Zachary Panzarino
zpanzarino3

Target 1

1. account.php line 23
2. Essentially the way that the XSRF prevention mechanism works is with a hash that is calculated on both the front and back end and compared to ensure that only submissions from the correct website are accepted. The hash is calculated by using the account number, routing number, and challenge token. In theory, an attacker would not be able to calculate the correct hash because they would not know the challenge token for a user. However, the backend is vulnerable since it uses a user-provided challenge token to calculate the verification hash. Essentially, it trusts that the challenge token provided by the user is correct. Since the hash function is on the public JavaScript of the page, an attacker can calculate the hash for any account number, routing number, and challenge token. Therefore, the attacker can set the account number and routing number to be their bank account and the challenge to be any arbitrary string. With these values, they can use the public JavaScript to calculate the hash for these values that will pass the server-side verification. By mocking the rest of the values in the form, the attacker can successfully send a request to the server which will modify the account and routing numbers.
3. Currently the vulnerable line reads as follows:
   `$teststr = $_POST['account'].$_POST['challenge'].$_POST['routing'];`
   One way to fix the vulnerability is to change this line to the following:
   `$teststr = $_POST['account'].$_SESSION['csrf_token'].POST['routing'];`
   Since the correct challenge value is stored as a session variable, it should use that rather than the user-provided one when calculating the verification hash. This ensures that the request has to use the correct challenge value for that user that is stored on the server, which an attacker would not be able to figure out without being logged into the user's account.

Target 2

1. index.php line 34
2. The vulnerable line is used to pre-fill the username field in the case that the previous login attempt was unsuccessful in order to make it easier for the user to input anther password. However, this input is pushed out to the user without being sanitized first, allowing an attacker to have anything desired printed out to the screen. In order to exploit this, an attacker can make a post request to the page with a special value for the username, and the page will output that value in the response. Therefore, an attack can end the input element and set custom html elements such as script tags. In this exploit, the custom script tag adds JavaScript which binds the form submit to a function. That function requests an "image" from hacker mail with the values of the username and password fields, effectively sending out an email with the login information. After that email is sent, the page continues with the action of the form, submitting as usual.
3. In order to fix this vulnerability, the username must be sanitized before being output. One potential way of combating this is through the PHP functions `htmlspecialchars` or `htmlentities`. These functions convert any characters that have special significance in HTML to HTML character codes. Therefore, any HTML code is no longer active when output and will be displayed exactly as inputted.

Target 3

1. auth.php lines 28-69
2. This script is vulnerable since it uses queries without properly escaping user input. First, it does not remove any single quotes, allowing user input to end strings defined in the query. In addition, it only checks for certain uppercase SQL keywords, so passing these keywords in as lowercase slips by. Essentially to execute this attack, an attacker can end the string containing the username, then add in SQL statements such as `or` to add extra conditions to the query. Finally, if the end of the input contains an unclosed string, it will be properly closed when put into the query, making a complete statement. Therefore, when an `or` is added to the condition with a true value, it will authenticate the user without checking if the password matches.
3. In order to fix this vulnerability, all user input into SQL statements should be properly filtered. The PHP function `mysqli_real_escape_string` is a correct implementation of the `sqli_filter` function implemented in this instance. This function will properly remove any input that could affect an SQL statement. While this works, the best way to filter input is through prepared statements. The PHP `mysqli` interface provides a series of methods that properly bind user-inputted variables into an SQL statement, ensuring that user input cannot cause an SQL injection attack.