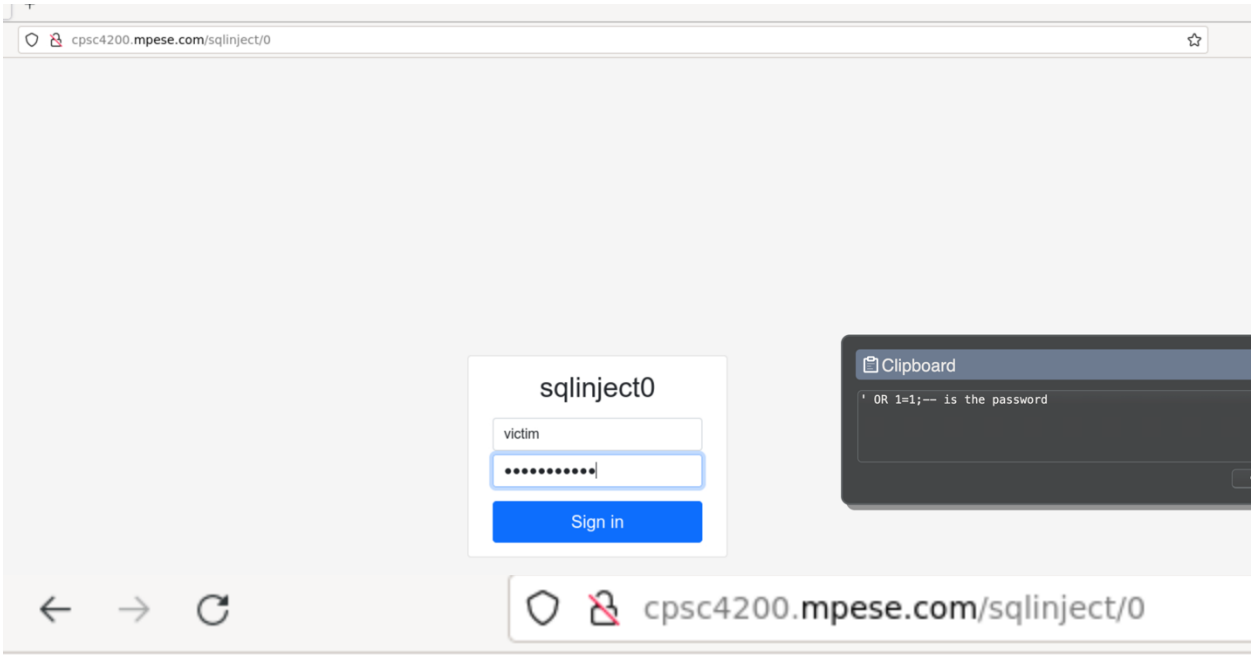
Homework 3 Writeup
Benjamin McDonnough

SQL Injections (1.1-1.3):

1.1 -



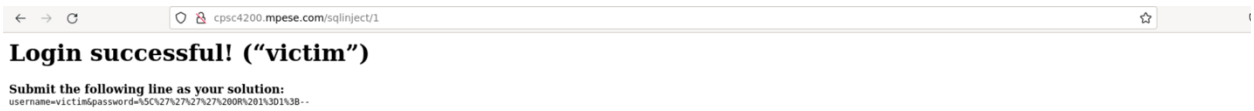
Login successful! (“victim”)

Submit the following line as your solution:

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

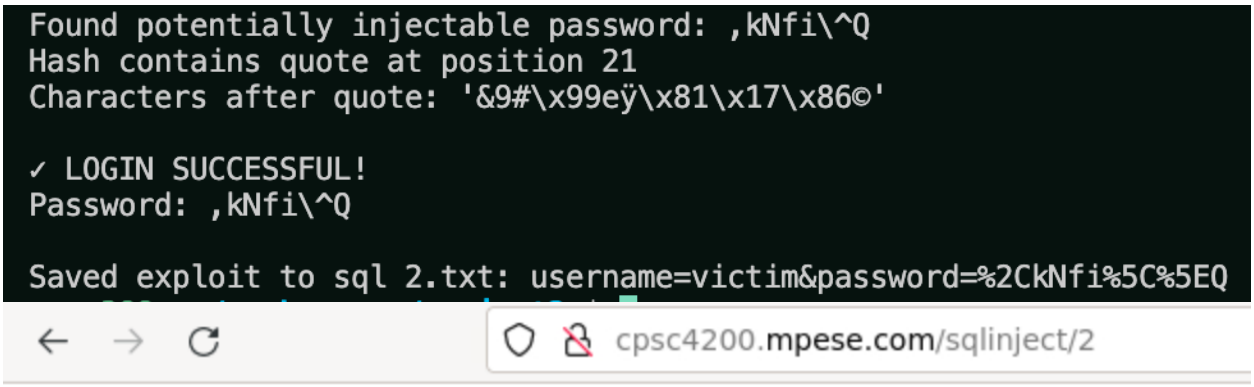
I logged into the victim account by using the ‘ OR 1=1;-- SQL injection. This essentially “tricked” the SQL database by returning the password as TRUE.

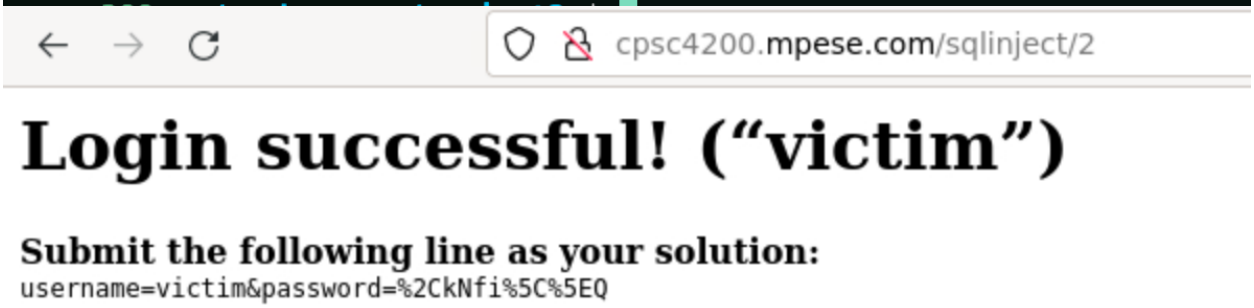
1.2 -



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 -

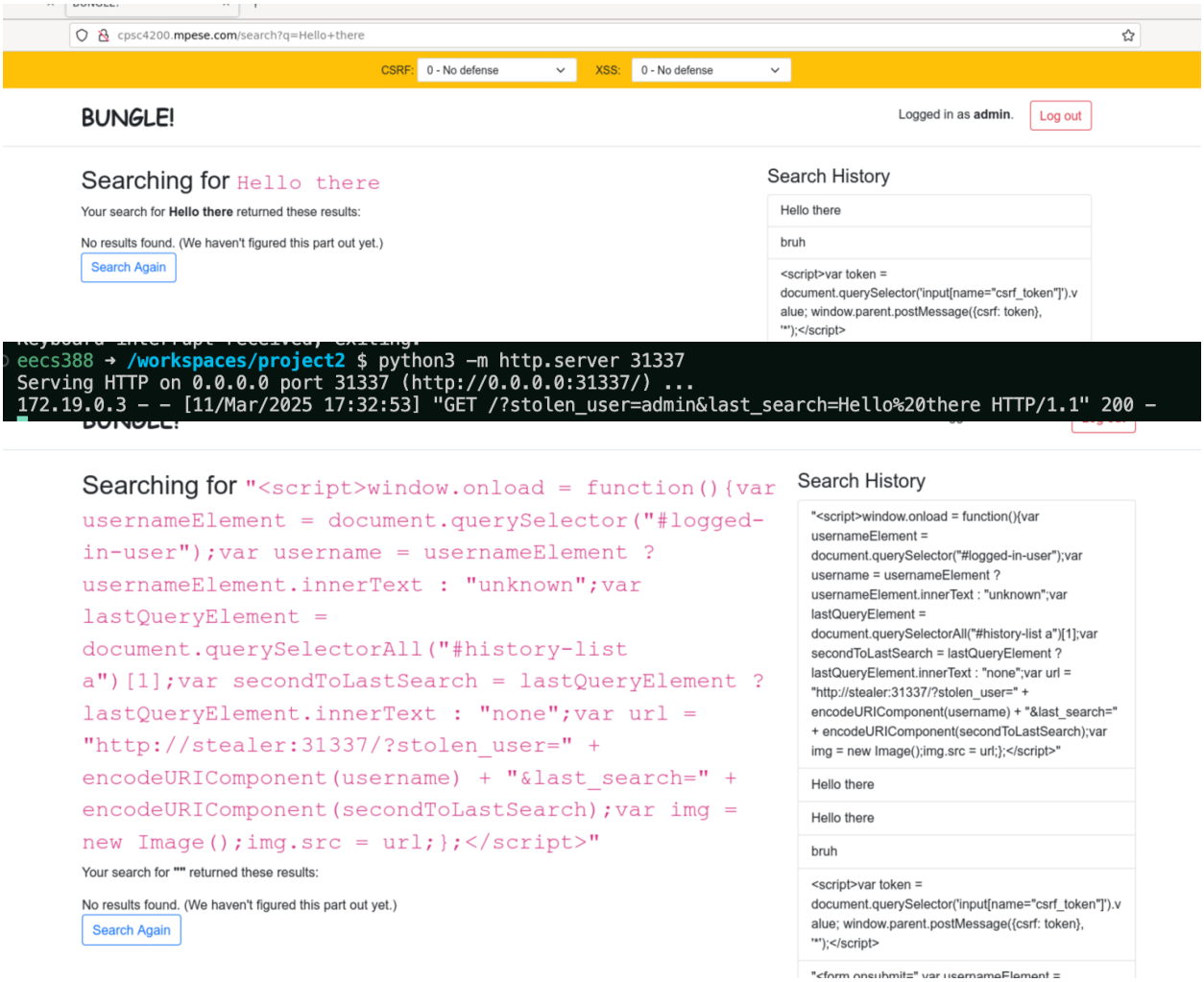




For the last part of the SQL injections, I created code that brute forces the problem. It creates a random 8 char long password that is comprised of all letters, numbers, and punctuation. It then adds the “bungle-“ to the beginning of the password and hashes the result. After that, it checks to see if 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.

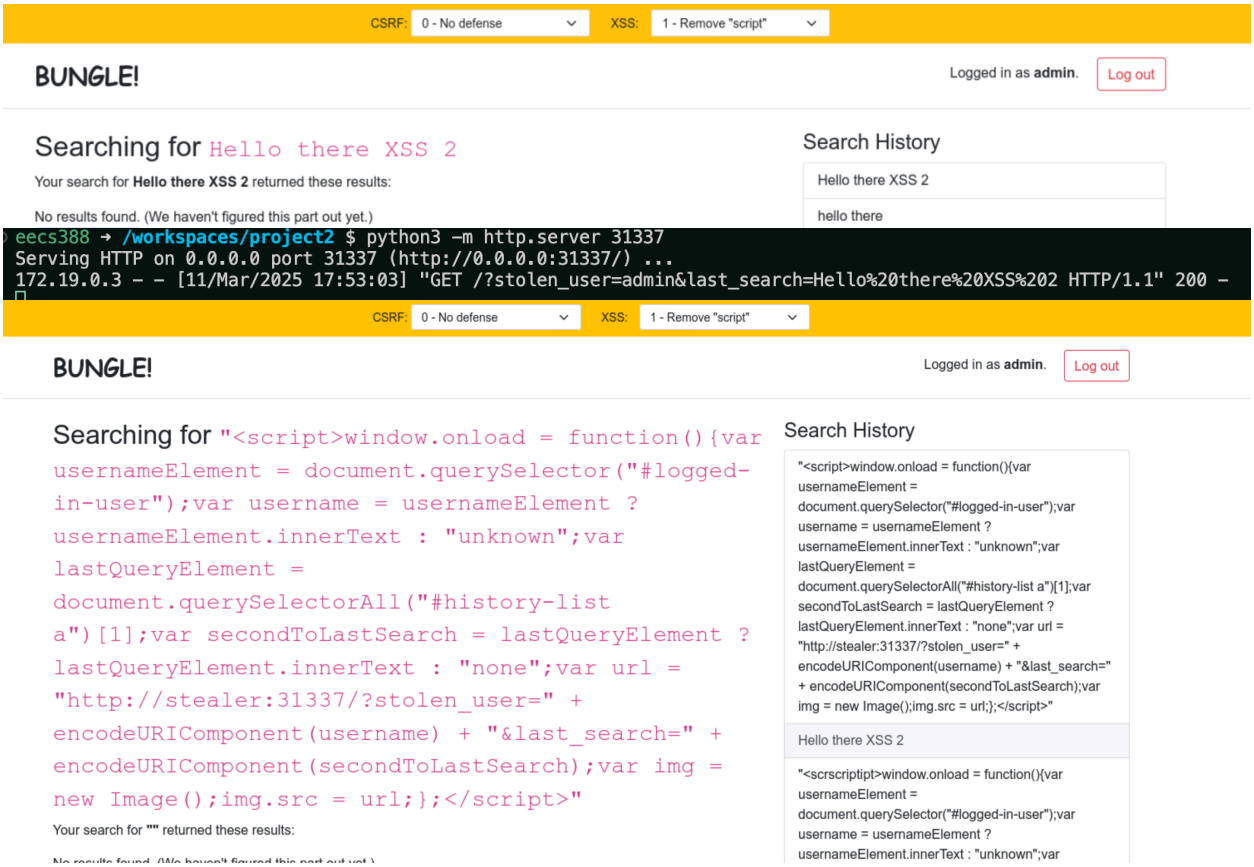
XSS (2.2.1-2.2.3):

2.2.1 -



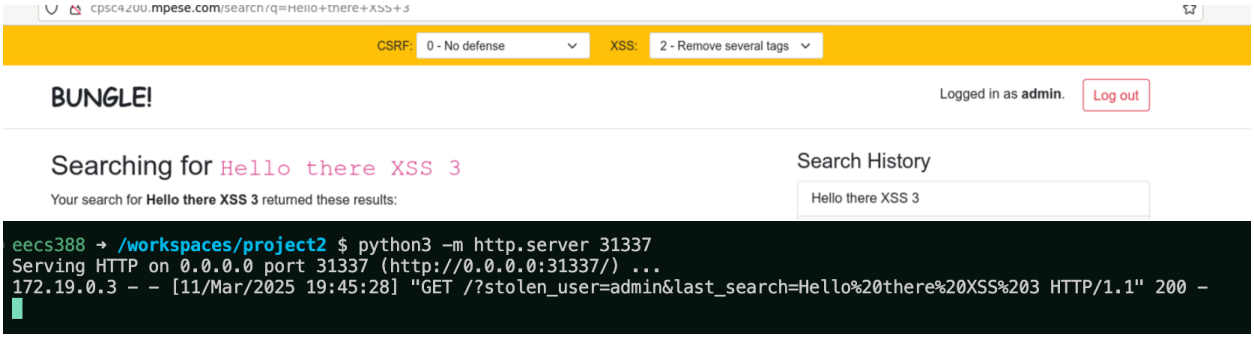
For the XSS attack, I created an html file that runs a JavaScript script. This script steals the user's username and last search by stealing the information from the website information. The username is saved under "logged-in-user" and the search history is saved under "history-list". I coded it to get 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=. 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.

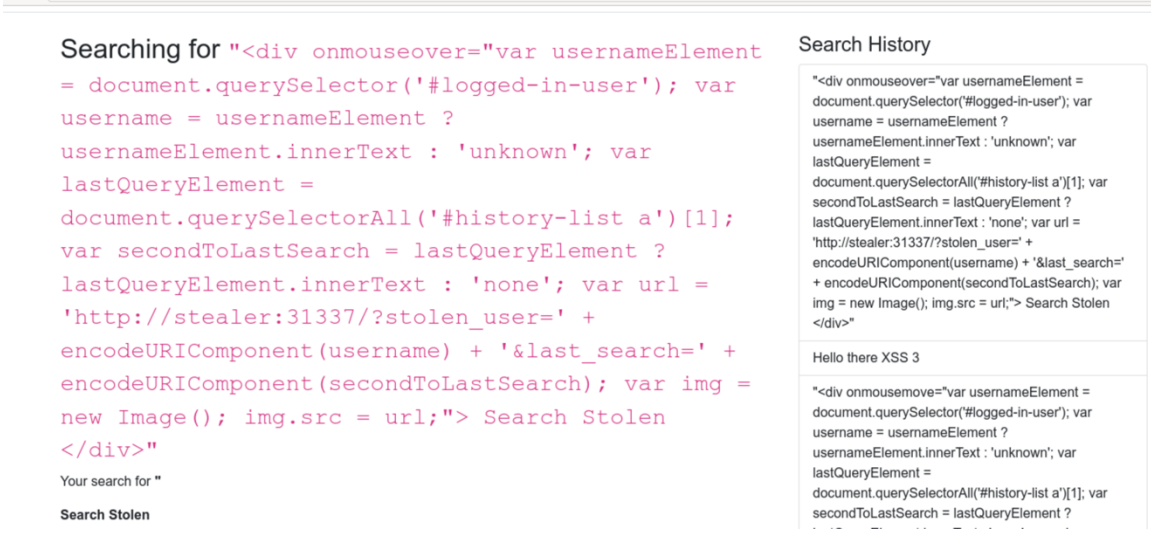
2.2.2 -



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 scrscriptipt), so that, when the script is taken out, it will still say script without the wrap around script being removed.

2.2.3 –

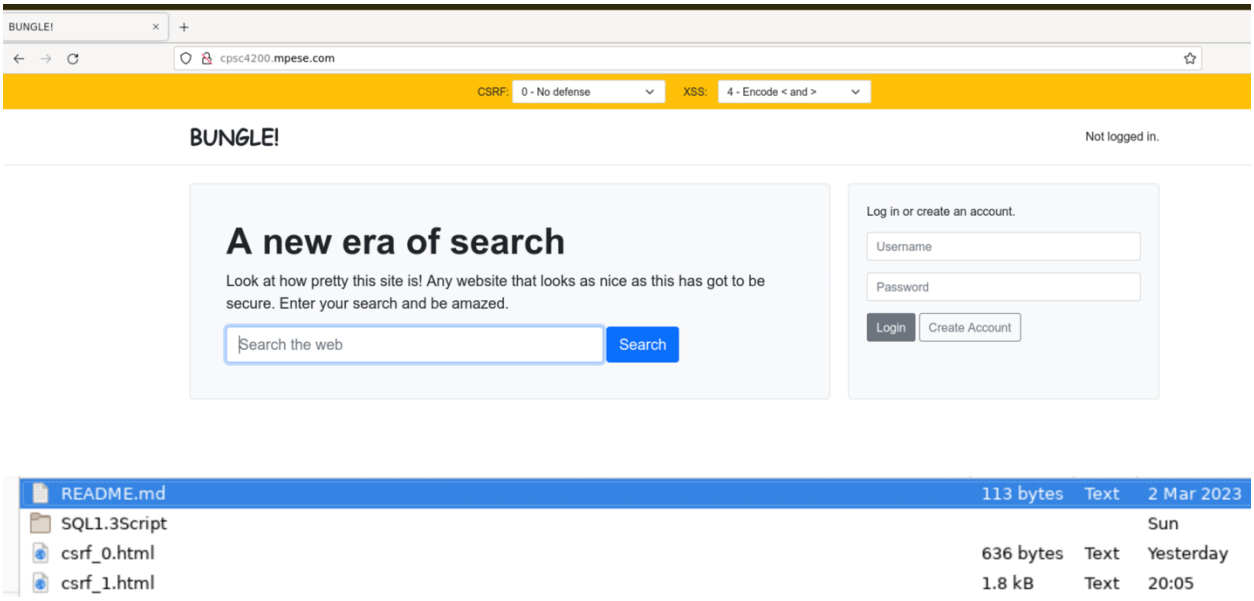


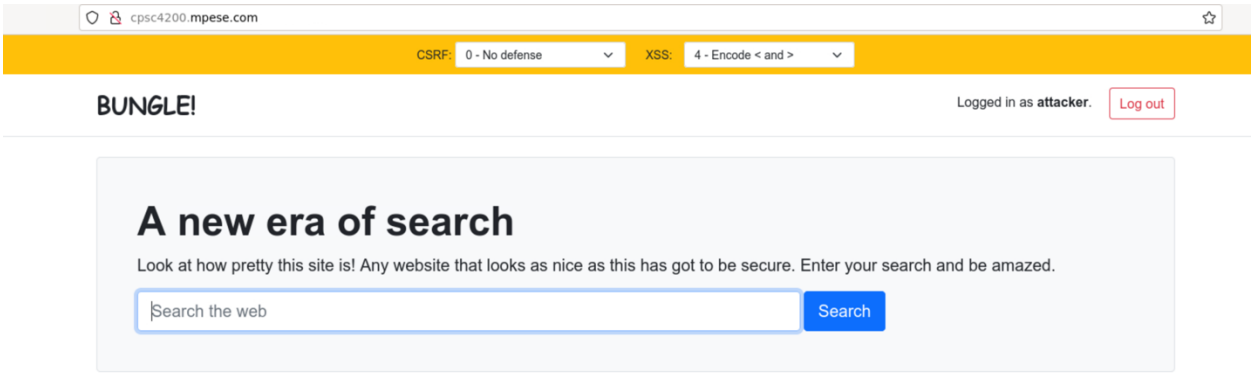


To change things up for the 3rd XSS attack, I created a div container that runs the same XSS script used before when the mouse moves anywhere on the page. While the script from the 2nd XSS attack would have worked on this, this new method provides a way to avoid the escaped tags completely.

CSRF (3.1 – 3.2):

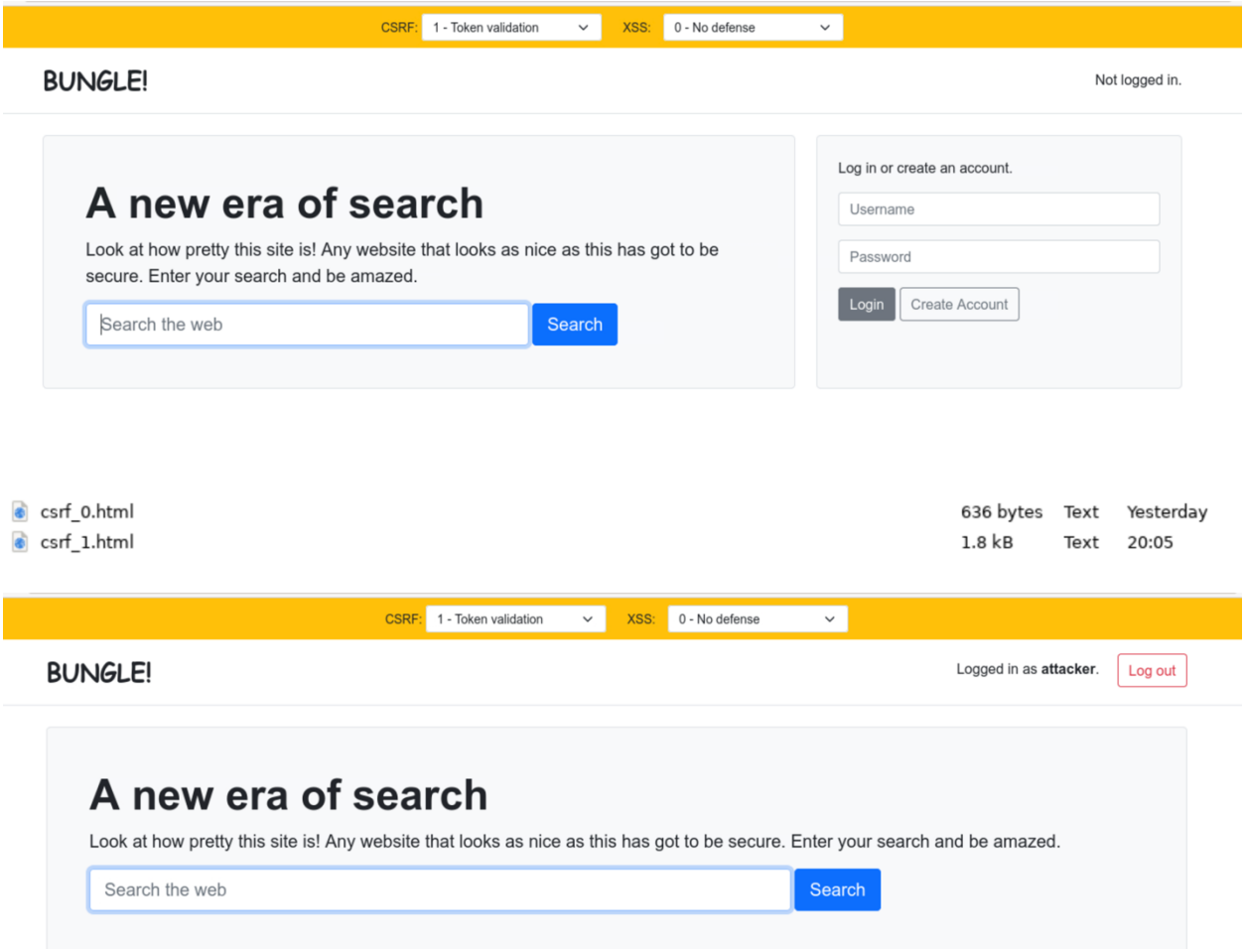
3.1 -





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 -



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.