**Identify vulnerabilities**

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

**Description**: Poor guidelines and enforcement when creating a password for your account. On this page there is no requirements for passwords other than having to type the same password twice.

**Possible consequences**: Easy to guess users’ passwords and take control of their accounts and leaking user data, which can further lead to reputational damage or legal consequences.

**File (s)**: newuser.jsp and/or NewUserServlet.java

**Code**: In newuser.jsp, should add some validation tags on the input (min length, regex making it contain numbers, capital letters, non-capital letters etc, can add max length)   
  
  
In NewUserServlet.java, add the same validation as the page above to password (or more if you skip frontend validation)  


**Payload**: username: Ivhene, password: , confirm\_password: ,first\_name: Ivar, last\_name: Nesheim, mobile\_phone: 47688381, dicturl: http://localhost:8080/DAT152WebSearchOblig4/V003/

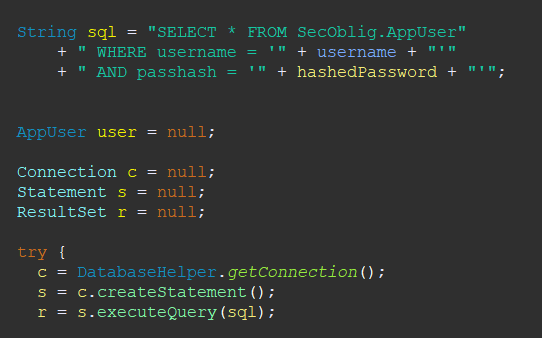
**Technique**: We found the vulnerability manually by attempting first a short password (123), then attempted to use no password and was allowed to make my account without a password.

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

**Description**: Can write malicious SQL in input fields which will manipulate the software the application and gives a user full control of said software.

**Possible consequences**: Gives any hacker full control over the application, they can give themselves admin privileges, get data from any user they want, or even delete or modify the data stored in the application

**File (s)**: AppUserDAO.java, SearchItemDAO (there is a marker on MyContextListener.java, but we are ignoring it since it is hardcoded in its current state)

**Code**: Common for all vulnerable code snippets are they have a variable in the sql sentence. This is the one we used, but there are about 10 other similar snippets where there is only a small difference, so we don’t gather all the snippets here and instead leave an image of the spotbugs scan showing where it detected SQL injection vulnerabilities as Appendix 1  


**Payload**: username: admin OR '1'='1' --, password: , client\_id: , response\_type: , scope: , redirect\_url: , state:

**Technique**: Found the bug by checking how the input was used in the SQL and figuring out how they could be manipulated. The input field used here is username during login. Used spotbugs to find further places where SQL injection could occur.

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

**Description**: Allows a user to pass in a script which will be executed upon loading the page. This can mean someone can create a link which will run a script once the page loads, or otherwise manipulate the page content.

**Possible consequences**: Allows attackers to create vulnerable links by creating a vulnerable script that will run upon loading the page. Could for example have a script that sends user data to an attacker if it is available to the script section.

**File (s)**: searchresult.jsp, mydetails.jsp, searchpage.jsp and updatepassword.jsp

**Code**: In searchresult.jsp  


In updatepassword.jsp (this will also go for all other pages where username is displayed and is part of the payload)



**Payload**: user: Ivhene, searchkey: <script>alert("XSS Vulnerability")</script>

**Technique**: Attempted to input this in all input fields and then checked where we would get alerts. Also thought where is any input displayed and checked if it would be possible to use here. Check appendix 2 for an image of the result of the testing script. Testing for reflected cross site scripting and testing for stored cross site scripting overlapped in our case

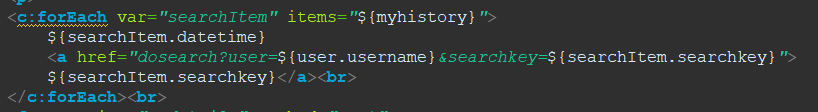
**Vulnerability #4**: WSTG-INPV-02: Testing for Stored Cross Site Scripting

**Description**: Allows a user to pass in a script which will be stored in the database and loaded by users when gathered from the database. The biggest vulnerability is where other users have this script loaded, which means in this case the searchpage.jsp is the most vulnerable page

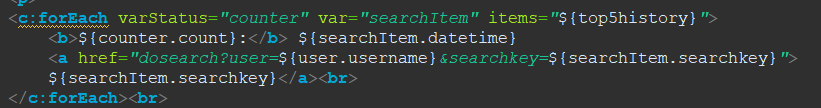
**Possible consequences**: Allows for scripts to be run when others are using the page, without the need for fabricated links. On this page, a consequence is that a user can create a script that will run on an admins pc since they always load the last 5 searches on the search page.

**File (s)**: mydetails.jsp and searchpage.jsp

**Code**: In mydetails.jsp



In searchpage.jsp



**Payload**:

**Technique**: Attempted to input this in all input fields and then checked where we would get alerts. Also thought where is any input displayed and checked if it would be possible to use here. Check appendix 2 for an image of the result of the testing script. Testing for reflected cross site scripting and testing for stored cross site scripting overlapped in our case

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

**Description**: An attacker tricks a user into unintentionally performing actions on a website they're authenticated into. The attacker creates a malicious request, which can for example be an image or a link, and then executes the request using the authenticated users account without the user knowing.

**Possible consequences**: Allows an attacked to perform unauthorized actions, like changing a user’s password or performing other actions in the name of another user.

**File (s)**: Specify the .java and/or .jsp files where the vulnerabilities are found.

**Code**: The vulnerable code snippet from the vulnerable file.

**Payload**: Specify the payload used (e.g., dictionary file)

**Technique**: How did you find the weakness? Manual, DAST, or SAST

**Vulnerability #6:** SSO OpenID authentication token

**Description**: Short description of the vulnerability you have found

**Possible consequences**: Short description of possible consequences (e.g., spoof identity of

victim)

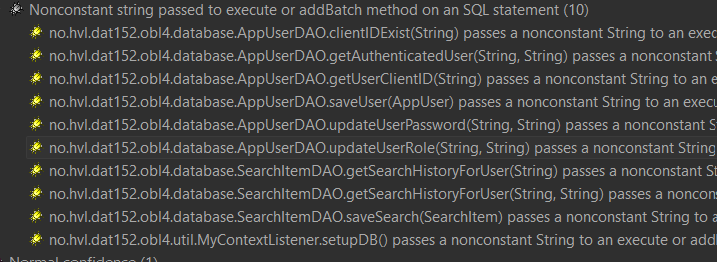
**File (s)**: Specify the .java and/or .jsp files where the vulnerabilities are found.

**Code**: The vulnerable code snippet from the vulnerable file.

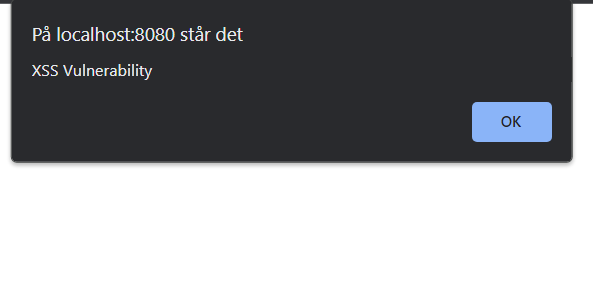
**Payload**: Specify the payload used (e.g., dictionary file)

**Technique**: How did you find the weakness? Manual, DAST, or SAST

**Appendix 1**

****

**Appendix 2**

****