Group 6

**Part 1: Warmup Examples**

Example 1:

Command Execution error: userinput reaches a sensitive sink

When the code does $\_GET[‘filename’] in line 4 there is no checking what the file type is or any other checks that stop this file from ‘executing any arbitrary system commands’. It is fixed by initializing the array with some constraints and making sure the file abides by those constraints in an if statement.

This system command executing function has been fixed so there are no errors.

Example 2:

Cross-Site Scripting error: Userinput reaches a sensitive sink

This vulnerability allows attackers to execute code on clients’ browsers, changing or modifying the embedded code. This can lead to hijacking for the website and should be fixed by encoding all user tainted data with PHP building functions before embedding the data into the output. This is done by setting ENT\_QUOTES to stop eventhandler injections and it specifies a charset.

We fixed this by adding the metaset rules.

Example 3:

File inclusion: userinput reaches sensitive sink

An attacker can use PHP or read from non-PHP files locally or remotely, these files will be read by this code and cause non-PHP code to be embedded in the output. We should build a whitelist to limit possible file names that we want, we shouldn’t constrain by extensions only.

Fixed

Example 4:

SQL Injection Userinput reaches sensitive sink

With this vulnerability SQL commands can be executed on our server by injecting SQL syntax into the query in our code. The fix is to anticipate the result of the query and constrain results based on anticipating that query. We should embed expected integers without and redefine the $query variable to remove vulnerabilities for SQL injection.

**Part 2**

Htaccess done

**Part 3**

The majority of errors are cross-site scripting and SQL injection. Of the other errors we had to research there was a lot of weak typing.

|  |  |  |
| --- | --- | --- |
| **Vulnerability Name** | **Location** | **Proposed fix\*** |
| **SQL Injection** | **\code\View/register\_user.php**  **Line 28** | **There are no escape characters so tampering with the inputs may occur. Add escape statements to fix vulnerabilities.** |
| **Cross-Site Scripting\*** | **\code\View/book\_details.php**  **Line 67** | **Secure this vulnerability by specifying a metachar set and encode them before the output.**  **This fix is similar to the cross-site scripting fixes below.** |
| **Cross-Site Scripting\*** | **\code\View/book\_details.php**  **Line 78** | **Secure this vulnerability by specifying a metachar set and encode them before the output.**  **Secure this vulnerability by specifying a metachar set and encode them before the output.** |
| **Cross-Site Scripting\*** | **\code\View/book\_details.php**  **Line 89** | **Secure this vulnerability by specifying a metachar set and encode them before the output.**  **Secure this vulnerability by specifying a metachar set and encode them before the output.** |
| **Cross-Site Scripting\*** | **\code\View/book\_details.php**  **Line 100** | **Secure this vulnerability by specifying a metachar set and encode them before the output.**  **Secure this vulnerability by specifying a metachar set and encode them before the output.** |
| **Cross-Site Scripting** | **\code\View/book\_details.php**  **Line 110** | **Secure this vulnerability by specifying a metachar set and encode them before the output.**  **Secure this vulnerability by specifying a metachar set and encode them before the output.** |
| **Cross-Site Scripting** | **\code\View/book\_details.php**  **Line 120** | **Secure this vulnerability by specifying a metachar set and encode them before the output.**  **Secure this vulnerability by specifying a metachar set and encode them before the output.** |
| **Cross-Site Scripting** | **\code\View/book\_details.php**  **Line 137** | **Secure this vulnerability by specifying a metachar set and encode them before the output.**  **Secure this vulnerability by specifying a metachar set and encode them before the output.** |
| **Weak typing : fetch\_array** | **\code\Model\library.php**  **Line 38, 104, 107**  **\code\model\shelf.php**  **Line 20, 23, 25, 32** | **Using built-in weakly written comparable array functions like fetch\_array lead to vulnerabilities. Use direct, well-written comparisons instead.** |
| **Weak typing:**  **== vs ===\*** | **\code\Model\library.php**  **Line 117** | **== is an implicit type conversion; === is more secure and is better practice. Use === instead.** |
| **Web Tampering** | **\code\Database\db\_connect.php** | **This file has information to access a server and where to access it. This file should be encrypted.** |
| **Unicode encoding** | **None?** | **None?** |
| **SQL Injection\*** | **\code\Database/register\_user.php**  **Line 12** | **There are no escape statements for the user inputs so tampering may occur.**  **Even though this is fixed it appears as a false negative on RIPS because the code is object oriented.** |
| **SQL Injection\*** | **\code\Database/register\_user.php**  **Line 30** | **Same as above, no escape statements for the user inputs so we added these. False negative still occurs.** |
| **SQL Injection** | **\code\Database/delete\_book.php**  **Line 4**  **(multiple errors x5 – see right)** | **All of these errors originate from line 4 where a user input that maybe tampered with is passed through the function parameters. The error messages point to vulnerabilities in line 128, 90, 125, and 126 in \code\model\library.php. Thus, all of these lines will need to be changed by embedding strings explicitly and escaping the string.** |

**\*Designates that these were fixed in the code.**

**Part 4**

To log in we came up with a statement that is always true under the parameters that form the variable $sql.

**Username: anything’ OR ‘x’=’x**

**Password: anything’ OR ’x’=’x**

These two-component parameters are always guaranteed to be true so that a username and password entry is taken at random when the query is made.

We can fix this problem by changing how we query the username by making sure it does not point to anything related to SQL statement parsing. We do this by making username point to a positional parameter #1 and password to #2. We then set the given username and password later and then run our query.

This is called a prepare statement.

**<?php**$username=$\_GET[***user***];  
$password=$\_GET[***pwd***];  
$sql=**"SELECT \* FROM usertable WHERE username= ? AND password = ?"**;  
***sql***.setString(1,***username***, 2, ***password***);  
$result=$db->query($sql);  
**if**($result->**num\_rows**=== 1){*/\*(successful login\*/*} **else**{*/\*(login failed\*/*}  
**?>**