Report Web Security

Gruppe 4: Maksim Ohvrill, Anton Tran, Bjørn Eide, Carl Alfred Emanuel Nordqvist

Repository (click me)

5.1 Part 1 - Identifying vulnerabilities

Vulnerability #1	WSTG-ATHN-07: Testing for Weak Password Policy
Description:	Evaluate if the web application has any form of weak password policy for their users. Like special characters, length, or any other requirements.
Possible consequences:	During an attack it will be easy to penetrate the application and run though very common and easily guessable passwords. This can result in bad security for the users that uses web application. That sets one of the CIA principles at risk.
File (s):	NewUserServlet.java UpdatePasswordServlet.java
Code:	NewUserServlet: line 36 -> 39 && 50 UpdatePasswordServlet: line 49 -> 59
Payload:	password, hello, 12345, “blank”, a
Analysis Technique:	Combination of manual testing like input on user creation, and SAST to find the vulnerability in backend going though code.

Vulnerability #2	WSTG-INPV-05: Testing for SQL Injection
Description:	Evaluating if there’s a possibility to breach the web application by SQL injection, on such as input dialogs.
Possible consequences:	Possibility to change or modify the existing SQL queries and retrieve data from backend. Or bypass the logon system for instance.
File (s):	LoginServlet.java,
Code:	LoginServlet: line 165 -> 173 AppUserDAO: line 14 -> 22
Payload:	1’ OR ‘1’=’1 admin’--
Analysis Technique:	Used DAST at first to test out if we could inject some queries in the input dialogs like on logon.

Vulnerability #3	WSTG-INPV-01: Testing for Reflected Cross Site Scripting
Description:	Evaluating application for eventual breaches though scripts that can be for example inserted though input dialogs. Since its reflected scripting, those scripts that was used we're targeting to manipulate only the user experience of the web application.
Possible consequences:	The user experience of the application can be affected, also the data that user is entering can be accessed or displayed for example. These two things on their own breaches the principles of CIA, both confidentiality and availability.
File (s):	LoginServlet.java SearchPageServlet.java -> Validator.java corresponding jsp's
Code:	LoginServlet: line 162 -> 173 Validator: line 8 -> 24
Payload:	<script>alert("mongus")</script> <input type="text" name="state" value="myInput ">
Analysis Technique:	Used DAST, just to try out the input dialogs in the application and they didn't filter any input at all it seemed. Then we looked at the source code, and our theory was almost true beside validator class for search servlet that had a simple check for that input wasn't a empty string or "null".

Vulnerability #4	WSTG-INPV-02: Testing for Stored Cross Site Scripting (XSS)
Description:	Evaluate application for potential breaches though scripting via input dialog on the page that can be stored system and effect other users experience.
Possible consequences:	The attackers can for example breach multiple users that is using the web application. Retrieve their input data or information stored in cookies for instance. This is a serious violation in security and cause data surveillance by attackers or just worse/abnormal application behavior.
File (s):	SearchPageServlet.java -> Validator.java corresponding jsp's
Code:	LoginServlet: line 162 -> 173 Validator: line 8 -> 24
Payload:	<script>alert("mongus")</script> <input type="text" name="state" value="myInput ">
Analysis Technique:	Used DAST, just to try out the input dialogs in the application and they didn't filter any input at all it seemed. Then we looked at the source code, and our theory was almost true beside validator class for search servlet that had a simple check for that input wasn't a empty string or "null".

Vulnerability #5	WSTG-SESS-05: Testing for Cross Site Request Forgery
Description:	An user that is identified and logged in, can be tricked into clicking some links or buttons. Those links and buttons can contain request, sometime automated javascript that can be harmful to the user. In a way that they can for example make a bank transaction from user's bank to attackers and so on. The principle in CSRF is that the request is coming from other site than origin one.
Possible consequences:	This type of request can cause both availability and integrity issues. Attacker can make request, that will add or remove data, in worse case access sensitive data, or just harm the users data as some examples.
File (s):	All jsp's with their corresponding servlets.
Code:	--||--
Payload:	<form action=" http://localhost:9092/DAT152WebSearchOblig3 /setusername" enctype="text/plain" method="POST"> <input name="username" type="hidden" value="CSRFd" /> <input type="submit" value="Submit Request" /> </form> As example attached to link sent via email, or XXS injected button
Analysis Technique:	Manually made a page outside of the project, that would make a request to project and tested.

Vulnerability #6	SSO OpenID authentication token (JWT)
Description:	Weaknesses in JWT authentication token
Possible consequences:	If it is possible to retrieve one single token, it can be used to log into multiple application that uses SSO authentication. That makes this extremely dangerous vulnerability.
Guiding questions/Answer format:	**What is the id_token (authentication token) used to authenticate too the DAT152WebBlogApp?** **Answer:** eyJhbGciOiJSUzI1NiJ9.eyJp c3MiOiJodHRwOi8vbG9jYWxob3N0OjkwOTIvREFUMTU yV2ViU2VhcmNoT2JsaWczIiwic3ViIjoiaHR0cDovL2xvY2Fs aG9zdDo5MDkxL0RBVDE1MkJsb2dBcHBPYmxpZzMvY2Fs bGJhY2siLCJhdWQiOiI5NjcyMjlCMDdFRUY1NDI5NjJDODYwR DdDM0QzNUQ1QiIsImlhdCI6MTY2NzIwOTU5NSwicm9sZSI6IlV TRVIiLCJ1c2VybmFtZSI6Im1tIn0.h2K25U1AExA1n418vsglQhN nIWkwN72mUXtv BQ5pOR_kfE8g8ZE22b4cJylRDKAJlz0b6fSxQ35xXcdHdpyw i_2eoVncA6rTMadA5gPcbTDWqK_dxHOnznM9xRcft69H k6VVZyFRIbdT3o5IwId5axf8XNMn7psDfjQe9Ey4Kkn3deng agBSW7iAtQpHZ1JG ZxZzQzaDmRdjvd4QwID7yf-KK0xAkKC0eJSilcNFnd2Ng9S nuUqCNAo8xd9CpB9ryqjuu N0DBZQIK6rLIjJ zcUli8lJdHfltLzLcjqqhWB0TCZGVcgd90eP-mOYtgiERvko Si3BbOhaEMKPdEWzg **Where is the id_token (authentication token) stored in the client environment?** **Answer**: as cookie in storage **What are the vulnerabilities that you think exist in this id_token both from the IdP and SPendpoints?** **Answer:** On register post request, there is possible to make an SQL – injection on IdP side. So there's possibility to retrieve SSO id_token to log in for the first user that was found, as an example. First, the id_token is stored as cookie, so even when the user is logging off normally from one application the token is still stored in the browser. Second, the id_token used to log in from the SP is passed on in redirect request and is visible on the page. That can be obtained by the attacker by "man in middle" principle.

	What security decisions are being made using this id_token? **Answer: (Mention what this id_token is used for within authentication and authorization)** To see if the user still has a valid SSO session. Logged in with SSO or exists in first place. Check if the request is coming from the same user who has the session. **Can a user elevate his privilege in this id_token?** **Answer: Show specific proof of your modifications and how you have exploited it.** Used “Burp Suite” to make modification to “POST – request” from a user with non-administrative privileges, where submit parameter were modified from “Post+comment” to “Delete+Comments”.
File (s):	RequestHelper.java Token.java BlogServel.java
Code:	Token: line 84 -> 86 RequestHelper: like 44 -> 50 BloggServlet: line 77
Payload:	POST request where parameter’s value submit were changed on intercept on proxy to “delete posts”
Analysis Technique:	SAST on backend code for analysis, but that didn’t give many results. Then went though backend code manually and found the “post – request” weakness in blogview.jsp. That makes it possible to delete comments for an user that don’t have administrative privileges aka “logged in as admin”.

5.2 Part 2 - Mitigating vulnerabilities

<table>
<tr><td colspan="2">Vulnerability #1: WSTG-ATHN-07: Testing for Weak Password Policy</td></tr>
<tr><td>Description:</td><td>When creating a new user it has no criteria for creating a password.
To fix this we need to implement criteria for password generation. Create a password checker that only passes through if the user enters a password with at least 5 characters. This can be expanded upon by adding other criterias like uppercase, special signs among others.</td></tr>
<tr><td>Part of code (fixes):</td><td>DoPost in NewUserServlet. Added new method for checking (PasswordValidering(String password)</td></tr>
<tr><td>Mitigation/control code:</td><td>

```
if (password.equals(confirmedPassword) && PassordValidering(password)) {
    AppUserDAO userDAO = new AppUserDAO();

    user = new AppUser(username, Crypto.generateMD5Hash(password), firstName, lastName, mobilePhone,
            Role.USER.toString(), Crypto.generateRandomCryptoCode());
    if (ValidateUser(user)) {
        successfulRegistration = userDAO.saveUser(user);

        if (successfulRegistration) {
            request.getSession().setAttribute("user", user);
            Cookie dicturlCookie = new Cookie("dicturl", preferredDict);
            dicturlCookie.setMaxAge(60 * 10);
            response.addCookie(dicturlCookie);

            response.sendRedirect("searchpage");
        }

    } else {
        request.setAttribute("message", "Registration failed!");
        request.getRequestDispatcher("newuser.jsp").forward(request, response);
    }
} else {
    request.setAttribute("message",
            "Password Must have at least one numeric character\r\n"
                    + "Must have at least one lowercase character\r\n"
                    + "Must have at least one uppercase character\r\n"
                    + "Must have at least one special symbol among @#$%\r\n"
                    + "Password length should be between 8 and 20");
    request.getRequestDispatcher("newuser.jsp").forward(request, response);
}
```

```
                && MobileMatch.matches());
  }

  public static boolean PassordValidering(String password) {
      String regex = "^(?=.*\\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[@#$%]).{8,20}$";
      Pattern pattern = Pattern.compile(regex);
      Matcher passwordMatch = pattern.matcher(password);
      return passwordMatch.matches();
  }
```

</td></tr>
</table>

Vulnerability #2: WSTG-INPV-05: Testing for SQL Injection	
Description:	AppUserDAO is vulnerable to sql injection, since the getAuthenticatedUser methode does not check the input.
Part of code:	getAuthenicatedUser in AppUserDAO.java
Mitigation/control code:	

```
public AppUser getAuthenticatedUser(String username, String password) {

    String hashedPassword = Crypto.generateMD5Hash(password);

    Pattern pattern = Pattern.compile("^[A-Za-z0-9]+$");
    Matcher match = pattern.matcher(username);
    boolean UsernameIsClean = match.matches();

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

    AppUser user = null;

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

    try {

        if (UsernameIsClean) {
            c = DatabaseHelper.getConnection();
            s = c.createStatement();
            r = s.executeQuery(sql);
```

Vulnerability #3: WSTG-INPV-01: Testing for Reflected Cross Site Scripting	
Description:	To eliminate reflected cross site scripting we have to control what the end user is allowed to input in the different fields. This is done by making a pattern i.e A-Za-z0-9. To make sure only letters from a to z in both uppercase and lower case are allowed to be used. This eliminated the ability to use <tags>. If the user enters allowed characters he is allowed to post it. Making sure we fill all the places where it's possible to input text.
Part of code:	SearchResultServlet.java

Mitigation/control code:	

```
        if (RequestHelper.isLoggedIn(request)) {

            String dicturl = RequestHelper.getCookieValue(request, "dicturl");
            if (dicturl == null) {
                dicturl = DEFAULT_DICT_URL;
            }

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

            // validate earchkey
            if (ValidateSearchKey(searchkey)) {

                Timestamp datetime = new Timestamp(new Date().getTime());
                SearchItem search = new SearchItem(datetime, user, searchkey);

                SearchItemDAO searchItemDAO = new SearchItemDAO();
                searchItemDAO.saveSearch(search);
                DictionaryDAO dict = new DictionaryDAO(dicturl);

                List<String> foundEntries = new ArrayList<String>();

                try {
                    foundEntries = dict.findEntries(searchkey);
                } catch (Exception e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

                request.setAttribute("searchkey", searchkey);
                request.setAttribute("result", foundEntries);
                request.getRequestDispatcher("searchresult.jsp").forward(request, response);
            } else if (!ValidateSearchKey(searchkey)) {

                AppUser authUser = (AppUser) request.getSession().getAttribute("user");
                List<SearchItem> top5history = new ArrayList<SearchItem>();

                if(authUser.getRole().equals(Role.ADMIN.toString())) {

                    SearchItemDAO searchItemDAO = new SearchItemDAO();
                    top5history = searchItemDAO.getSearchHistoryLastFive();
                }
                request.setAttribute("top5history", top5history);
                request.getRequestDispatcher("searchpage.jsp").forward(request, response);
            } else {
                request.getSession().invalidate();
                request.getRequestDispatcher("index.jsp").forward(request, response);
            }
        }

    }

    public static boolean ValidateSearchKey(String searchKey) {
        Pattern pattern = Pattern.compile("^[A-Za-z0-9]+$");
        Matcher match = pattern.matcher(searchKey);
        return match.matches();
    }
```

Vulnerability #4: WSTG-INPV-02: Testing for Stored Cross Site Scripting	
Description:	This is mostly eliminated the same way as reflected stored cross site scripting. By making sure the user only is allowed to enter Strings containing the letters A-Z and numbers 0-9, especially when registering an user. We can easily remove any possibility of an attacker storing dangerous scripts in the database for the admin users to be attacked by.
Part of code:	saveeSearch in SearchItemDAO.java New User in newUserServlet.java
Mitigation/control code:	

```
public void saveSearch(SearchItem search) {

    Pattern pattern = Pattern.compile("^[A-Za-z0-9]+$");
    Matcher match = pattern.matcher(search.getSearchkey());
    boolean IsSearchClean = match.matches();

    if (IsSearchClean) {
        String sql = "INSERT INTO SecOblig.History VALUES (" + "'" + search.getDatetime() + "', " + "'"
                + search.getUsername() + "', " + "'" + search.getSearchkey() + "')";

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

        try {
            c = DatabaseHelper.getConnection();
            s = c.createStatement();
            s.executeUpdate(sql);

        } catch (Exception e) {
            System.out.println(e);
        } finally {
            DatabaseHelper.closeConnection(r, s, c);
        }
    }else {
        System.out.println("Ulovelig Søk");
    }

}
```

```
  user = new AppUser(username, Crypto.generateMD5Hash(password), firstName, lastName, mobilePhone,
          Role.USER.toString(), Crypto.generateRandomCryptoCode());
  if (ValidateUser(user)) {
      successfulRegistration = userDAO.saveUser(user);

      if (successfulRegistration) {
          request.getSession().setAttribute("user", user);
          Cookie dicturlCookie = new Cookie("dicturl", preferredDict);
          dicturlCookie.setMaxAge(60 * 10);
          response.addCookie(dicturlCookie);

          response.sendRedirect("searchpage");
      }

public static boolean ValidateUser(AppUser appuser) {
    Pattern pattern = Pattern.compile("^[A-Za-z0-9]+$");
    Matcher UsernameMatch = pattern.matcher(appuser.getUsername());
    Matcher FirstnameMatch = pattern.matcher(appuser.getFirstname());
    Matcher LastnameMatch = pattern.matcher(appuser.getLastname());
    Matcher MobileMatch = pattern.matcher(appuser.getMobilephone());

    return (UsernameMatch.matches() && FirstnameMatch.matches() && LastnameMatch.matches()
            && MobileMatch.matches());
}
```

Vulnerability #5: WSTG-SESS-05: Testing for Cross Site Request Forgery	
Description:	Eliminating by creating CSRF-tokens as input element on page and adding same token to cookie in the JSP when it's accessed. When request is made the validation on servlet is made by comparing the token from input element to cookie token. If this check returns true, then the whole request is processed. Else the 401 exception is throwed "unauthorized access"
Part of code:	doAction() method in every servlet in controller package + corresponding adjustment in every JSP in WebContent that adds CSRF – tokens.
Mitigation/control code:	As an example, did this for every JSP and servlet.

```
196    public void doAction(HttpServletRequest request, HttpServletResponse response) {
197        // get the CSRF cookie
198        String csrfCookie = null;
199        for (Cookie cookie : request.getCookies()) {
200            if (cookie.getName().equals("csrfToken")) {
201                csrfCookie = cookie.getValue();
202            }
203        }
204        // get the CSRF form field
205        String csrfField = request.getParameter("csrfToken");
206
207        // validate CSRF
208        if (csrfCookie == null || csrfField == null || !csrfCookie.equals(csrfField)) {
209            try {
210                response.sendError(401);
211            } catch (IOException e) {
212                // ...
213            }
214            return;
215        }
216        // ...
217    }
218 }
219
```

```
23
24        <%
25        // generate a random CSRF token
26        String csrfToken = CSRF.getToken();
27
28        // place the CSRF token in a cookie
29        javax.servlet.http.Cookie cookie = new javax.servlet.http.Cookie("csrfToken", csrfToken);
30        response.addCookie(cookie);
31        %>
32        <input type="hidden" name="csrfToken" value="<%= csrfToken %>"/>
33
```

Vulnerability #6: SSO OpenID authentication token (JWT)	
Description:	SSO and weakness in JWT authentication token
Part of code:	Token.java authorizationCodeRequest method RequestHelper.java isLoggedInSSO method
Mitigation/control code:	**What vulnerabilities exist in this id_token from the IdP and SP endpoints?** Mitigation: Must fix in the “authorizationCodeRequest” method in the “Token.java” class in DAT152WebSearch (IdP) and the “RequestHelper.java” in the DAT152WebBlogApp (SP) and paste your code fixes here

```
                    jwt.setIat(new Date());

                    //fix
                    Date expDate = new Date();
                    expDate.setTime(expDate.getTime() + 10000);
                    jwt.setExp(expDate);
```

```
            //fix
            jwt.setExp(expDate);
```

```
    public static boolean isLoggedInSSO(HttpServletRequest request, String keypath) {

        String id_token = RequestHelper.getCookieValue(request, "id_token");
        doJWT(request, id_token);

        //fixed
        return JWTHandler.verifyJWT(id_token, keypath) && JWTHandler.verifyJWTSignature(id_token, keypath);

    }
```