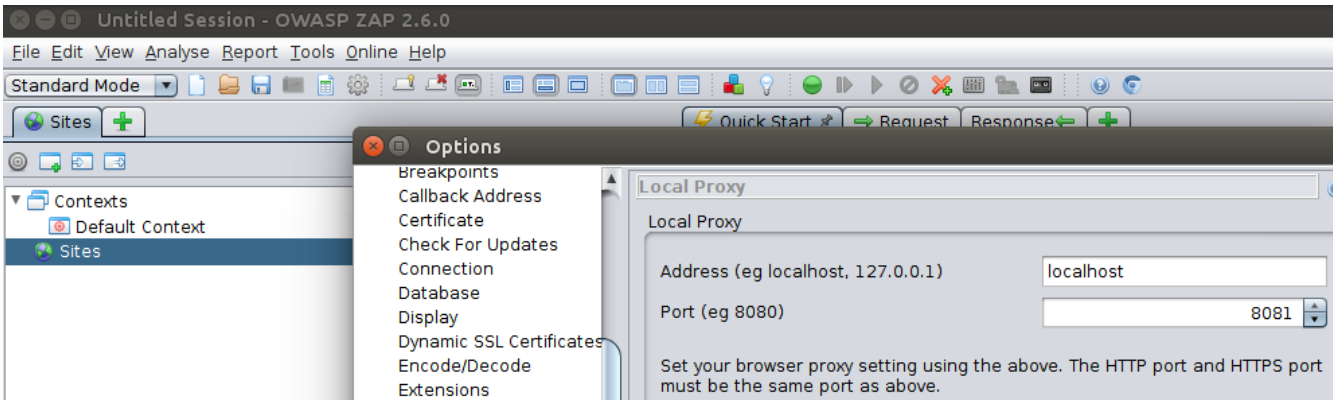
2.4.1 AuthorizationFilter class

In the doFilter method the filter will redirect user to login page if he/she tries to access another page without logging in first.

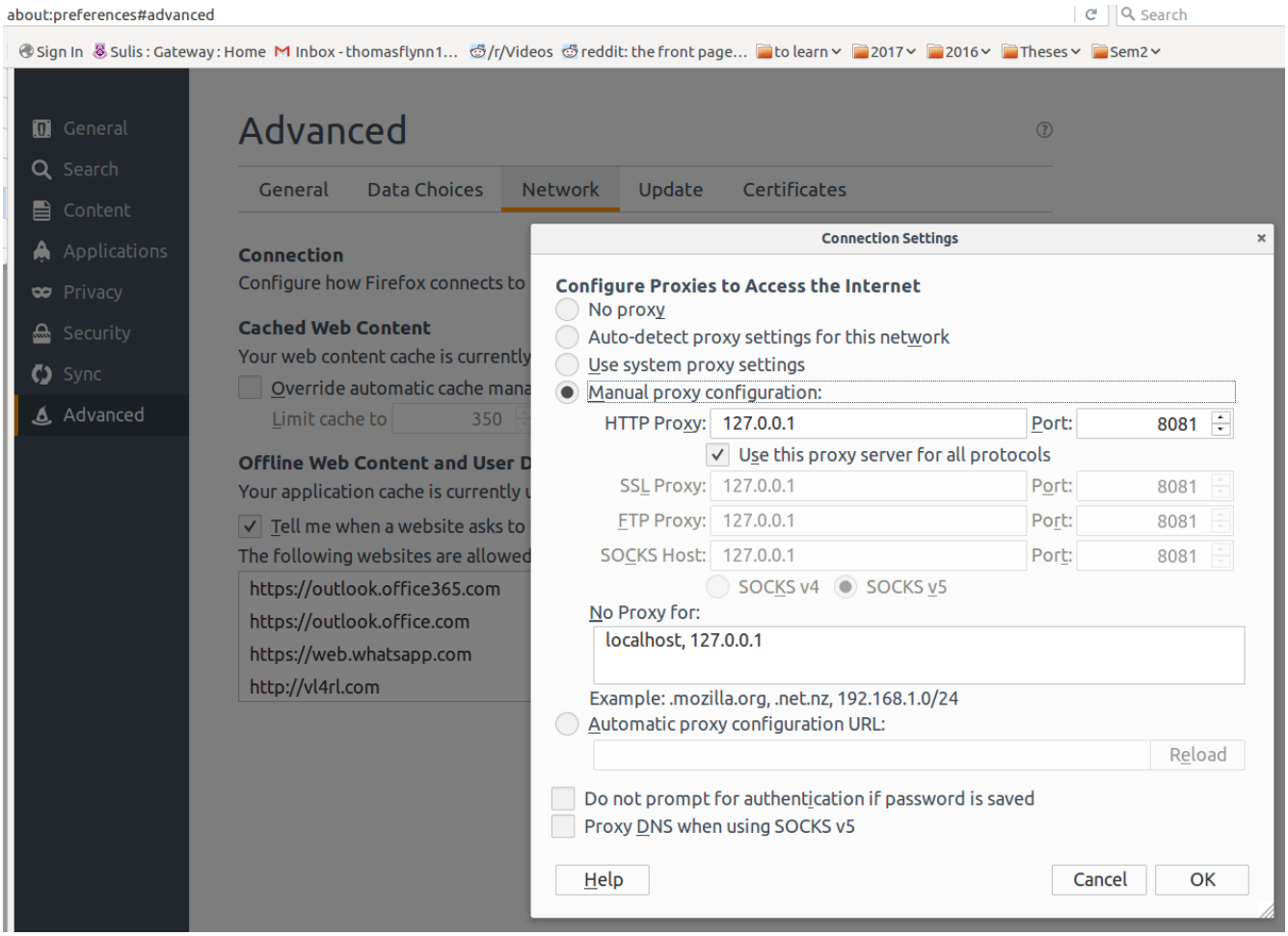
```
@WebFilter(filterName = "AuthFilter", urlPatterns = { "*.xhtml" })
public class AuthorizationFilter implements Filter {

	public AuthorizationFilter() {
	}

	@Override
	public void init(FilterConfig filterConfig) throws ServletException {

	}

	@Override
	public void doFilter(ServletRequest request, ServletResponse response,
			FilterChain chain) throws IOException, ServletException {
		try {
			HttpServletRequest reqt = (HttpServletRequest) request; //cast request from ServletRequest to HttpServletRequest
			HttpServletResponse resp = (HttpServletResponse) response; //cast response from ServletResponse to HttpServletResponse
			HttpSession ses = reqt.getSession(false);//set session as false

			String reqURI = reqt.getRequestURI();

			System.out.println("Request URI :" + reqURI);

			//if any of the below statements are true, perform chain.doFilter
			if (reqURI.indexOf("/login.xhtml") >= 0
					|| (ses != null && ses.getAttribute("username") != null)
					|| reqURI.indexOf("/public/") >= 0
					|| reqURI.contains("javax.faces.resource"))
				chain.doFilter(request, response); //send request and response to authorization and xss filter
			else
				resp.sendRedirect(reqt.getContextPath() + "/faces/login.xhtml");//redirect user to login page
		} catch (Exception e) {
			System.out.println(e.getMessage());
		}
	}
```

3 Testing

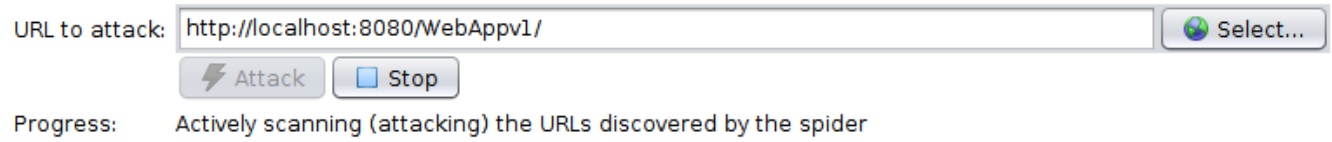
3.1 ZAP Proxy Configuration

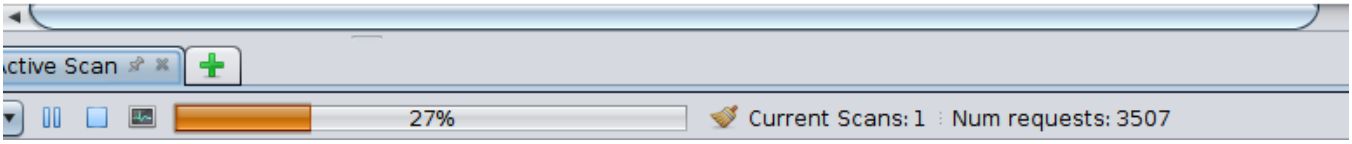


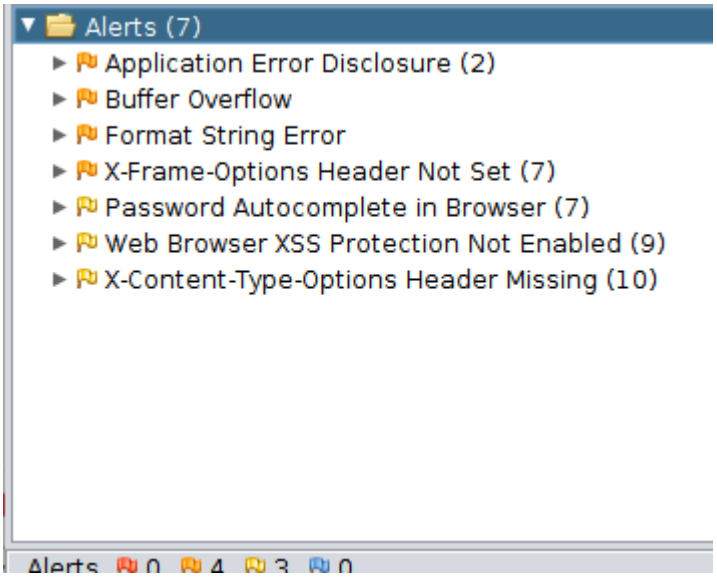
3.2 Mozilla Proxy Configuration



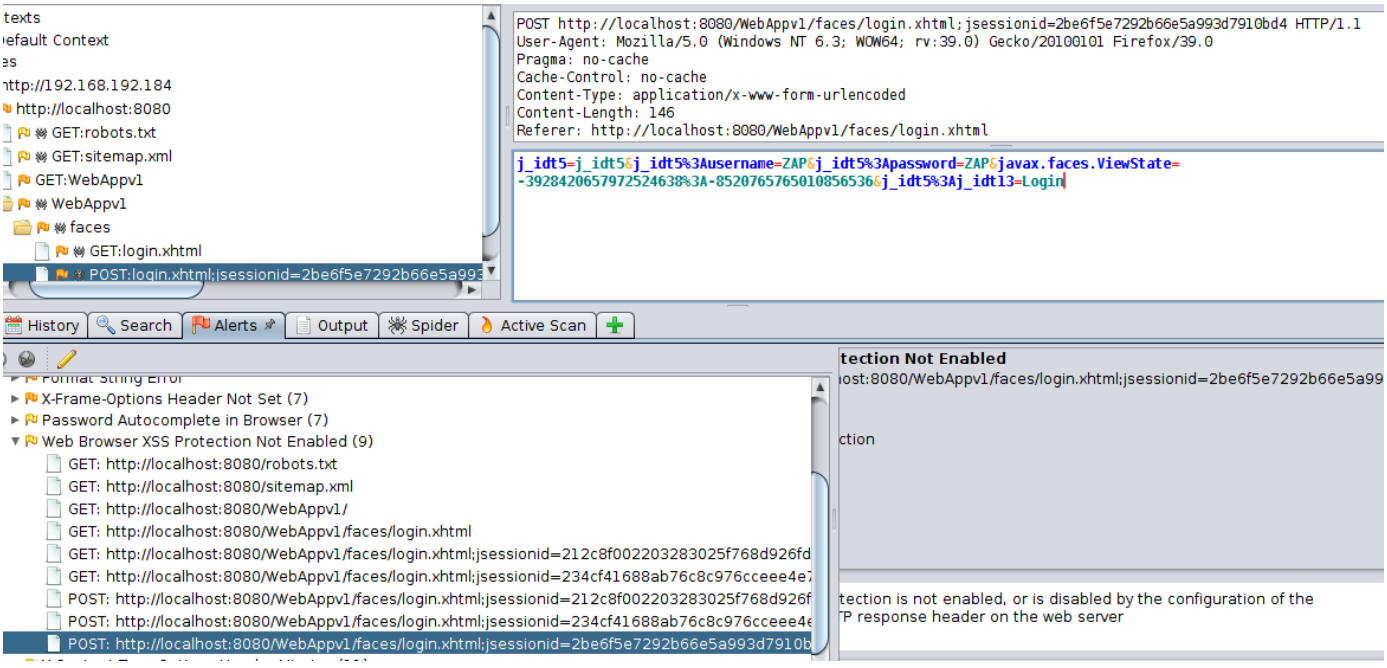
3.3 ZAP Scan



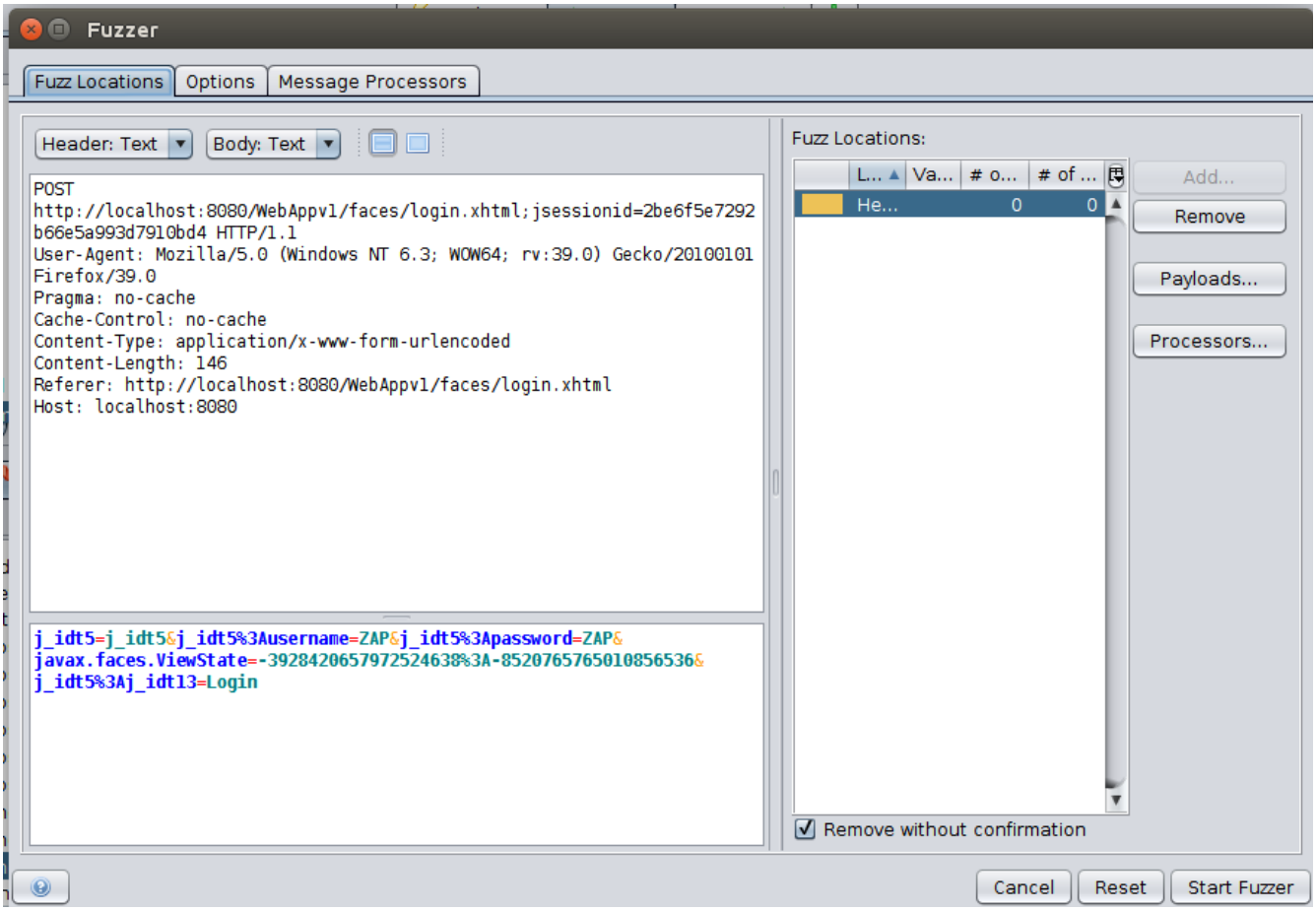




The scan did not find any exploits for A1, A7 and A8 exploits. It did however find some low risk XSS vulnerabilities however.



Using the fuzzer to tool to further examine the vulnerability.

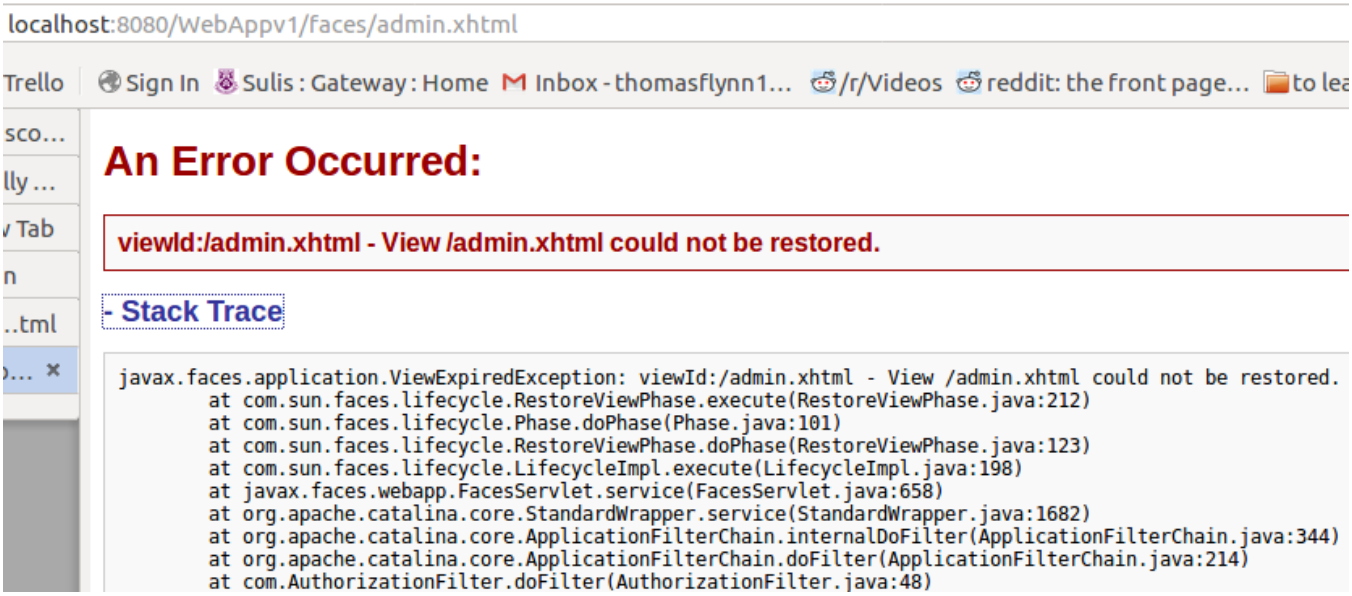


The vulnerability seems to be related to the authenticate user method. The builtin CSRF prevention using `javax.faces.ViewState` hidden field can be seen here. In JSF 1.x this value was namely pretty weak and too easy predictable. In JSF 2.0 this has been improved by using a long and strong autogenerated value instead of a rather predictable sequence value and thus making it a robust CSRF prevention.

Further XSS pen testing is required before deploying the web application to the public.

3.4 Customer tries to access admin page test

The test consisted of logging in as a customer and trying to access the admin page. This can be seen as a form of A7 prevention. Also if the attacker somehow gained access to the admin page, they would not be able to view, add, remove or update quantity of items as those functions require the “isAdmin” field of the login bean to be set to true.

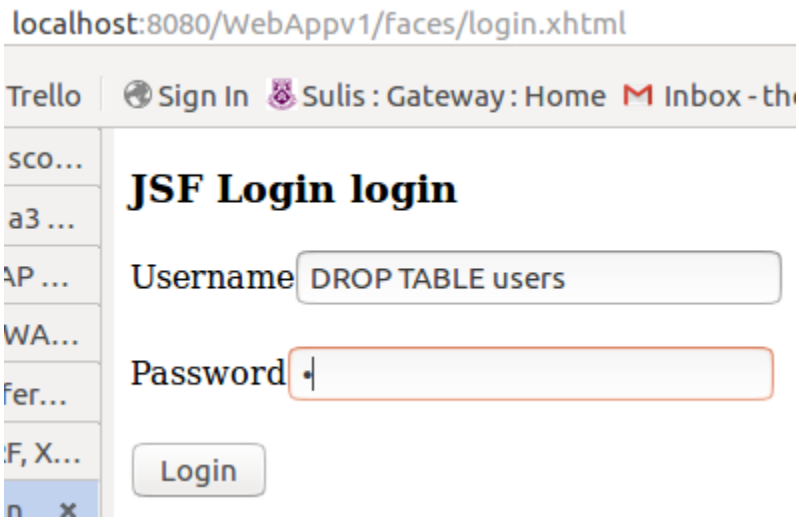


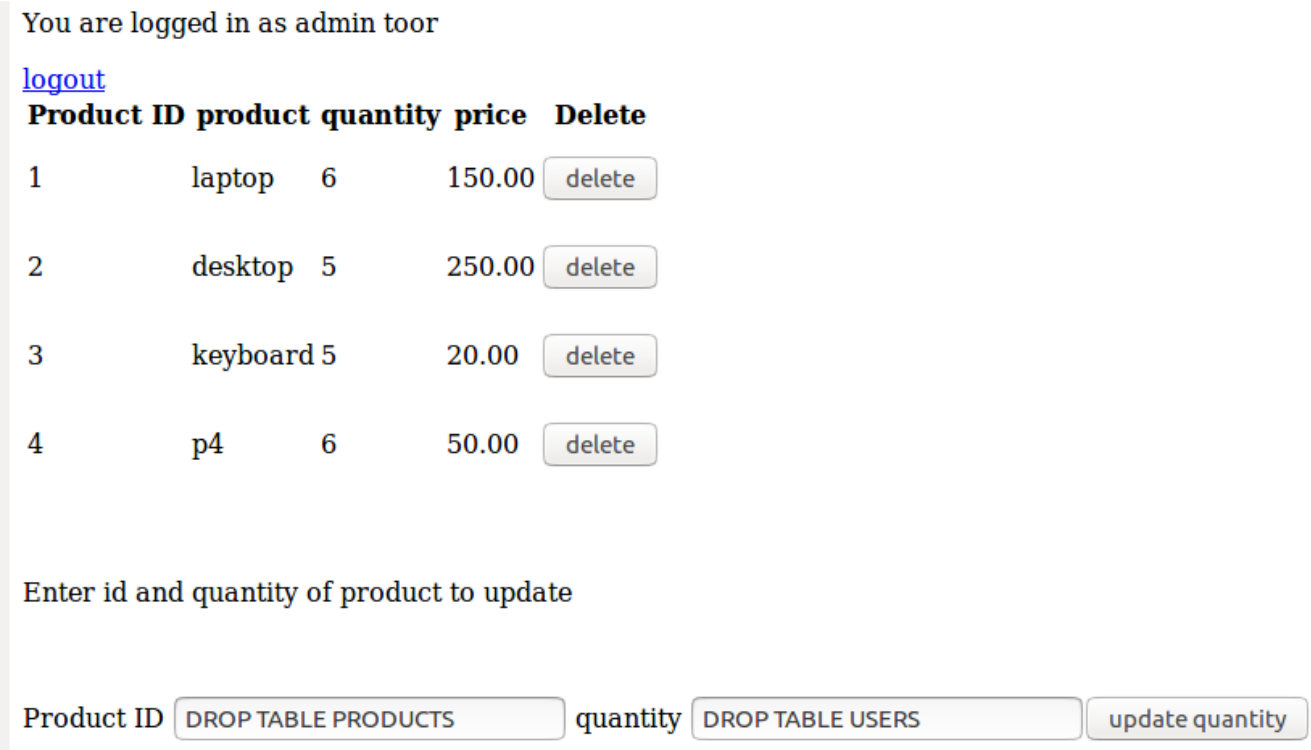
```
Info:   Request URI :/WebAppv1/faces/login.xhtml
Info:   is admin = true
Info:   username = toor
Info:   pwd = 4uIdo0!
Info:   Request URI :/WebAppv1/faces/login.xhtml
Info:   is admin = false
Info:   username = bob
Info:   pwd = 123
Info:   Request URI :/WebAppv1/faces/shop.xhtml
Info:   Request URI :/WebAppv1/faces/login.xhtml
Info:   is admin = false
Info:   username = bob
Info:   pwd = 123
Info:   Request URI :/WebAppv1/faces/admin.xhtml
FATAL:   JSF1073: javax.faces.application.ViewExpiredException caught during processing of RESTORE_VIEW 1
FATAL:   viewId:/admin.xhtml - View /admin.xhtml could not be restored.
javax.faces.application.ViewExpiredException: viewId:/admin.xhtml - View /admin.xhtml could not be restore
        at com.sun.faces.lifecycle.RestoreViewPhase.execute(RestoreViewPhase.java:212)
        at com.sun.faces.lifecycle.Phase.doPhase(Phase.java:101)
        at com.sun.faces.lifecycle.RestoreViewPhase.doPhase(RestoreViewPhase.java:123)
        at com.sun.faces.lifecycle.LifecycleImpl.execute(LifecycleImpl.java:198)
        at javax.faces.webapp.FacesServlet.service(FacesServlet.java:658)
        at org.apache.catalina.core.StandardWrapper.service(StandardWrapper.java:1682)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:34
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:214)
        at com.AuthorizationFilter.doFilter(AuthorizationFilter.java:48)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:25
```

The above glassfish server stacktrace demonstrates how the Authorizaion filter prevented the customer from accessing the admin page.

3.5 A1: Injection Testing

To test the prepared statements used through the application, the statement “DROP TABLE” was injected into various web fields that would be used as parameters in sql statements.





Both of these injection attacks failed to drop the PRODUCTS and USERS table due to the use of “prepared statements”.

3.6 A8 Testing

In order to create the scenario where an unauthorized user had gained access to the admin page, the isAdmin field was set to false for both the customer and the admin. This was done by commenting 1 line of code as seen below.

```
//validate login
public String validateUsernamePassword()
{
    try
    {
        int valid = LoginDAO.validate(user, pwd);
        if (valid == 2) {
            HttpSession session = SessionUtils.getSession();
            session.setAttribute("username", user);
            //isAdmin = true;
            return "admin";
        }
        else if (valid == 1) {
            HttpSession session = SessionUtils.getSession();
            session.setAttribute("username", user);
            return "shop";
        }
```

Now the scenario where an unauthorized user is looking at the admin page has been created. The attacker tries to use the “update quantity” button. Since the “isAdmin” field is set to false, the user us notified that their attack has been prevented.

4 Deployment

4.1 About

5 Bibliography

[1]"Top 10 2013-A1-Injection - OWASP", *Owasp.org*, 2017. [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-A1-Injection. [Accessed: 25- Mar- 2017].

[2]"Top 10 2013-A7-Missing Function Level Access Control - OWASP", *Owasp.org*, 2017. [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-A7-Missing_Function_Level_Access_Control. [Accessed: 25- Mar- 2017].

[3]"OWASP Top Ten Series: Missing Function Level Access Control - Load Balancers", *Load Balancers*, 2017. [Online]. Available: https://kemptechnologies.com/blog/owasp-top-ten-series-missing-function-level-access-control/. [Accessed: 25- Mar- 2017].

[4]"Top 10 2013-A8-Cross-Site Request Forgery (CSRF) - OWASP", *Owasp.org*, 2017. [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-A8-Cross-Site_Request_Forgery_(CSRF). [Accessed: 25- Mar- 2017].

[5]"Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet - OWASP", *Owasp.org*, 2017. [Online]. Available: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet. [Accessed: 25- Mar- 2017].

https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

https://www.owasp.org/index.php/Top_10_2013-A1-Injection

https://www.owasp.org/index.php/Top_10_2013-A7-Missing_Function_Level_Access_Control

https://kemptechnologies.com/blog/owasp-top-ten-series-missing-function-level-access-control/

https://www.owasp.org/index.php/Top_10_2013-A8-Cross-Site_Request_Forgery_(CSRF

https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet

link to cart code

[7]"How to create shopping cart using JSF", *YouTube*, 2017. [Online]. Available: https://www.youtube.com/watch?v=IH5TeMObs2s. [Accessed: 25- Mar- 2017].