Maureen Hsu (yh863)
Cealia Pai (yp332)

Short Answer Questions 1.6:

1. The countermeasure you implemented in Exercise 1.5 re-uses the session token as a CSRF token. This prevents the session token from having the “HTTPOnly” attribute.

• Why does re-using the session token in this way prevent it from being marked HTTPOnly?

HttpOnly prevents Javascript from accessing session directly. Re-using the session token means that our server would have to rely on reading session to verify a payment. Therefore, session token could not be marked as HttpOnly.

• What are the security implications of not having HTTPOnly set on a session token?

When it’s not marked as HTTPOnly, other malicious javascript can also read session directly, retrieve essential information and send payment accordingly.

2. In class we discussed login CSRF, where an attacker logs a user into a service (using the attacker’s credentials) without their knowledge. Does the countermeasure you implemented in Exercise 1.5 prevent login CSRF? Why or why not? If not, how would you build a token-based login CSRF countermeasure?

No, the attacker knows his own session id. Therefore, with the attacker’s credentials, the checking mechanism can be bypassed. To counter this problem, we can set some value in the user's cookie that only the user knows and check that value when the user tries to login. This way, even with the attacker's credentials, the login will fail because the value in the user's cookie does not match.

Short Answer Questions 2.6:

1. Identify 1 other reflected XSS vulnerability and 1 other stored XSS vulnerability in app/. Briefly describe how you could exploit them.
    - Reflected XSS: An attacker can put scripts in url to obtain user’s sensitive information like amount of money.
    - Stored XSS: An attacker can inject scripts in dom to make victims send posts to crash the server.

Short Answer Questions 2.8:

1. What is the difference between reflected and stored XSS?
    - Reflected XSS attack is where an attacker stores malicious script in the url and trick a victim to open the url.
    - Stored XSS is where a malicious script is stored in a website, so when a victim visits the webpage, the malicious script executes itself.

2. Cross-origin resource sharing (CORS) is a way to relax the same origin policy and allow scripts from other origins to access some site resources via XMLHTTPRequest. To

Maureen Hsu (yh863)
Cealia Pai (yp332)

implement CORS, the web server uses the header “Access-ControlAllow-Origin” to tell the browser which origins can access its resources. Describe what attacks can arise if a web server set this header’s value to “*” (meaning any origin can access it).

A malicious third-party can redirect a victim to the vulnerable web server with a malicious js script, and trigger XSS to obtain sensitive data.

Short Answer Questions 3.2:

1. Imagine a SQL injection countermeasure that used client-side Javascript to preprocess user input and remove strings that looked like SQL injection attacks before sending the input to the server. Is this an effective SQL injection countermeasure? Why or why not?

No, because it is almost impossible to filter characters that might cause SQL injection and too much filtering might prevent users from using some characters (like O’Brien). Plus, the attacker can use ‘executejs’ in sql to make it executes javascript

2. Compare and contrast server-side sanitization (for example, removing SQL comment characters) and prepared SQL statements as SQL injection countermeasures. Is one clearly better than the other?

Prepared SQL statements are better because they can prevent malicious input to change the meaning of the queries since most of the attacks try to trick applications into interpreting data as code.