Identifying vulnerabilities

Group: Assignment 3 3
Crew : Felix Esposito Duran, Jakub Volak, Javier Mateos Manzano, Niki Florian Hendel, Venthan Vigneswaran

Vulnerability #1: WSTG-ATHN-07: Testing for Weak Password Policy

Description:
I found there is no validation for user password. User can use very weak and easy passwords such as 123456. Also in code is used weak hashing function MD5, which is considered to be broken.

Possible consequences:
If there is leak of passwords, attacker can easily crack these passwords because of weak hashing function. Also attacker can try to guess passwords because users can use easy passwords. An attacker can penetrate the system and act as another user.

File(s):
Crypto.java, AppUserDAO.java, NewUserServlet.java, UpdatePasswordServlet.java

Code:
```
public static String generateMD5Hash(String value) {

        return DigestUtils.md5Hex(value);

}
```
Validation code is missing.

Payload:
Username: Test
Password: 123456 || aaaaa

Technique:
I manually tried to create new user with weak and easy passwords. Also I looked into the code, how are passwords stored and I discovered there is used weak hashing function.

Vulnerability #2: WSTG-INPV-05: Testing for SQL Injection

Description:
The Login has a SQL Injection Flaw

Possible consequences:
Get Access to all accounts

File (s):
AppUserDAO.java

Code:
the Function “ getAuthenticatedUser “

Payload:
= 1' or '1' = '1'--

Analysis Technique:
I looked at the code an debug it

Vulnerability #3: WSTG-INPV-01: Testing for Reflected Cross Site Scripting

Description:
An attacker can send a link which has scripts as a value for the parameters.

Possible consequences:
Stealing user's sensitive data, using his/her computer for mining cryptocurrencies, using all his memory...

File (s):
Validator.java

Code:

```
public static String validString(String parameter) {

        return parameter != null ? parameter : "null";

}
```

Payload:
http://localhost:8080/DAT152WebSearchOblig3/dosearch?searchkey=%3Cscript%3Ealert%281%29%3C%2Fscript%3E

Analysis Technique:
Manual. The search key was suspicious to be the place to put some scripting.

Vulnerability #4: WSTG-INPV-02: Testing for Stored Cross Site Scripting

Description:
An attacker can write scripts in input fields that will be stored in a database and loaded in further accesses to the webpage.

Possible consequences:
Same as in Vulnerability #3 but potentially affecting even more devices/users.

File (s):
Same as in Vulnerability #3

Code:
Same as in Vulnerability #3

Payload:
http://localhost:8080/DAT152WebSearchOblig3/dosearch?
searchkey=%3Cscript%3Ealert%281%29%3C%2Fscript%3E

Analysis Technique:
Manual. The search key was suspicious to be the place to put some scripting.

Vulnerability #5: WSTG-SESS-05: Testing for Cross Site Request Forgery

Description:
An attacker can send us a link that submits a form changing the role of a user.

Possible consequences:
Getting admin privileges and then deleting data, change role of the administrators, information hijacking…

File (s):
UpdateRoleServlet.java, UpdatePasswordServlet.java

Code:
UpdateRoleServlet.java:
doPost method does not have any measures to avoid CSRF
UpdatePasswordServlet.java:
doPost method does not have any measures to avoid CSRF

Payload:
```
<html>
<head></head>
<body>
   <form method="POST" value="Not a bad button" action="http://localhost:8080/
DAT152WebSearch/updaterole">
       <input type="hidden" value="juan" name="username">
       <input type="hidden" value="ADMIN" name="role">
       <input type="submit" value="Submit">
   </form>
   <form method="POST" action="http://localhost:8080/DAT152WebSearch/updatepass-
word">
       <input type="hidden" value="Juanito12*" name="passwordnew">
       <input type="hidden" value="Juanito12*" name="confirm_passwordnew">
       <input type="submit" value="Submit">
   </form>
</body>
</html>
```

Analysis Technique:
Static code analysis and manual attempt to check the vulnerability.

Vulnerability #6: SSO OpenID authentication token (JWT)

Description:
Weaknesses in JWT authentication token - DAT152WebBlogApp

Possible consequences:

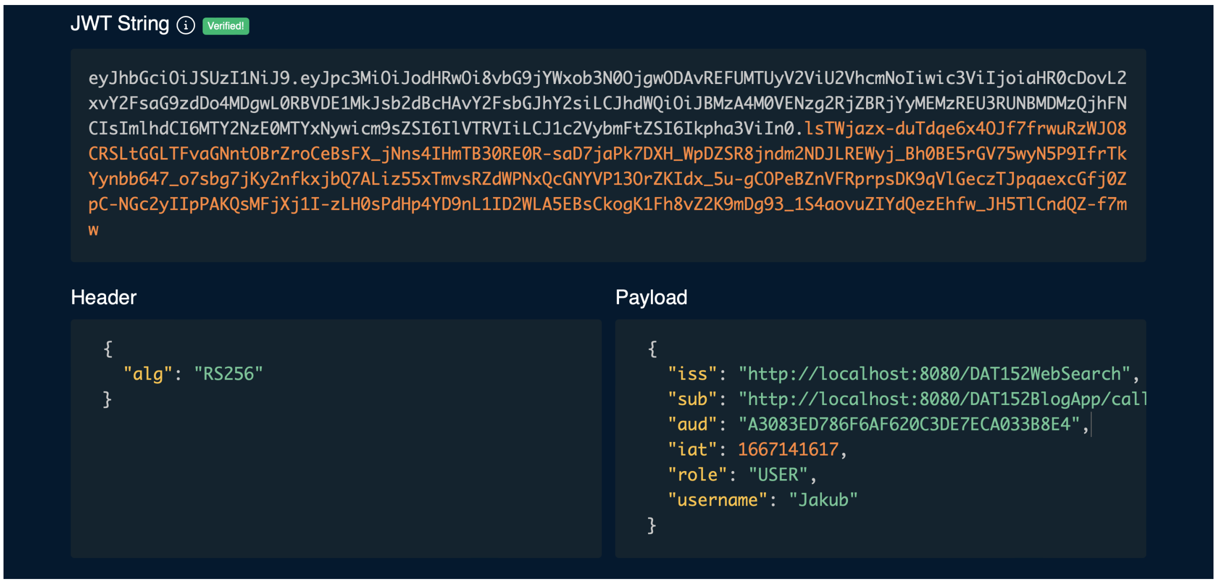
• **What is the id_token (authentication token) used to authenticate to the DAT152WebBlogApp?**
o Answer:
eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwOi8vbG9jYWxob3N0OjgwODAvRE-
FUMTUyV2ViU2VhcmNoIiwic3ViIjoiaHR0cDovL2xvY2FsaG9zdDo4MDgwL0RB-
VDE1MkJsb2dBcHAvY2FsbGJhY2siLCJhdWQiOiJBMzA4M0VENzg2RjZBRjYy-
MEMzREU3RUNBMDMzQjhFNCIsImV4cCI6MTY2NzE0MTkxNywiaWF0IjoxNj-
Y3MTQxNjE3LCJyb2xlIjoiVVNFUiIsInVzZXJuYW1lIjoiSmFrdWIifQ.V36Z7GV3D-
N8yElq6w7PXqWw0kyZAGZDJ9_W-eG8duIm6xIoPcDbfAjoIJiMparu_8grUCD6eoe-
PKt1wIe0asJgDNmn5BTRCt2gexVrRhsS9NK0uR4h94CMiBOf447scz89lcW-jWqbkr-
cYZWKp_-cY8o_CAkHpOKo6Ud0ZvVUfeT05HC2rPy6IsfN_vWXJOCe10D-
b4o0F_GjYjdTKRbG7JK9hEDQihAocgIm8NjtWPxhoVsQw-cwDccdaQc-
M2_M_ZBc0wvccr-uEvuPnjJUUJDVeN2jKWWsLpCUSHlCfJ3oAOdJIK3TWKF4-
ZOt6eb7zIQ479R20J44pDoNPpcVn-Lw

• **Where is the id_token (authentication token) stored in the client environment?**
o Answer:
In browser cookies

• **What are the vulnerabilities that you think exist in this id_token both from the IdP and SP endpoints?**
o Answer:
The token is not properly verified. Token doesn't have expiration time and it could be altered to provide ADMIN role.

• **What security decisions are being made using this id_token?**
o Answer:
The token sets username and role of the user.

• Can a user elevate his privilege in this id_token?
o Answer:
Original token



JWT String Verified!

```
eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwOi8vbG9jYWxob3N0OjgwODAvREFUMTUyV2ViU2VhcmNoIiwic3ViIjoiaHR0cDovL2
xvY2FsaG9zdDo4MDgwL0RBVDE1MkJsb2dBcHAvY2FsbGJhY2siLCJhdWQiOiJBMzA4M0VENzg2RjZBRjYyMEMzREU3RUNBMDMzQjhFN
CIsImlhdCI6MTY2NzE0MTYxNywicm9sZSI6IlVTRVIiLCJ1c2VybmFtZSI6Ikpha3ViIn0.lsTWjazx-duTdqe6x4OJf7frwuRzWJO8
CRSLtGGLTFvaGNntOBrZroCeBsFX_jNns4IHmTB30RE0R-saD7jaPk7DXH_WpDZSR8jndm2NDJLREWyj_Bh0BE5rGV75wyN5P9IfrTk
Yynbb647_o7sbg7jKy2nfkxjbQ7ALiz55xTmvsRZdWPNxQcGNYVP13OrZKIdx_5u-gCOPeBZnVFRprpsDK9qVlGeczTJpqaexcGfj0Z
pC-NGc2yIIpPAKQsMFjXj1I-zLH0sPdHp4YD9nL1ID2WLA5EBsCkogK1Fh8vZ2K9mDg93_1S4aovuZIYdQezEhfw_JH5TlCndQZ-f7m
w
```

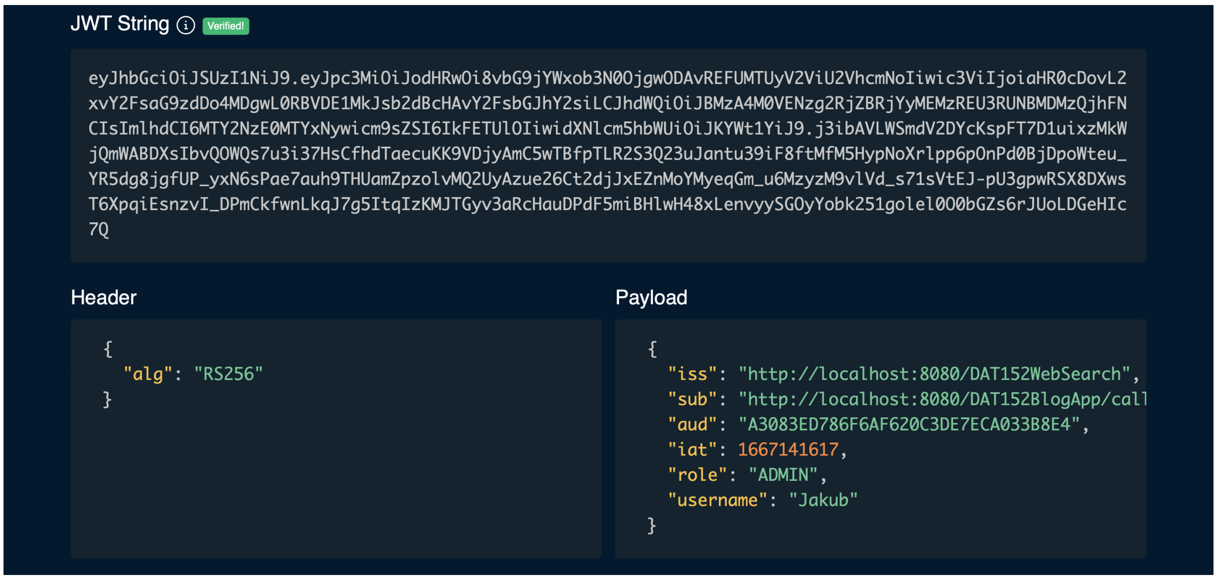
Header

```
{
  "alg": "RS256"
}
```

Payload

```
{
  "iss": "http://localhost:8080/DAT152WebSearch",
  "sub": "http://localhost:8080/DAT152BlogApp/call
  "aud": "A3083ED786F6AF620C3DE7ECA033B8E4",
  "iat": 1667141617,
  "role": "USER",
  "username": "Jakub"
}
```

Altered token with ADMIN privilege



JWT String Verified!

```
eyJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwOi8vbG9jYWxob3N0OjgwODAvREFUMTUyV2ViU2VhcmNoIiwic3ViIjoiaHR0cDovL2
xvY2FsaG9zdDo4MDgwL0RBVDE1MkJsb2dBcHAvY2FsbGJhY2siLCJhdWQiOiJBMzA4M0VENzg2RjZBRjYyMEMzREU3RUNBMDMzQjhFN
CIsImlhdCI6MTY2NzE0MTYxNywicm9sZSI6IkFETUlOIiwidXNlcm5hbWUiOiJKYWt1YiJ9.j3ibAVLWSmdV2DYcKspFT7D1uixzMkW
jQmWABDXsIbvQOWQs7u3i37HsCfhdTaecuKK9VDjyAmC5wTBfpTLR2S3Q23uJantu39iF8ftMfM5HypNoXrlpp6pOnPd0BjDpoWteu_
YR5dg8jgfUP_yxN6sPae7auh9THUamZpzolvMQ2UyAzue26Ct2djJxEZnMoYMyeqGm_u6MzyzM9vlVd_s71sVtEJ-pU3gpwRSX8DXws
T6XpqiEsnzvI_DPmCkfwnLkqJ7g5ItqIzKMJTGyv3aRcHauDPdF5miBHlwH48xLenvyySGOyYobk251golel0O0bGZs6rJUoLDGeHIc
7Q
```

Header

```
{
  "alg": "RS256"
}
```

Payload

```
{
  "iss": "http://localhost:8080/DAT152WebSearch",
  "sub": "http://localhost:8080/DAT152BlogApp/call
  "aud": "A3083ED786F6AF620C3DE7ECA033B8E4",
  "iat": 1667141617,
  "role": "ADMIN",
  "username": "Jakub"
}
```

Possible consequences: Normal user can act as admin. Also token cannot expire so it can be used by another user if he gets it.
File (s): Token.java, RequestHelper.java
Code: -
Payload: Token from cookies
Analysis Technique: I manually tried to change role in the token and use it in application to get admin privilege.

Mitigating vulnerabilities

Vulnerability #1: WSTG-ATHN-07: Testing for Weak Password Policy

Description:

I used hashing function PBKDF2WithHmacSHA1 which use salt. This function is more safer than MD5. Salt is always different for every password. This salt is stored in database together with hashed password. For this mitigation I had to change App-User table structure in database. Next I added password validator dependency. With this validator I was able to create validation function with specific rules. Passwords now have to be long between 8 to 16 characters. Also password has to contain at least one upper-case and lower-case character, at least one digit and one special symbol. Validator also checks for some illegal sequences.

Part of code (fixes):

I fixed code in files *Crypto.java, Validator.java*, *AppUser.java, AppUserDAO.java, MyContextListener.java, NewUserServlet.java and UpdatePasswordServlet.java.*

Mitigation/control code:

```
public static byte[] getSalt() {

        SecureRandom random = new SecureRandom();

        byte[] salt = new byte[16];

        random.nextBytes(salt);

        return salt;

}


public static String generateHashWithSalt(final String password, final byte[] salt) throws No-
SuchAlgorithmException, InvalidKeySpecException {

        KeySpec spec = new PBEKeySpec(password.toCharArray(), salt, 65536, 128);

        SecretKeyFactory factory =
SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");

        return new String(Base64.getEncoder().encode(

                factory.generateSecret(spec).getEncoded()));

}
```

```
public static Boolean validPassword(String password) {
        PasswordValidator validator = new PasswordValidator(
                new LengthRule(8, 16),
                new CharacterRule(EnglishCharacterData.UpperCase, 1),
                new CharacterRule(EnglishCharacterData.LowerCase, 1),
                new CharacterRule(EnglishCharacterData.Digit, 1),
                new CharacterRule(EnglishCharacterData.Special, 1),
                new IllegalSequenceRule(EnglishSequenceData.Alphabetical, 5, false),
                new IllegalSequenceRule(EnglishSequenceData.Numerical, 5, false),
                new IllegalSequenceRule(EnglishSequenceData.USQwerty, 5, false),
                new WhitespaceRule());

                return validator.validate(new PasswordData(password)).isValid();
}
```

I also changed existing code in AppUser.java, AppUserDAO.java, MyContextListener.java, NewUserServlet.java, UpdatePasswordServlet.java.

Vulnerability #2: WSTG-INPV-05: Testing for SQL Injection

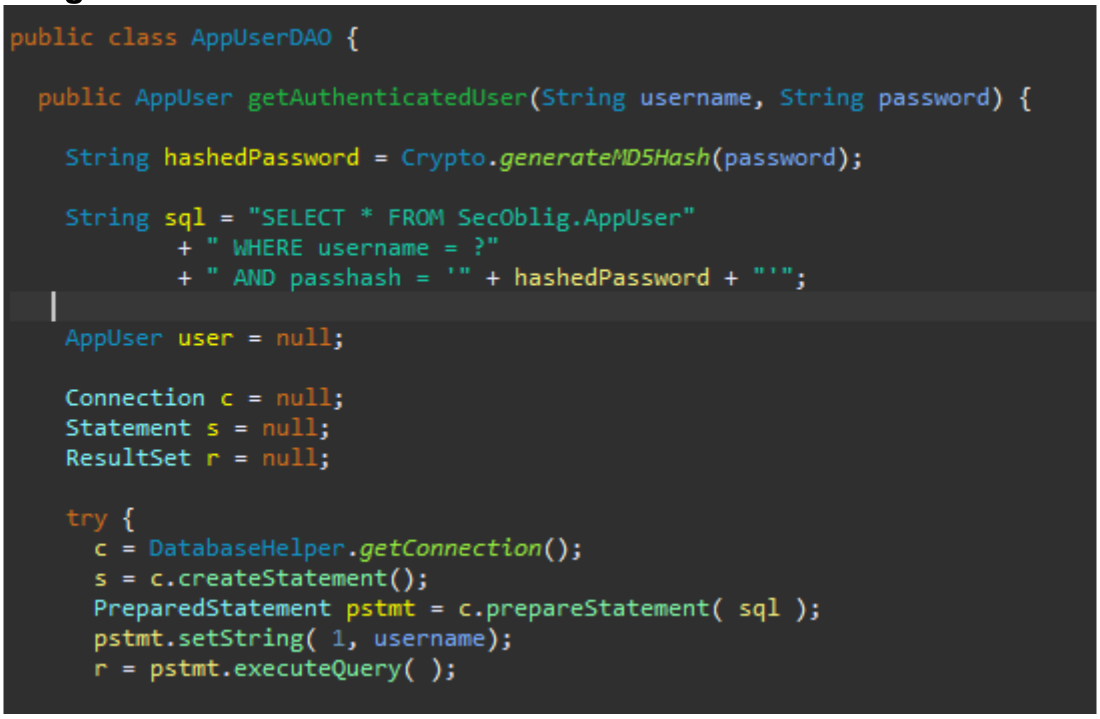
Description:

The login is Vulnerable to SQL Injection because the “getAuthenticatedUser” pass the input without checking it to the SQL Select and this is used to find the next user

Part of code:

The Function “ getAuthenticatedUser “

Mitigation/control code:



```
public class AppUserDAO {

  public AppUser getAuthenticatedUser(String username, String password) {

    String hashedPassword = Crypto.generateMD5Hash(password);

    String sql = "SELECT * FROM SecOblig.AppUser"
            + " WHERE username = ?"
            + " AND passhash = '" + hashedPassword + "'";

    AppUser user = null;

    Connection c = null;
    Statement s = null;
    ResultSet r = null;

    try {
      c = DatabaseHelper.getConnection();
      s = c.createStatement();
      PreparedStatement pstmt = c.prepareStatement( sql );
      pstmt.setString( 1, username);
      r = pstmt.executeQuery( );
```

Vulnerability #3 & #4: WSTG-INPV-01: Testing for Reflected Cross Site Scripting

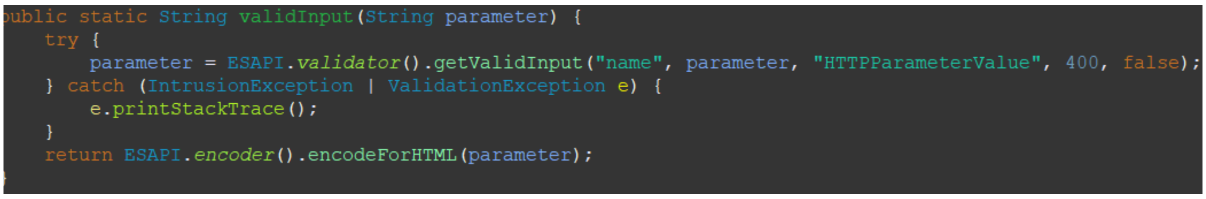
Description:

The *validString* function from the Validator.java file was just checking wether or not the String was null. Now we check if the string is valid with a new method with the ESAPI validator from OWASP.

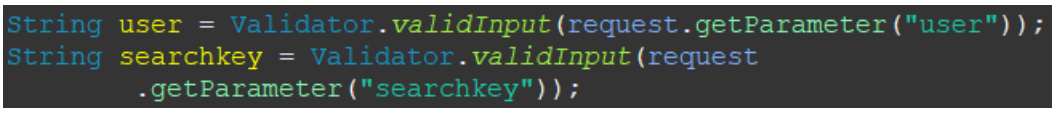
Part of code:

validInput function from Validator.java
doGet in SearchResultServlet.java

Mitigation/control code:



```
ublic static String validInput(String parameter) {
    try {
        parameter = ESAPI.validator().getValidInput("name", parameter, "HTTPParameterValue", 400, false);
    } catch (IntrusionException | ValidationException e) {
        e.printStackTrace();
    }
    return ESAPI.encoder().encodeForHTML(parameter);
```



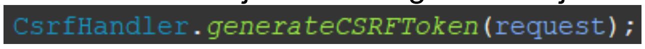
```
String user = Validator.validInput(request.getParameter("user"));
String searchkey = Validator.validInput(request
        .getParameter("searchkey"));
```

Vulnerability #5: WSTG-SESS-05: Testing for Cross Site Request Forgery

Description:
Synchronizer token pattern mitigation method has been implemented in the code.

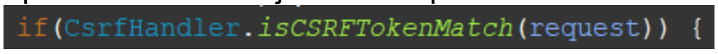
Part of code:
NewUserServlet.java, LoginServlet.java, UpdateRoleServlet.java, UpdatePassword-Servlet.java, CsrfHandler.java, updatepassword.jsp and updaterole.jsp

Mitigation/control code:
NewUserServlet.java and LoginServlet.java:



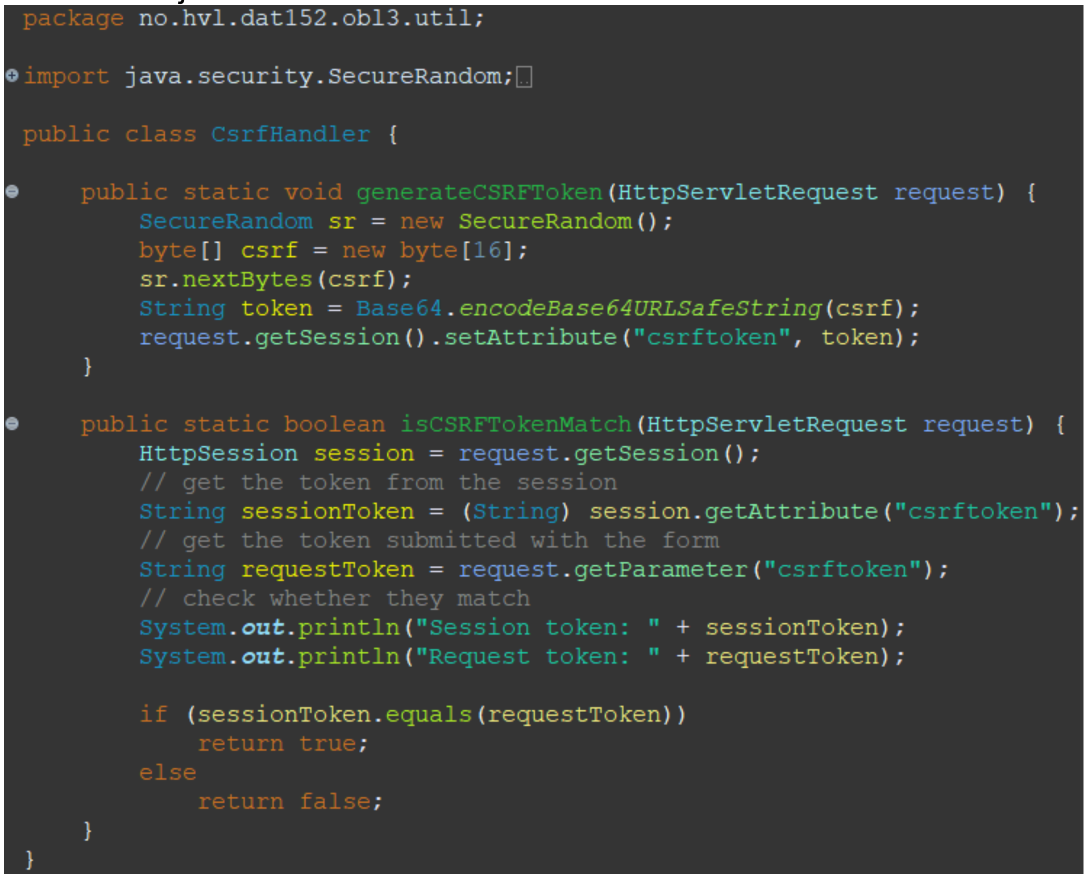
```java
CsrfHandler.generateCSRFToken(request);
```

UpdateRoleServlet.java and UpdatePasswordServlet.java:



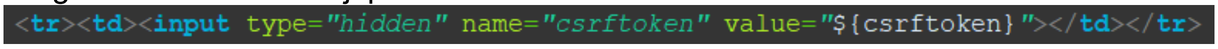
```java
if(CsrfHandler.isCSRFTokenMatch(request)) {
```

CsrfHandler.java



```java
package no.hvl.dat152.obl3.util;

import java.security.SecureRandom;

public class CsrfHandler {

    public static void generateCSRFToken(HttpServletRequest request) {
        SecureRandom sr = new SecureRandom();
        byte[] csrf = new byte[16];
        sr.nextBytes(csrf);
        String token = Base64.encodeBase64URLSafeString(csrf);
        request.getSession().setAttribute("csrftoken", token);
    }

    public static boolean isCSRFTokenMatch(HttpServletRequest request) {
        HttpSession session = request.getSession();
        // get the token from the session
        String sessionToken = (String) session.getAttribute("csrftoken");
        // get the token submitted with the form
        String requestToken = request.getParameter("csrftoken");
        // check whether they match
        System.out.println("Session token: " + sessionToken);
        System.out.println("Request token: " + requestToken);

        if (sessionToken.equals(requestToken))
            return true;
        else
            return false;
    }
}
```

Mitigation code for both jsp files:



```html
<tr><td><input type="hidden" name="csrftoken" value="${csrftoken}"></td></tr>
```

Vulnerability #6: SSO OpenID authentication token (JWT)

Description:

I added expiration time into token and used verifyJWTSignature for verifying if the token is original.

Part of code:

```
public static boolean isLoggedInSSO(HttpServletRequest request, String keypath) {
        String id_token = RequestHelper.getCookieValue(request, "id_token");
        doJWT(request, id_token);
        String idP = "http://localhost:"+Constants.IDP_PORT+"/"+Constants.IDP_PATH+"/
WEB-INF/";
        return JWTHandler.verifyJWT(id_token, keypath) && JWTHandler.verifyJWTSigna-
ture(id_token, idP);
}
```

Part of code in authorizationCodeRequest in Token.java:

```
Date expire = new Date();
expire.setTime(expire.getTime() + TimeUnit.MINUTES.toMillis(5));
jwt.setExp(expire);
```