Submitted By: Tasnim Ahmed
Email: tasnimahmed@iut-dhaka.edu
The assignment is available here.

DVA 489: Web Security

Assignment III: My first social media web application

Introduction

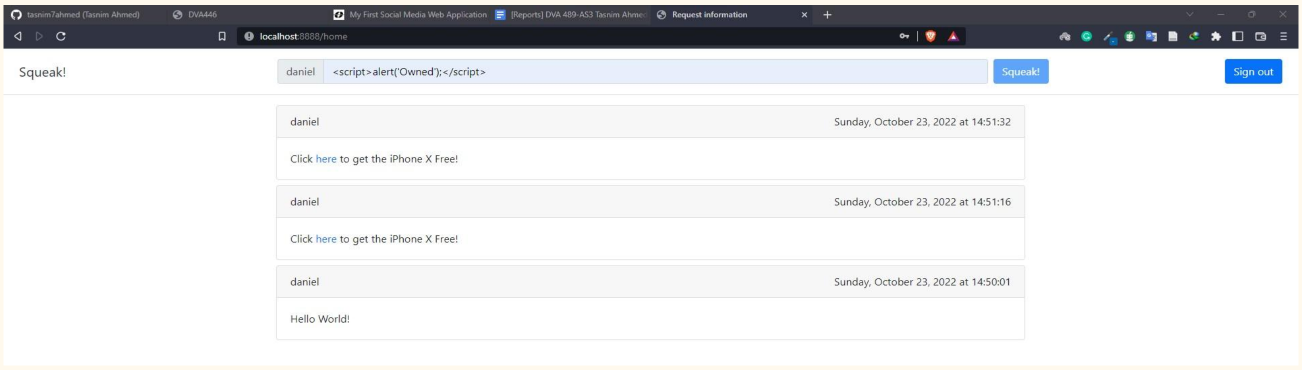
For our third project, we created an application called Squeak! which is a straightforward social networking tool that lets users sign up, log in, and publish brief messages. In accordance with the assignment's instructions, we added the authentication and authorization flow, which allows users to log in to the system by entering a valid username and password combination and clicking Sign In, or they can fill out the username and password fields and click Sign Up to register a new user, whose credentials will be stored in the "passwd" file. Our controller.js file's equivalent **postSignUp** and **postSignIn** functions handle the two distinct routes for these two occurrences, which are **/signup** and **/signin**, respectively. Prior to this, the application's base route (/) checks the cookie to verify whether a valid session is already present. If so, it immediately leads the user to the **/home** route; if not, the user is presented a page that combines the **Sign Up** and **Sign In** processes. The controller's **getIndex** function takes care of this. The **postSignUp** method also performs the following security checks as per our requirements:

1. The username is at least 4 characters long
2. The username is not already contained in the **“passwd”** file
3. The password is at least 8 characters long
4. The password does not contain the username

After registering a new user with a valid username and password, the credentials get appended to the **“passwd”** file. Upon registering, the user is directed to the **Sign In** page, where they may now login using the credentials they used to sign up. The user is shown the home page after successfully signing in, which has a textbox with a Squeak button at the top and the squeaks posted by all users across the remainder of the page. The **getHome** function in our controller loads this page by retrieving the squeaks from the **"squeaks"** file. The **postHome** function in our controller adds the username, post content, and time of each new squeak that a signed-in user publishes to the **"squeaks"** file. The user is then sent to the **/signout** route, which invokes the **getSignOut** function in our controller if they click the Sign Out button. Rerouting to the landing page via the base route, this approach deletes the cookie and logs the user out.

Stored XSS

Our program is hosted at **http://localhost:8888** in its unpatched form. and there was no mechanism added for filtering or escaping special characters in the post, which was done on purpose to make it vulnerable to the Cross-Site Scripting attack. Additionally, because we purposefully do not use the "httpOnly" and "secure" attributes, it is feasible to steal the cookie. Therefore, if we write a script in the post-input field, it will be executed as seen in the following images.

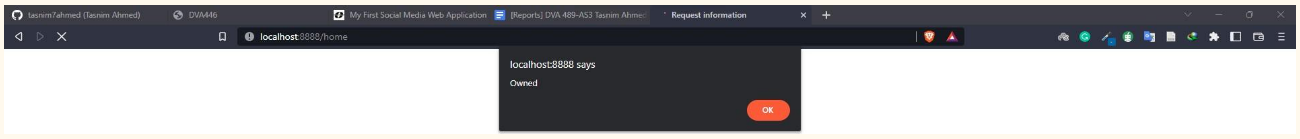

tasnim7ahmed (Tasnim Ahmed)
DVA446
My First Social Media Web Application
[Reports] DVA 489-AS3 Tasnim Ahmed
Request information
localhost:8888/home
Squeak!
Squeak!
Sign out

daniel <script>alert('Owned');</script>

daniel Sunday, October 23, 2022 at 14:51:32
Click here to get the iPhone X Free!

daniel Sunday, October 23, 2022 at 14:51:16
Click here to get the iPhone X Free!

daniel Sunday, October 23, 2022 at 14:50:01
Hello World!


tasnim7ahmed (Tasnim Ahmed)
DVA446
My First Social Media Web Application
[Reports] DVA 489-AS3 Tasnim Ahmed
Request information
localhost:8888/home
OK

localhost:8888 says
Owned

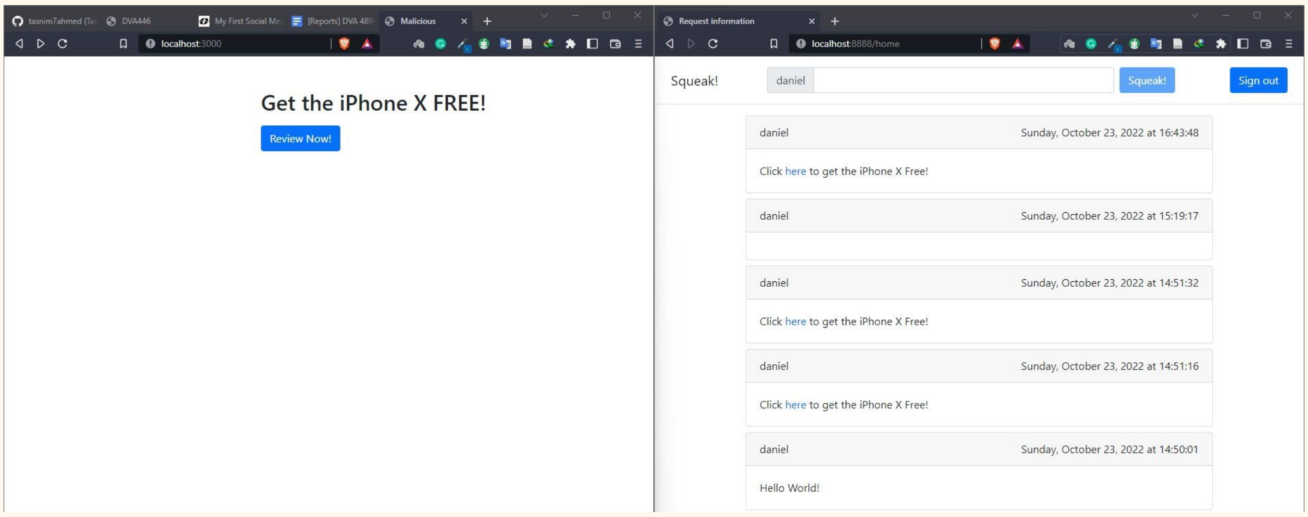
Protecting the Application

In order to defend against Stored XSS assaults, there are two actions to take. To start, escape all malicious data from the posts. The cookie's attributes should then be set to safeguard it. Our program, hosted at http://localhost:8888, uses these measures in its Patched version. As an alternative to utilizing our own escaping mechanism, we reimplement our application using Mustache, which automatically escapes all data while generating pages. We create two files in the "**views**" directory with the names "**landingPageView.mustache**" and "**homeView.mustache**", reusing the code from our HTML files. We also added a "**maxAge**" attribute and set the "**httpOnly**" and "**secure**" properties to true when setting the cookie after a successful sign-in, which further protects it.

Cross-Site Request Forgery (CSRF)

The CSRF attack is a cyberattack carried out by a malicious website against a legitimate user-authenticated website. The automatic transmission of authorization data e.g., cookie, along with every request makes this attack viable. Our program is purposefully designed to be vulnerable to CSRF attacks when it is not patched. The unpatched version is hosted at http://localhost:8888.

We create another attacker-controlled webpage and host it at http://localhost:3000 to show the CSRF attack. The rogue site has a single "**Review Now**" button that, when clicked by an authenticated user, immediately posts a squeak with a link to the malicious site in the Squeak! Homepage. Any person who clicks the link in the post will be sent to the malicious website, where they risk being attacked once more.



Get the iPhone X FREE!

Review Now!

Squeak! daniel Squeak! Sign out

daniel Sunday, October 23, 2022 at 16:43:48
Click here to get the iPhone X Free!

daniel Sunday, October 23, 2022 at 15:19:17

daniel Sunday, October 23, 2022 at 14:51:32
Click here to get the iPhone X Free!

daniel Sunday, October 23, 2022 at 14:51:16
Click here to get the iPhone X Free!

daniel Sunday, October 23, 2022 at 14:50:01
Hello World!

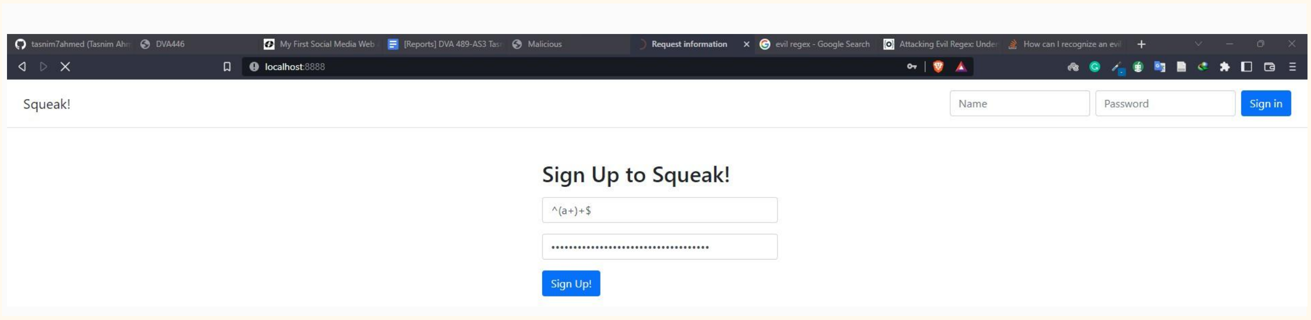
Protecting the Application

We assign a randomly generated CSRF token to each session in order to protect our application from CSRF attacks. The /**signin** route effectively starts the development of the random CSRF token as it is produced each time a user successfully logs in from the Sign In/Sign Up page. We have not assigned a CSRF token to the /**signup** route since, in our implementation, it just adds a new user to our "**passwd**" file rather than establishing a valid session. Next, this CSRF token is kept in **SESSION IDS**, which the server has specified, and is kept in an index depending on the cookie. The Squeak form page includes a hidden field that contains the CSRF token as well. If the embedded token and the value saved on the server do not match when a new squeak is about to be posted, we simply stop the post from being added to our "**squeaks**" file. Our application is prepared to produce a new CSRF token and store it at the new location based on the new cookie that is established for the new valid session if we hit the /**signin** route again because the /**signout** route deletes the cookie, rendering our current CSRF token unlocatable.

Regular Expression Denial of Service (ReDOS)

Since the portion of our code that validates the integrity of the login and password employs Regular Expressions, as can be seen from the image, the unpatched version of our program is susceptible to ReDOS attacks. Our server appears to stop responding when a user enters an "evil Regex" like "**^(a+)+$**" in the username and "**aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa!**" in the password field.

```
if (signuppassword.match(new RegExp(signupusername)) != null) {
    console.log(signuppassword.match(new RegExp(signupusername)));
    flag = true;}
```



^(a+)+$

Protecting the application

In order to protect our application, we implement our own input validator which checks whether the input contains special characters like “^”, “*”, “+”, etc.