Inventory Management System Report

Evidence Monday

June 29, 2020

1 Problem Description

As an application security engineer, I was assigned to do vulnerability testing on an inventory management system web application.

2 Procedure

The scope of coverage involves using a novel tool called *Pixy* to automatically test for vulnerabilities in different pages that make up this applications. The tool, *Pixy* automaticlaly detects potential xss attacks in each web pages but it has an overtainting limitation thereby leading to some false positives. This is where manual testing comes in. Since *Pixy* output is in `.dot`, there is need to first convert each results to images which can be viewed then go through each images and doing a static taint analysis on the corresponding code shown in the resulting image. After which for every true positives found, it is important to write a automation script for proof of concept using Selenium with Java. Because, converting each image will be tedious, I wrote a simple Python code to automate the process. The tools used in this project are Pixy, Selenium, Pyhton, Java, IntelliJ IDE.
The steps in simple terms

1. Convert `.dot` files to images using python script
2. Do static taint analysis on each corresponding code.
3. Write a automation script for true positives showing for proof of concept purpose using Selenium with Java.
4. Fix code and check that code is not longer vulnerable

3 Results

Below shows the details of each images and code analyzed:

1. **False positive:** the variable `$valid` in `changeBio.php` is coming from the system not a user input.

2. **False positive:** the variable `$valid` in `changePasssword.php` echoed on line 43 is coming from the system not a user input.

3. **False positive:** the variable `$valid` in `changeUsername.php` echoed on line 23 is coming from the system not a user input.

4. **False positive:** the variable `$valid` in `createBrand.php` echoed on line 25 is coming from the system not a user input.

5. **False positive:** the variable `$valid` in `createCategories.php` echoed on line 25 is coming from the system not a user input hence an xss is not possible here.

6. **False positive:** the variable `$valid` in `createOrder.php` echoed on line 70 is coming from the system not a user input hence an xss is not possible here.

7. **False positive:** the variable `$valid` in `createProduct.php` echoed on line 43 is coming from the system not a user input hence an xss is not possible here.

8. **False positive:** the variable `$valid` in `createUser.php` echoed on line 30 is coming from the system not a user input hence an xss is not possible here.

9. **False positive:** the variable `$countProduct` in `dashboard.php` being echoed on line 75 to the dashboard is an integer and cannot contain xss payload.

10. **False positive:** the variable `$countLowStock` in `dashboard.php` being echoed to the dashboard is an integer and cannot contain xss payload

11. **False positive:** the variable `$countOrder` in `dashboard.php` being echoed to the dashboard is an integer and cannot contain xss payload.

12. **True positive:** the variable `$username` echoed on the dashboard is vulnerable and can be exploited by injecting `<h1>admin</h1>` in the username field in `settings.php`

13. **False positive:** the variable `$totalorder` echoed on the dashboard cannot be exploited because it is the result of arithmetic operation on the discount field and result would be an integer which cannot contain an xss payload.

14. **False positive:** the variable `$valid` echoed on line 25 in the `editBrand.php` page is coming from the system not a user input.

15. **False positive:** the variable `$valid` echoed on line 25 in the `editCategories.php` page is coming from the system not a user input.

16. **False positive:** the variable `$valid` echoed on line 87 in the `editOrder.php` page is coming from the system not a user input.

17. **False positive:** the variable `$valid` echoed on line 31 in the `editPayment.php` page is coming from the system not a user input.

18. **False positive:** the variable `$valid` echoed on line 21 in the `editProduct.php` page is coming from the system not a user input.

19. **False positive:** the variable `$valid` echoed on line 35 in the `editProductImage.php` page is coming from the system not a user input.

20. **False positive:** the variable `$valid` echoed on line 27 in the `editUser.php` page is coming from the system not a user input.

21. **True positive:** the sink statement in `fetchBrand.php` at line 48 echoes the `$output` variable which contains input from the user `$BrandName` and not sanitized hence, making the `$output` variable vulnerable to xss attack. A possible attack vector is `<h1>Apple</h1>`

22. **True positive:** the sink statement in `fetchCategories.php` at line 48 echoes the `$output` variable which contains input from the user `$CategoryName` and not sanitized hence, making the `$output` variable vulnerable to xss attack. A possible attack vector is `<h1><b>SmartPhone</b></h1>`

23. **True positive:** the sink statement in `fetchOrder.php` at line 71 echoes the `$output` variable which contains input from the user `$ClientName` in `EditOrder.php` and not sanitized hence, making the `$output` variable vulnerable to xss attack. A possible attack vector is `<h1>Smith</h1>`

24. **False positive:** The user input, `$orderId` is used in the sql query which in fact cannot be vulnerable to xss because the query expects an integer which cannot be used for xss attack. Hence, the variable `$valid` (having entities of the sql query) echoed in `fetchOrderData.php` on line 19 is safe because the query itself is not vulnerable to an xss attack.

25. **True positive:** : The variable `$productName` from `editProduct.php` which is all vulnerable are entities of the variable `output` which is echoed on line 83 in `fetchProduct.php`

26. **False positive:** There is no user input used in the variable `$data` echoed on line 12 in `fetchProductData.php`

27. **False positive:** The user input, `$productId` is used in the sql query which in fact cannot be vulnerable to xss because the query expects an integer which cannot be used for xss attack. Hence, the variable `$result` (having entities of the sql query) echoed on line 13 is safe because the query itself is not vulnerable to an xss attack.

28. **False positive:** the variable, `$row` echoed on line 16 in `fetchselectedBrand.php` comes from the sql query which only accepts integers and cannot contain html scripts, hence is not vulnerable to xss attack.

29. **False positive:** the variable, `$row` echoed on line 16 in `fetchselectedCategories.php` comes from the sql query which only accepts integers and cannot contain html scripts, hence is not vulnerable to xss attack.

30. **False positive:** the variable, `$row` echoed on line 16 in `fetchselectedProduct.php` comes from the sql query which only accepts integers and cannot contain html scripts, hence is not vulnerable to xss attack.

31. **False positive:** : the variable, `$row` echoed on line 16 in `fetchselectedUser.php` comes from the sql query which only accepts integers and cannot contain html scripts, hence is not vulnerable to xss attack.

32. **True positive:** the `$username` variable which is a part of the output variable echoed on line 47 in the `fetchUser.php` page is coming from the `editUser.php` page is vulnerable to xss. A possible attack vector is `<h1>user1</h1>`

33. **False positive:** The result `$variable` on line 30 is coming from the sql query from line 17 which is not vulnerable because it does not contain user input in fact, it only contains calendar input which cannot be tainted. Hence, the `$table` variable echoed on line 49 in the `getOrderReport.php` page is safe.

34. **False positive:** False positive: Here, the `$_SERVER['PHP_SELF']` function Returns the filename of the currently executing script which in this case is `index.php`. The user has no control over this function whatsoever hence, an xss attack is impossible.

35. **False positive:** `$_GET['i']` the `$i` here is gotten from the system and not a user input hence, the echo statement on line 37 in the `order.php` page is safe.

36. **False positive:** the `$result` echoed at line 111 is provided by the system. There are no user input here hence the echo statement on line 111 in the `orders.php` page is not vulnerable to xss.

37. **False positive:** the `$data[1]` echoed on line 287 has no input from the user, in fact it refers to a date item which is system generated hence `$data[1]` echoed on line 287 in the `orders.php` page is safe and not vulnerable to xss attack

38. **True positive:** `$data[2]` echoed on line 293 contains the `$clientName` variable from the `editOrders.php` page which is vulnerable to xss hence `$data[2]` echoed on line 293 in the `orders.php` page is also vulnerable. A possible attack vector here is `<h1><b>Smith</b></h1>`

39. **True positive:** `$data[3]` echoed on line 299 contains the `$clientContact` variable from the `editOrders.php` page which is vulnerable to xss hence `$data[3]` echoed on line 293 in the `orders.php` page is also vulnerable. A possible attack vector here is `<h1>+38976663</h1>`

40. **False positive:** the items of `$ProductData` echoed at line 345 are provided by the system. There are no user input associated with any items hence, the echo statement on line 345 in the `orders.php` page is not vulnerable to xss.

41. **False positive:** the `$orderItemData['rate']` echoed on 353 in the `orders.php` page contains the value rate which is not tainted because it is not a user provided input hence, it is not vulnerable to xss.

42. **False positive:** the echo `$orderItemData['rate']` echoed on 354 in the `orders.php` page contains the value `$rateValue` which is not tainted because it is not a user provided input hence, it is not vulnerable to xss.

43. **False positive:** the item, product_id of `$ProductData` echoed at line 365 is provided by the system and available_quantity is not tainted, it only accepts integer values hence, the echo statement on line 365 in the `orders.php` page is not vulnerable to xss.

44. **False positive:** the `$orderItemData['quantity']` echoed on line 380 only accepts integer values, `$x` represents the row number which of course is an integer hence, the echo statement on line 380 in the `orders.php` page is not vulnerable to xss.

45. **False positive:** the `$orderItemData['total']` echoed on line 384 is the result of an arithmetic operation which contains integer values hence, the echo statement on line 384 in the `orders.php` page is not vulnerable to xss.

46. **False positive:** the `$orderItemData['total']` echoed on line 385 is the result of an arithmetic operation which contains integer values hence, the echo statement on line 385 in the `orders.php` page is not vulnerable to xss.

47. **False positive:** the `$data[4]` echoed on line 404 is the result of an arithmetic operation called `$subTotal` which contains integer values hence, the echo statement on line 404 in the `orders.php` page is not vulnerable to xss.

48. **False positive:** the `$data[4]` echoed on line 405 is the result of an arithmetic operation called `$subTotalValue` which contains integer values hence, the echo statement on line 405 in the `orders.php` page is not vulnerable to xss.

49. **False positive:** the `$data[6]` echoed on line 412 is the result of an arithmetic operation called `$totalAmount` which contains integer values hence, the echo statement on line 412 in the `orders.php` page is not vulnerable to xss.

50. **False positive:** the `$data[6]` echoed on line 413 is the result of an arithmetic operation called `$totalAmountValue` which contains integer values hence, the echo statement on line 413 in the `orders.php` page is not vulnerable to xss.

51. **False positive:** the `$data[7]` echoed on line 419 is a user input called `$discount` which accepts only integer values and html script is not accepted hence, the echo statement on line 419 in the `orders.php` page is not vulnerable to xss.

52. **False positive:** the `$data[8]` echoed on line 425 is the result of an arithmetic operation called `$grandTotal` which contains integer values hence, the echo statement on line 425 in the `orders.php` page is not vulnerable to xss.

53. **False positive:** the `$data[8]` echoed on line 426 is the result of an arithmetic operation called `$grandTotalValue` which contains integer values hence, the echo statement on line 426 in the `orders.php` page is not vulnerable to xss.

54. **False positive:** the `$data[5]` echoed on line 432 is a variable called vat which is being used for the arithmetic operations producing `$grandTotal`, `$TotalAmount`, `$GST values`. So since it contains only integer values, the echo statement on line 432 in the `orders.php` page is not vulnerable to xss.

55. **False positive:** the `$data[5]` echoed on line 433 is a variable called `$vatValue` which is being used for the arithmetic operations producing `$grandTotal`, `$TotalAmount`, `$GST values`. So since it contains only integer values, the echo statement on line 433 in the `orders.php` page is not vulnerable to xss.

56. **False positive:** the `$data[14]` echoed on line 439 is a variable called `$gstIN` which accepts only integer values hence, the echo statement on line 439 in the `orders.php` page is not vulnerable to xss.

57. **False positive:** the `$data[9]` echoed on line 448 is a variable called `$amountPaid` which accepts only integer values hence, the echo statement on line 448 in the `orders.php` page is not vulnerable to xss.

58. **False positive:** the `$data[10]` echoed on line 454 is the result of an arithmetic operation called `$due` which contains integer values hence, the echo statement on line 454 in the `orders.php` page is not vulnerable to xss.

59. **False positive:** the `$data[10]` echoed on line 455 is the result of an arithmetic operation called `$dueValue` which contains integer values hence, the echo statement on line 455in the `orders.php` page is not vulnerable to xss.

60. **False positive:** `$_GET['i']` the i here in the `orders.php` page is gotten from the system and not a user input hence, the echo statement on line 513 is safe.

61. **True positive:** The variable `$table` echoed on line 193 contains the vulnerable fields from `editOrders.php` is reflected in the `printOrders.php page` hence, this code is vulnerable to xss attack.

62. **False positive:** the variable `$row[1]` which is `$BrandName` echoed on line 109 is not vulnerable because even if it is exploited on the `editbrand.php` page, it is not reflected when echoed on the `addproduct.php page`.

63. **False positive:** the variable `$row[1]` which is `$categoryName` echoed on line 128 is not vulnerable because even if it is exploited on the `editbrand.php` page, it is not reflected when echoed on the `addproduct.php` page.

64. **False positive:** the variable `$row[1]` which is `$BrandName` echoed on line 267 is not vulnerable because even if it is exploited on the `editbrand.php` page, it is not reflected when echoed on the `editProduct.php` page.

65. **False positive:** the variable `$row[1]` which is `$categoryName` echoed on line 286 is not vulnerable because even if it is exploited on the `editbrand.php` page, it is not reflected when echoed on the `editproduct.php` page.

66. **False positive:** the variable `$valid` echoed on line 24 is system generated hence, an xss attack is not possible in `removeBrand.php` page

67. **False positive:** the variable `$valid` echoed on line 24 is system generated hence, an xss attack is not possible in `removeCategory.php` page.

68. **False positive:** the variable `$valid echoed` on line 26 is system generated hence, an xss attack is not possible in `removeOder.php` page.

69. **False positive:** the variable `$valid` echoed on line 24 is system generated hence, an xss attack is not possible in `removeProduct.php` page.

70. **False positive:** the variable `$valid` echoed on line 24 is system generated hence, an xss attack is not possible in `removeUser.php` page.

71. **True positive:** the variable `$username` echoed on line 35 is user input and can be exploited using the attack vector `<h1>admin</h1>`. Because the session remains active after inserting the payload the effect will not be seen till after the user logs in and out to start a new session. Then the exploit will be seen on the `dashboard.php` page

72. **False positive:** the variable `$user_id` echoed on line 41 is of hidden type and the user cannot see it on the `settings.php` page hence, an xss attack is not possible here.

73. **False positive:** all occurrence of the `$bio` variable are just text areas and were not particularly echoed on any page hence the effect of an attack vector insertion is not visible anywhere. Therefore, an xss attack is not possible here.

74. **False positive:** the variable `$user_id` echoed on line 63 is of hidden type and the user cannot see it on the `settings.php` page hence, an xss attack is not possible here.

75. **False positive:** the variable `$user_id` echoed on line 99 is of hidden type and the user cannot see it on the `settings.php` page hence, an xss attack is not possible here.

76. **False positive:** `ssp.php` file is a file from a third party plugin used on the inventory management system webapp and it can't affect what is being echoed on different pages in the webapp. It is typically a helper hence, a xss attack is not possible here.

4 Fix Vulnerabilities

After careful analysis, there exists 10 true positives (12, 21, 22, 23, 25, 32, 38, 39, 61, 71). So I automated each true positive process to reproduce the exploit using Selenium. The source code can be found here: `https://github.com/mondayevidence/SecurityTestingProject`. Afterwards, I patched each vulnerable file:
A simple function used in patching the vulnerabilties across different pages is the **htmlentities** function. I sanitized all input coming from the user with **htmlentities** function. This function function converts all characters which have html entity equivalent to their respective entities. With this, if a user inserts html or javascript characters, they are converted to strings that can be compiled or executed by the compiler.

- **12.** In order to fix the vulnerability in the `settings page`, I added "html entities" function to the user input declared as `$clientName` in `changeUsername.php`:
  $username = htmlentities($_POST['username']);

- **21.** To fix the vulnerability in the `brand page`, I added "html entities" function to the user input `$brandName` on line 9 in `editBrand.php`:
  $brandName = htmlentities($_POST['editBrandName']);

- **22.** To fix the vulnerability in the `Categories.php` page, I added "html entities" function to the user input on line 9 in `editCategories.php`:
  $brandName = htmlentities($_POST['editCategoriesName']);

- **23.** Added "html entities" function to the user input `$clientName` and `$clientContact` on line 10 and 11 in `editOrder.php`: $clientName = htmlentities($_POST['clientName']); & $clientContact = htmlentities($_POST['clientContact']);
- **25.** To fix the vulnerability in the `Product.php` page, I added "html entities" function to the user input on line 10 and 11 in `editProduct.php`: $productNamee = htmlentities($_POST['editProductName']); & $rate = htmlentities($_POST['editRate']);
- **32.** To fix the vulnerability in the `user.php` page, I added "html entities" function to the user input variable declared as `$edituserName` in `editUser.php`: $edituserName = htmlentities($_POST['edituserName']);
- **38.** Added "html entities" function to the user input declared as `$clientName` in `editOrder.php`: $clientName = htmlentities($_POST['clientName']);
- **39.** Added "html entities" function to the user input declared as `$clientContact` in `editOrder.php`: $clientName = htmlentities($_POST['clientContact']);
- **61.** Added "html entities" function to the aforementioned user input, fixes this vulnerability.
- **71.** Added "html entities" function to the user input declared as `$clientContact` in `editOrder.php`: $clientName = htmlentities($_POST['clientContact']);

To prove verify that each patch worked and ran my automated script after sanitization. Each test cases failed because the xss attack vector have been converted to their respective html entities.

```
org.junit.ComparisonFailure:
Expected :<h1>Samsung</h1>
Actual   :&lt;h1&gt;Samsung&lt;/h1&gt;
<Click to see difference>
```

Figure 1: Test case failure message for patched editProduct page

```
org.junit.ComparisonFailure:
Expected :<h1><b>SmartPhone</b></h1>
Actual   :&lt;h1&gt;&lt;b&gt;SmartPhone&lt;/b&gt;&lt;/h1&gt;
<Click to see difference>
```

Figure 2: Test case failure message for patched editCategory page