Group 14 Presents:

BrickFlix

**Developed by Conrad Schneggenburger, Jared Short, Gianna Mule (and a little bit of Brian Bullis until he abandoned our cause)**

**Users Manual:**

To get started, pull the repository from [*https://github.com/GiaQM115/CSEC380Group14*](https://github.com/GiaQM115/CSEC380Group14). Navigate to the directory titled “docker” and execute the following command:

docker-compose up --build

Wait about 30 seconds after the image is built for the database to initialize.

**Activity 3:** Authentication

*Test:* test\_login.py

For authentication, we chose to use PHP and the Delight/IM library. Our user information is stored in the database, passwords are salted, and sessions are taken care of via the library. When you login, if your credentials are incorrect, you are redirected back to the login page, with empty fields for the username and password – oops, try again! If proper credentials are provided, authentication is successful and you can now see our homepage. Have fun!

*Questions:*

1. **Provide a link to the test cases you generated for this activity.**

[**https://github.com/GiaQM115/CSEC380Group14/tree/master/tests**](https://github.com/GiaQM115/CSEC380Group14/tree/master/tests)

2. **How do you ensure that users that navigate to the protected pages cannot bypass authentication requirements?**

Every page on the site will run a check to see if the user is authenticated (by referring to the session ID). If the user is not authenticated, they will be redirected to the login page. If the user attempts to access a not visitable php file (such as a form handler file), they will received a 401 error in response.

3. **How do you protect against session fixation?**

To protect against session fixation, we will need prevent users from being able to specify their own session ID upon login. If users can specify their own ID, then sessions must also be checked for integrity when they are provided to the server to protect against tampering/forging.

4. **How do you ensure that if your database gets stolen passwords aren't exposed?**

Protecting the confidentiality of the passwords in the database will require the use of hashing combined with salts. Even if the attacker manages to dump the database, the passwords remain cryptographically secure if exposed.

5. **How do you prevent password brute force?**

In order to prevent password brute force, we will need to significantly slow down the time it will take to do so. Much like ssh\_guard, we can implement a time-based lockout system that will block the user's IP for a certain amount of seconds if the attacker exceeds the maximum amount of tries per second.

6. **How do you prevent username enumeration?**

To prevent username enumeration, we will not only need to secure the database from SQL injections, but we will also need to NOT disclose whether or not the username is incorrect. In other words, if an attacker guesses the correct username and is met solely with a "wrong password" message, they will know the username is legitimate.

7. **What happens if your sessionID is predictable, how do you prevent that?**

If session IDs are predictable, it will allow the attacker to forge a legitimate session ID that may belong to another user, granting them access to their session. This can be prevented by generating cryptographically secure session IDs - i.e., truly random bit sequences.

**Activity 4:** The Content

*Test:* test\_video.py

Uploading and deleting videos is easy on BrickFlix. Simply navigate to your dashboard using the button in the upper right hand corner of the home page. You can upload a file from your machine, or a new video via it’s URL. To delete a video you’ve posted, use the drop down menu at the bottom of the dashboard. To view your videos, simply go back to the home screen. You can also search for a video or a user with our handy search bars. If you’re searching for a video but don’t remember the name, try the first letter followed by an “\*”. This will trigger a type of auto-fill and show you all videos that start with that letter.

*Questions:*

1. **How do you prevent XSS is this step when displaying the username of the user who uploaded the video?**

We prevent XSS by using CSP to not allow any scripts to run on the page. This isn’t a problem because we do not use Javascript on our server anyway.

2. **How do you ensure that users can’t delete videos that aren’t their own?**

We ensure that users can delete only their videos by using a drop down menu to limit their options to only the videos they’ve uploaded. There is also a check on our end prior to deletion to ensure the requesting user is the uploader.

**Hackers Manual**

**Activity 5:** SQL Injection

*Tests:* test\_blind\_sql.py, test\_classic\_sql.py

To achieve classic SQL injection, exploit our search by user feature. You will be able to see database information (be it an error or a success message) which will help you to gather information. For blind SQL injection, exploit our video naming feature when uploading from a URL. You won’t get any information from the database (because it’s blind), but the servers behavior may help you out.

*Questions:*

1. **How would you fix your code so that these issues were no longer present?**

Any input into the fields will not only need to be sanitized after being submitted, but will also need to be sanitized on output. Output should also be the least verbose as possible.

2. **What are the limitations, if any that, of the SQL Injection issues you’ve included?**

The SQL injections in this context will only affect the brickflix database, but not any other databases. This is because we have restricted the PHP DB user to only be allowed to modify the "brickflix" database. Otherwise, the possibilities are, for the most part, limitless.

**Activity 6:** SSRF

*Test:* test\_ssrf.py

You can leverage the URL download feature to achieve server side request forgery. Here, we won’t bother checking if you’re actually uploading an mp4 – BrickFlix is built on trust!

*Questions:*

1. **How would you fix your code so that this issue is no longer present?**

We would build a check into the code before the video is downloaded from the URL to ensure the content is the expected file type. This is not the strongest protection. Some further protections include storing the file outside of the webroot (ie another server or another domain), assign cryptographically random filenames, checking headers, and serving data with a plaintext MIME type.

2. **How does your test demonstrate SSRF as opposed to just accessing any old endpoint?**

To demonstrate SSRF, we not only verify that there is content at the endpoint, but we also check the content for the expected output. Because we know what command is being executed from our test code, we know the expected output. We parse the output to ensure it is what we anticipated.

**Activity 7:** Command Injection

*Test:* test\_command\_injection.py

Mess around with our video search feature and see what bash commands you can execute from here. You’d be surprised the level of sensitive information you can display on this page.

*Questions:*

1. **How would you fix your code so that this issue is no longer present?**

We would fix this issue by removing the users privilege to execute shell commands, as well as filter the users input to remove “;”. This would prevent any possibility of multi-command queries. We would also search for videos by using the database instead of the directory they’re stored in.