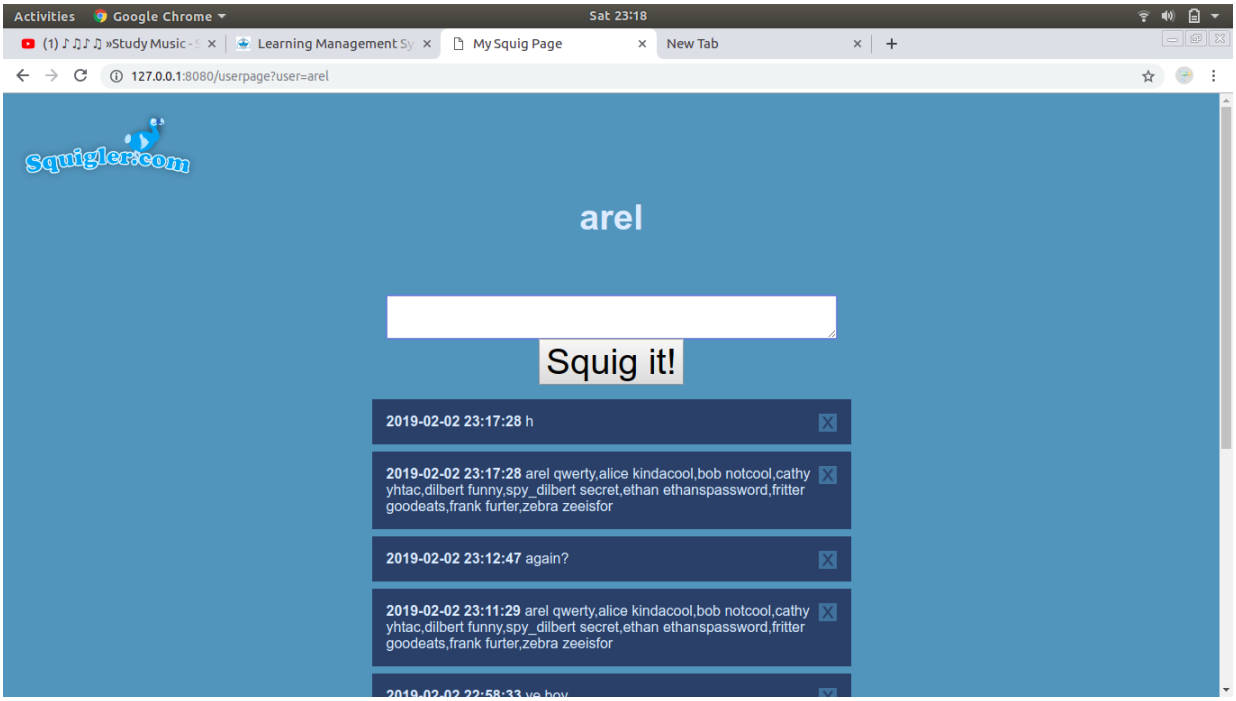
Take Home Tasks
Muhammad Haseeb
20100192

T 1.1- Change A User Password
Malicious Input: arel'; UPDATE accounts SET password='qwerty' WHERE username = 'arel';--
Input Field: Username Field in Login Form
Bried Explanation:

I have exploited the fact that the server can process more than one SQL statements at once because **executescript()** is used in the code. So, I have formatted an input with a SQL query to change a user's password.
How I Formatted: Basically, the first apostrophe in my input is for closing the matching quote in code then I ended the first statement with ';' and wrote my own statement. At the end, I have used comment sign for commenting out any remaining quotes and semi colons of the original code.

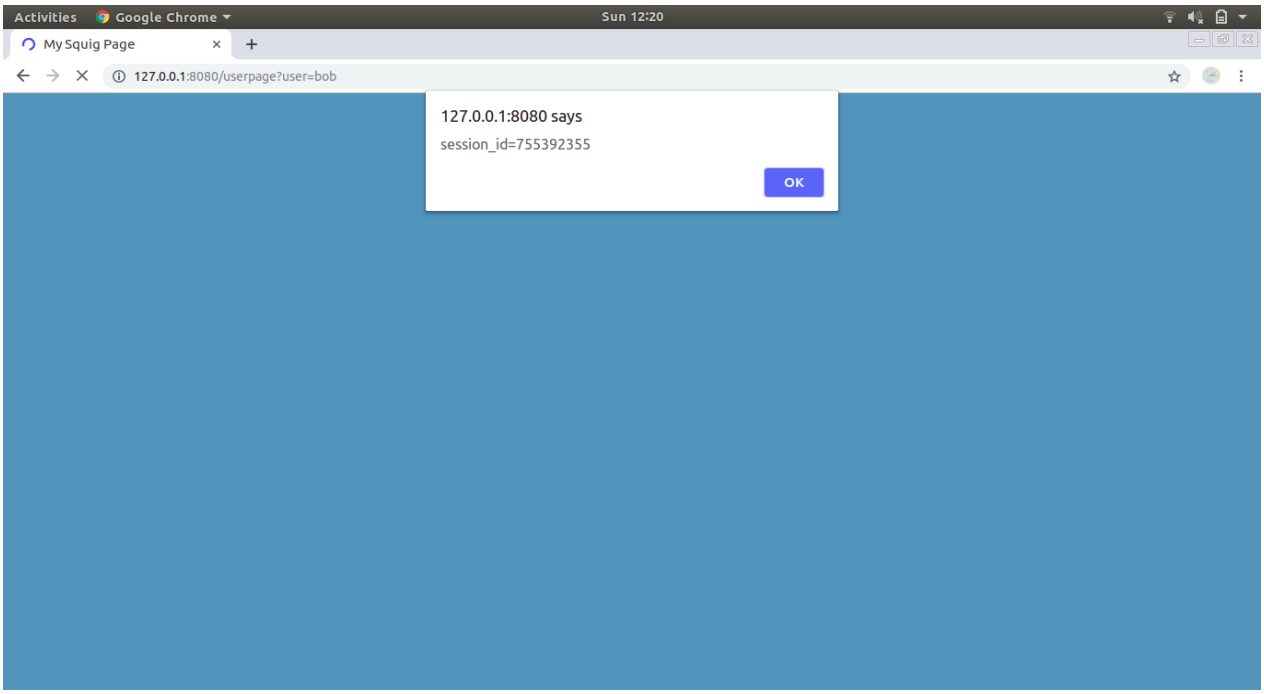
The screenshot shows that I am logged in with username "arel".

T 1.2- Password Dump Through Post
Malicious Input: hello', datetime('now', 'localtime'));INSERT INTO squigs VALUES('arel',(SELECT GROUP_CONCAT(username || ' ' || password) from accounts) ,datetime('now' ,'localtime')); --
Input Field: Post Field
Bried Explanation: I have again exploited the above discussed vulnerability of executing more than one SQL statements. Formatting semantics are also same as discussed above. Here, the tricky part was to write a SQL statement which does the job. And I have done it alright.



The second post shows the result of my attack.

T 2- Extended CSRF
Malicious Input: <a href = "do_squig?redirect=%2Fuserpage%3Fuser%3Ddilbert&squig=<script>alert(document.cookie)</script>">See the Hidden
Input Field: Post Field
Bried Explanation: Here I have merged CSRF and XSS attack and posted a link on my wall which leads any visitor to forcefully make a post (just like **Did I Post That?**). The only difference is that the content of that forced post is a script which pops up an alert box with user's cookie.



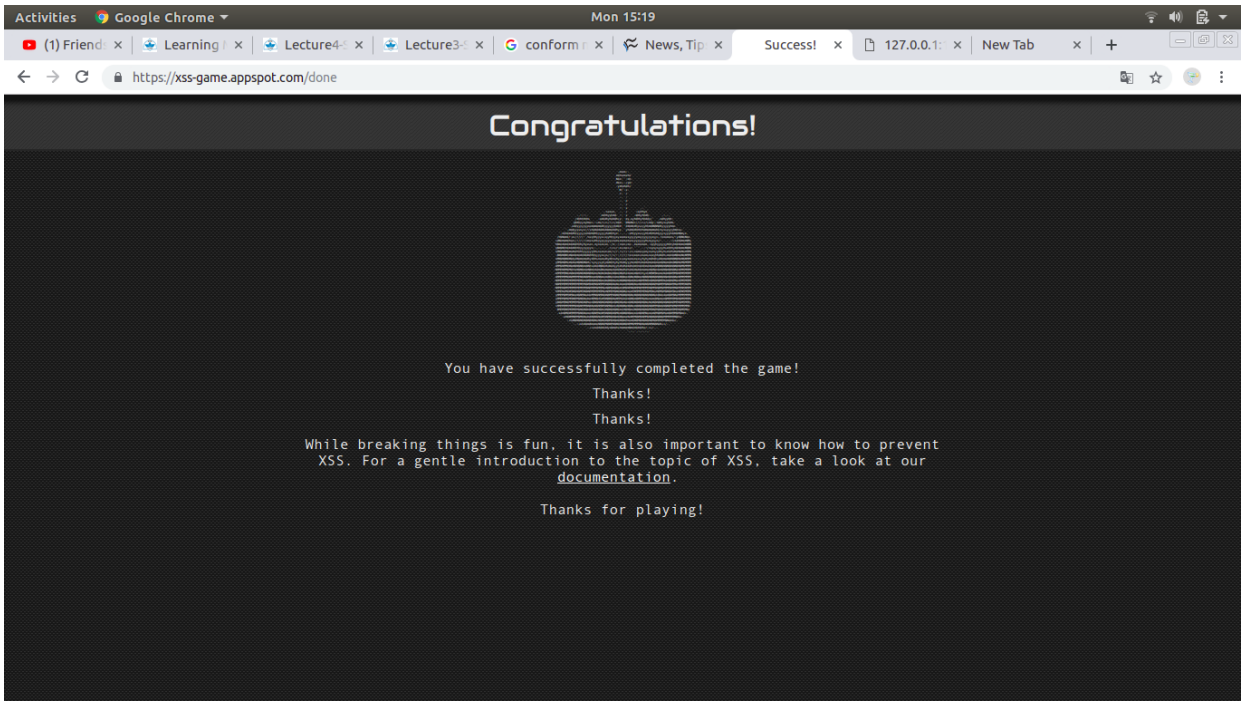
Screenshot shows victim's account with an alert box with cookie.

T 3- Vulnerability Defenses

I first tried to incorporate a secret key for each user which will get embedded to the page when a user requests a page. My goal was to make the page send the secret key to server along with any request so that it can be verified against wrong requests. But after googling a bit, I found that it is not a solid way to secure any site and it will assert many restrictions on user's actions. So, I discarded this approach and adopted the method to whitelist/blacklist the user's input. By checking for <script> and </script> tags **(for CSRF and extended CSRF),** my XSS defense was okay. Then I also checked for "logout\" (for **Task 1- log em out**)and some SQL (for **SQL injections**)specific words for making a working defense.
Now the site is secure and **none of the attacks can be made** to it as checks against script tags won't let any user embed any scripts and the checks against SQL will make users avoid putting SQL statements in any input fields. And any link going to 'logout\' will also be prevented.

Bonus Tasks:

Proof:



Brief Description:

Part 5 was fairly simple and straightforward as Sign Up form was taking a parameter from URL to do next step. I just wrote **javascript:alert("Hurrah")** as parameter to **next:** in the URL and that was it.

Part 6 was a bit confusing at first, but hints helped me a lot here. Actually, the page was showing a link from where it loaded a JS file (**/static/gadget.js).** Observing this gave me a hint to load external JS file. So, I just made a simple 5 lines' nodeJS host and hosted a small JS file containing the line **alert("Hurrah")** and gave its link **https:127.0.0.1:12000/JSFile.js.** Then the page displayed an error saying that https cannot be used so I observed the checks and found that the regular expression was not catering for case of letters (capital or small). So, I changed **https** with **Https** and it worked (final link: **Https:127.0.0.1/JSFile.js**).