CS 5435 – Security and Privacy Concepts in the Wild

Homework 2

Name: Arystan Tatishev(at855) , Nikhil Reddy(nc463) *Spring 2024*

Exercise 1.6

1. **HTTPOnly**

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

     Re-using the session token as a CSRF token prevents it from being marked as HTTPOnly because the CSRF token needs to be accessible by client-side scripts to include it in requests. If the token were marked as HTTPOnly, it would prevent client-side scripts from accessing it, thus making it unusable for CSRF protection. The HTTPOnly attribute is a security measure that prevents client-side scripts (such as JavaScript) from accessing the cookie via the document.cookie API.

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

     When a session token lacks the HTTPOnly attribute, it becomes vulnerable to XSS attacks. If an attacker successfully injects a malicious script into the web application, that script could potentially read the session token from the cookie and send it to the attacker-controlled server, allowing the attacker to hijack the user's session.

2. **Login CSRF**
   The countermeasure of using a session_id as a token alone does not fully prevent login CSRF. While session_id can help mitigate some attacks, it is not sufficient to prevent login CSRF because session IDs can still be obtained by attackers if they have access to the HTML or URL of the page. Additionally, if the session-independent nonce is used, it can be overwritten by subdomains or network attackers. If a hash of the userID is used, it can also be forged by attackers.

   To build a more robust token-based login CSRF countermeasure, you can consider the following approach:

   (a) **Add Secret Token to Forms**: Include a secret token in all forms related to authentication. This token serves as an additional layer of protection against CSRF attacks.

   (b) **Bind Session ID to the Token**: Ensure that the session ID is securely bound to the token. One approach is to use a keyed HMAC (Hash-based Message Authentication Code) of the session ID along with the secret token. This creates a unique and verifiable token for each session, without the need for additional state management on the server.

(c) **Server-side State Management**: Use server-side state management techniques like CSRFx or CSRFGuard to manage the state table at the server. This helps in validating the authenticity of the token and session ID combination.

By implementing these measures together, you can create a more robust token-based login CSRF countermeasure. This approach adds an extra layer of security by ensuring that requests are authenticated not only based on the session ID but also on a unique token that is securely bound to the session.

Exercise 2.6

- **1 other reflected XSS vulnerability in** *app/*

  During registration in login.py, the new registered users' usernames are not sanitized. This means that we can inject malicious code as usernames and have that go directly into the database

- **1 other stored XSS vulnerability in** *app/*

  The user's username is not sanitized in profile.html when displaying user's username on the page. We can implement a worm that displays victim's username but underneath is a hidden worm script.

Exercise 2.8

1. **What is the difference between reflected and stored XSS?**
   Reflected XSS promptly injects and runs malicious scripts in a web app's response via user inputs like URLs or forms. Stored XSS stores the harmful script on the server, usually in a database, executing it when certain conditions are met, such as when other users access the compromised content.
2. **Describe what attacks can arise if a web server set this header's value to "*" (meaning any origin can access it).**
   Setting the `Access-Control-Allow-Origin` header to `*` permits any origin to access server resources, which can lead to several security risks:

   (a) **CSRF**: Attackers can forge requests from their domains, exploiting user trust.
   (b) **Information Leakage**: Sensitive data may be exposed to unauthorized parties.
   (c) **XSS**: Cross-origin scripts can exploit vulnerabilities, accessing server resources.
   (d) **Data Exfiltration**: Attackers can steal sensitive information from the server.
   (e) **Resource Abuse**: Unlimited access may result in bandwidth consumption and server overload.

   In essence, a wide-open CORS policy invites these risks and should be used judiciously, favoring restricted access to trusted origins.

Exercise 3.2

1. **A SQL injection countermeasure**

   Client-side input validation offers the benefit of saving network round trips that would otherwise be required to reject invalid input. This can lead to performance improvements by reducing server load and providing immediate feedback to the user. However, there's a problem with relying solely on client-side validation: the client is untrusted.

   Since client-side code can be manipulated or bypassed by an attacker, any validation performed on the client side cannot be relied upon for security. The server must assume that all client input could be malicious and must re-validate and sanitize all input on the server side. This is essential for protecting against many types of attacks, including SQL injection, where attackers can exploit vulnerabilities in the way a server processes input to execute arbitrary SQL commands.

2. **Compare and contrast server-side sanitization and prepared SQL statements**

   - Prepared statements are more secure than server-side sanitization, as they are designed to prevent the execution of untrusted input as part of the SQL statement inherently.
   - Sanitization can allow SQL injection through if not comprehensive, whereas prepared statements correctly handle all inputs when used properly.
   - Sanitization can lead to false positives or negatives, whereas prepared statements reliably manage input.
   - Prepared statements also improve code clarity by separating SQL logic from data.

   While server-side sanitization can be part of a multi-layered security strategy, prepared SQL statements are a more robust and effective method for preventing SQL injection. They do not rely on pattern matching or character escaping and provide a built-in mechanism for safely handling data in SQL queries.