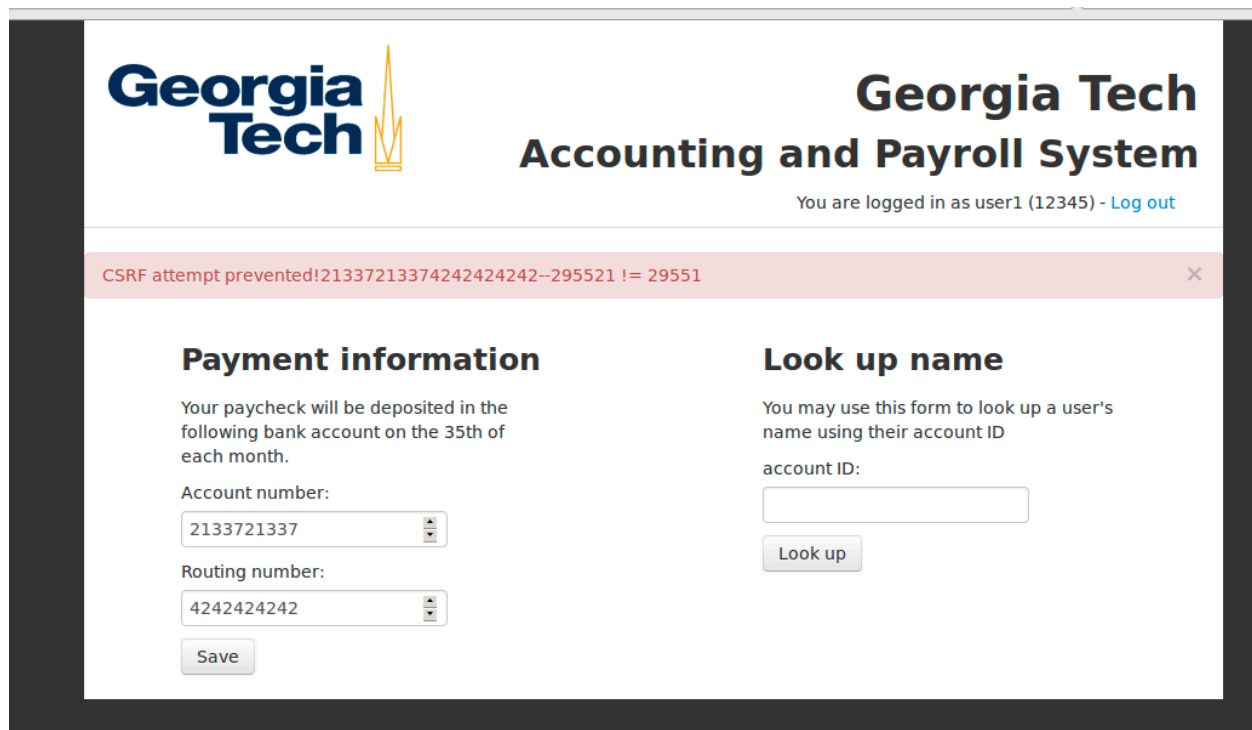


Task 1 – XSRF

The vulnerable code is in account.php: 21 because the '\$expected' response value that I pass in my request is static and is always the same value. Also, due to the fact that the HTML page tells you what the expected value is (i.e. – if you provide the incorrect response value, the page updates the DOM with a message telling you what the expected response value is – See 'Figure A' below) Therefore, once we know the expected value of 29551, we simply update the response value in our POST.

Figure A



The screenshot displays the Georgia Tech Accounting and Payroll System interface. At the top, the Georgia Tech logo is on the left, and the text "Georgia Tech Accounting and Payroll System" is on the right. Below the title, it says "You are logged in as user1 (12345) - [Log out](#)". A red error message banner at the top reads: "CSRF attempt prevented!21337213374242424242--295521 != 29551". The main content area is divided into two columns. The left column, titled "Payment information", contains text about paycheck deposits and input fields for "Account number" (2133721337) and "Routing number" (4242424242), with a "Save" button. The right column, titled "Look up name", contains text about looking up a user's name and an input field for "account ID", with a "Look up" button.

Task 2 – XSS-password theft

The vulnerable code is in index.php:29 which allows for arbitrary code to be entered. The attack I crafted inserts a single tic's (') into the input box. From there, we are able to insert malicious javascript that takes the user's input for login and password and forwards them to the hackmail.org application. To fix this, the user should be performing input handling on all text boxes to prevent XSS.

Task 3 – SQL Injection

The vulnerable code is in index.php:14

The code is vulnerable due to the fact that index.php allows single tick ' characters to be inserted into the text boxes, allowing for a second order SQL injection. In this particular type of SQL injection, we use:

`' OR '1'='1`

which forces the selection of every data field stored in the database, rather than for one single user name as the application expects, evaluating `1=1` which is always true.

Epilogue

Task 1

To fix the vulnerability in Task 1, a simple could be, when a user initially signs in to the application, they are given a randomized token that only they and the application know. On future requests / posts to the application, the application will require that the user present that token on each of their requests / posts. The token will change each time the user logs in and logs out and will have a set time value before it is no longer valid.

Task 2

The vulnerability in Task 2 could easily be patched with input handling performed on all text entry fields, on the client side. It is a general rule of thumb that sanitization of text user input should be handled at the client rather than at the server where the server could potentially be exploited. The input handling that should be employed on this particular should prevent single-tick marks or encode them to where they cannot be interpreted simply as text.

Task 3

Similar to the previous answer, better input handling, and also parameterized queries which allows users to enter whatever type of text strings they like. Prepared statements also help to prevent sql injection attacks since this makes sure nothing could be passed to the application to modify the SQL query (such as in the attack we performed in which we added a 'WHERE 1=1' clause)