Project 1 Questions

CSEC 380 Team 4

11/24/19

Activity 3-7 Questions

CSEC 380 Team 4: Jacob Ruud, Jack McKenna, Daniel Monteagudo, Dylan Rabasca

# Activity 3: Authentication Questions

1. Provide a link to the test cases you generated for this activity.
   1. <https://github.com/hulto/CSEC380-Team4/tree/master/testing>
2. How do you ensure that users that navigate to the protected pages cannot bypass authentication requirements?
   1. We ensure users that find a protected page are not able to bypass authentication through blocking access to said webpage from standard website users, where as allowing specific admin and maintenance accounts to access the page.
3. How do you protect against session fixation?
   1. By assigning a random value to a session when its initiated, we can authenticate the users session, stopping the attack.
4. How do you ensure that if your database gets stolen passwords aren’t exposed?
   1. Hashing the passwords keeps them from immediate exposure if they are stolen, and adding a salt in will keep them protected from a rainbow table attack on the hash.
5. How do you prevent password brute force?
   1. Brute forcing user passwords can be prevented by a strong CAPTCHA, and an account lockout after a certain amount of login attempts, with a delayed unlock or requiring an admin to manually unlock the account, depending on the nature of the account.
6. How do you prevent username enumeration?
   1. By printing an identical message to a password failure, with all input cleared on reloading the login page, we make it difficult to perform any enumeration on usernames without knowledge of a correct password.
7. What happens if your sessionID is predictable, how do you prevent that?
   1. If our session ID was predictable, malicious users would be able to hijack sessions and performs tasks on behalf of another user. For example, if I hijacked a users session I would be able to upload and more importantly, delete videos on behalf of that user. in order to prevent predictable sessionID generation, we aren’t using sessionID’s. Instead, we are letting flask-login handle our session management. All that’s required from us as developers is a secret key that flask will use to generate session info for each user.

# Activity 4: The Content Questions

1. How do you prevent XSS is this step when displaying the username of the user who uploaded the video?
   1. by using Jinja2 to generate the html of the video owner, we are automatically protected from XSS because by default Flask configures Jinja2 to auto escape all values loaded in the page.
2. How do you ensure that users can’t delete videos that aren’t their own?
   1. we used a combination of frontend and backend techniques to ensure that users are unable to delete videos from the database that don’t belong to them. On the frontend, we used jinja2 to ensure the delete button is only visible to users who are logged in. On the backend, we wrote a helper function to query the database and ensure the user who is trying to delete the video is the user that owns the video, if the user fails this test they are redirected to the home page and the video is not deleted.

# Activity 5: SQL Injection (Classic/Blind) Questions

1. How would you fix your code so that these issues were no longer present?
   1. to fix the sql injection issue, all we would need to do is use the built in session library that flask provides instead of creating the query manually ourselves (which we did at the beginning and had to go back and change)
2. What are the limitations, if any that, of the SQL Injection issues you’ve included?
   1. As mentioned above, because we needed to include SQL injections, we were not able to use flasks built in session library, thus being forced to create the queries ourselves.

# Activity 6: SSRF Questions

1. How would you fix your code so that this issue is no longer present?
   1. The SSRF would be fixed by adding a list of domains and protocols that the server is allowed to interact with. We’d also attempt to block any user input that tries to trick the server in accessing a malicious web server.
2. How does your test demonstrate SSRF as opposed to just accessing any old endpoint?
   1. This is demonstrated through abuse of the SSRF to access a service located internally to our service, and revealing it to the user

# Activity 7: Command Injection Questions

1. How would you fix your code so that this issue is no longer present?
   1. To fix command injection, all we would have to do is specify the file type that can be uploaded to the server, then users would be unable to upload shell scripts instead of videos on the upload page