HW2 Short Answers

Jiahao Dong(jd787) Minghan Tsai(mt627)

Task 1.6

1.

- Because `HTTPonly` is a flag included in a `Set-Cookie` HTTP response header and we only put session token in a hidden form field.
- It implicates that the anti-CSRF token will not be automatically included by the browswer that makes CSRF attacks possible, on the other hand it loss the protection offered by `HTTPonly` against XSS attacks.

2. It won't prevent login CSRF because session token doesn't exist before a successful login (that creates a session) so the same strategy does not work for login forms. After the victim logs in to malicious account they will do all operations in the actually UI and all requests generated will have legit session token. To prevent login CSRF, we can ask the site to generate a token even before authentication as a anti-CSRF token, then all forms including logins include this token as a hidden field.

Task 2.6

- Reflected XSS vulnerability - Attackers can use a similar attack as the one in Task 2.1 and 2.2 to steal the victim's credentials. For example, the coin amount or some personal information shown on the user's profile page. Attackers can use reflected XSS technique to send what has been "seen" on users' logged in page to arbitrary destinations./
- Stored XSS vulnerability - Any entry point on the profile page could be subject to stored XSS attack. Attackers can insert malicious scripts through any `<input>` tag, "pay" button as an example, using the similar method we used in Task 2.5 to contaminate a user's page and force them to send coins to any destination once the page is viewed.

Task 2.8

1. Stored XSS - Malicious scripts were stored directly into the database or server where no sanitation check was performed. When the user visits the compromised website, the script is then executed automatically./ Reflected CSS - The attacker injects malicious code as part of the HTML rendered in a webpage. Only when the user clicks on it, it then becomes effective.
2. When a web server set the Access-Control-Allow-Origin header to "*", it means that the resource in its origin can be accessed by any origin. In this case, any site can send an XHR request to the site and access the server's response on behalf of their visitors. In a CSS attack, attackers may easily exfiltrate user credentials using cookies they gain through user's visit to a malicious website. They don't even need to think of a way to bypass the header.

Task 3.2

1. It is not effective as it only prevent SQL injection through the site's UI and most such attacks will be made through scripts that has nothing to do with JavaScript like what we implemented in task 3.1.

2. Server-side sanitization is able to ensure all user input can be converted to a non-String, like a date, numeric, boolean, enumerated type, etc and check if all input is valid before append it to a query, Prepared Statements allows the database to distinguish between SQL code and data, regardless of what user input is supplied so no user input is able to change the intent of a query. None of the two is clearly better than the other, they both are good countermeasures but has to be done right to be effective.