CS 5435 – Computer Security

Homework 2

Name: David Kogan (dk448), Andrew Palmer (ajp294) *Fall 2019*

Exercise 1.6

1. When we reuse the session token as the anti-csrf token, we do so by accessing the session token via client-side scripting. This allows us to read the session token and place it on the form as the anti-csrf hidden token value. When a cookie is marked HTTPOnly, it means that it cannot be accessed by any client-side script. Therefore, we would not be able to use the session token as the anti-csrf token in our script if it were marked HTTPOnly.

   By not having the session token marked as HTTPOnly, we give access to the token through any client-side script in the origin. As such, the token becomes available to potential malicious users to use in a CSRF attack or a reflected XSS attack.

2. Our anti-csrf countermeasure does not protect against login CSRF. As we included the anticsrf-token only on the pay form, the login form is still vulnerable. With login CSRF, the attacker can force the user to login to a website using the attacker's account. Once logged in, the attacker can view browsing information about the victim, since it will be through the attacker's credentials.

   To protect against this, we can implement a login csrf token as such. By providing an anti csrf token on the login form, the web server can verify that it receives that token back when processing a login request. Therefore, when the attacker attempts to login with his credentials through the victim, the login web server will detect the missing anti csrf token and deny the login. As the attacker cannot know the anti csrf token beforehand, this protects the victim from the login csrf attack.

Exercise 2.6
Reflected: Looking at the login form functionality, we see that when an unregistered user attempts to login, the web server returns an error message along with the 401 response. This error message is reflected to the user on the login page. By crafting a malicious URL, an attacker can trick a user into sending a post request via the login form, which will get denied and reflected back. An attacker can use this vulnerability in the login form to execute arbitrary html and Javascript.

To test this vulnerability, we tried to login with an unregistered user. As a first test, we logged in with the html <b>Hello</b> and verified that the error message had Hello rendered in bold. Additionally, we tested logging in with the unregistered user given by <script>alert(' reflected ');</script> and verified the browser produced the alert pop up when the error was returned.

Stored: A stored XSS vulnerability, like the About Me worm created in the exercises, allows for permanent injection of a malicious script. Thus, to located an additional persistent XSS vulnerabilty, we looked for other areas of app that allow a user to store data on the web server. The first place a user comes across this ability is actually before even logging into an account. With the Register functionality, a user can store a malicious script embedded in the username field of the login form. Once the malicious script is stored, a victim user can be tricked into visiting that new user's malicious page and execute the script when the web server loads the username field.

We tested very basic functionality of this stored XSS vulnerability by creating a username with the username <script>alert('test');</script> and verified that the browser generated an alert popup when the infected profile, which was previously with the malicious script, was visited by another user.

Exercise 2.8

1. Some major differences between reflected and stored XSS is where the malicious script lives and how it gets executed by the victim user. In a stored XSS attack, the malicious script is injected into a web server in a place where the web server will take the script and store it in its database. With HTML5, the script can also be stored in the victim's browser, and never sent to an actual server[2]. In order to execute the injected script, the user must visit a URL that causes the web server to retrieve the script from its database and render it to the user, thus executing it.

   In a reflected XSS attack, unlike the stored XSS attack, the malicious script does not need to be stored to or retrieved from a database by the web server. A reflected attack can occur when user input is returned by a web application with the results rendered by that web server. For example, an error message or search result will render back the user input, which can be abused in a reflected XSS attack.

2. By using CORS and allowing scripts from other origins to access site resources, care must be taken not to allow bad actors access to resources they shouldn't. When the wildcard $*$ is used to let any origin can access the resources, we open up attacks that would have normally be blocked by the same-origin policy. Now, a malicious user, in a different origin than the victim, can send requests to a web server for specific resources. The attacker's Javascript can read and write to resources belonging to a different origin. Therefore, a server that responds with a $*$ wildcard for Access$-$Control$-$Allow$-$Origin can be a large security risk and should only be used if the application absolutely requries it, such as with a public API[1].

Exercise 3.2

1. Client-side Javascript pre-processing of an input string to prevent SQL injection attacks is a potential, but incomplete defensive measure. As there are many potential SQL injection vectors in a SQL command string, the logic for pre-processing the string would be logically complicated, which gives a higher chance of defects. Additionally, too strict of pre-processing may block benign users and requests, which reduces the sites usability.

   Additionally, a major flaw of client-side pre-processing is that many attacks will not use a client at all for the attempted hack. A malicious user can script the SQL injection attacks, sending them directly to the server, and completely bypass the client pre-processing. Because of this, client-side Javascript pre-processing is not an effective SQL injection countermeasure.

2. Server-side sanitization and prepared SQL statements are both server-side defensive measures against SQL injection attacks. However, they differ in effectiveness and implementation. With server-side sanitization, developers are responsible for defining rules to reject inputs to forms that look suspicious. For example, if only numbers are expected, they could reject any input that contains alphabetic characters. Like with client-side pre-processing, defining and detecting suspicious inputs is not an exact science and reduces the effectiveness if this countermeasure. Additionally, analyzing inputs can be time consuming and resource intensive.

   Prepared SQL statements, in contrast to server-side sanitization, does not attempt any analysis or interpretation of the given inputs. Instead, a parameterized SQL string and its parameters are passed to the SQL driver separately. This allows the driver to interpret the inputs correctly[3]. In essence, the command parameters are never treated like SQL commands and instead are treated as data. Since prepared SQL statements separate SQL commands from parameters, do not suffer from gaps in input analysis, and are more efficient, they are a better choice of countermeasure for SQL injection attacks.

References

[1] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[2] https://www.owasp.org/index.php/Types_of_Cross-Site_Scripting

[3] https://www.hacksplaining.com/prevention/sql-injection