XSS Detection in Web Request and Server Response

Sudhir S. Dhekane¹, Prof. V. B. Gaikwad²
¹²Computer dept, Terna Engineering College, Navi Mumbai, India

Abstract-- In the today’s world most of the electronic transactions, web services run via website with stateful reliable communication by transmitting, storing, retrieving essential data on network, client and webserver. The major problem with such reliable communication is that they become vulnerable to an XSS type of attack which mostly carried out to steal essential and confidential information, gain control of stateful communication, to change browser settings and cause fraudulent activities for financial gain.

This paper introducing basic types of XSS attack, the review of different literatures to find out plus minus points in existing XSS detection systems and, understanding their importance with respect to security for web applications and proposing system to prevent such attacks on client and server side named as “XSS Detection System”

The fifth Section is about performance measure and evaluation and the sixth section conclude the paper with discussion of how proposed method may better than available ones.

II. WHAT IS XSS?

Cross-Site Scripting (XSS) attacks occur when:

1. Data input in Web application through an untrusted source, mainly a web request.
2. The data is included in dynamic content that is sent to a web user as HTTP Response without being validated for malicious script.

The malicious content sent to the web browser is a piece of JavaScript, but it may also include HTML or any other type of code that the browser may able to execute. The variety of attacks based on XSS is very vast, but commonly they include transmitting confidential data like cookies or other essential session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the appearance of the vulnerable site.

Classification of XSS Attacks

It is too difficult to categorize XSS attacks. Generally they are categorized into two categories, stored and reflected. There is a third, much less well known type of XSS attack called DOM Based XSS.

Stored XSS Attacks:

These are the attacks where the injected script is permanently stored on the targeted servers, such as in a database, in a message forum, visitor log, comment field, etc. The injured party then retrieves the malicious script from the server when it requests the stored information. Stored XSS is also sometimes called Persistent XSS or Type-I XSS. The Figure 1 showing how this type of attack is carried out.
Reflected XSS Attacks

These are the attacks where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another path, such as in an e-mail message, or on some other web site. When a user will attempt click on a malicious link, submitting a specially crafted form, or even just browsing to a malicious web site, the injected code travels to the vulnerable web site, which reflects the attack back to the user’s browser. The browser then executes the code because it assumes that came from a reliable server. Reflected XSS is also sometimes known as Non-Persistent or Type-II XSS. The Figure 2 showing how this type of attack is carried out.

XSS Attack Consequences

- The end result of an XSS attack is the same apart from whether it is stored (Persistent), reflected (Non-Persistent). The difference is in how the payload arrives at the server.
- The most severe XSS attacks involve leak of the user’s session cookie, which allows an attacker to take control on user’s session and get hold of the account.
- Other injurious attacks include the disclosure of end user documentations, installation of Trojan horse programs, redirect the user to some other page or site, or modify presentation of content.
- An XSS vulnerability allow an attacker to modify a press release or news item could affect a company’s stock price or lessen consumer confidence.

After review of the literature [2][3][4][5][6][7], We proposed new system which will combine two approaches [4][7] to detect an XSS attack in web request and in server response. The following are the same approaches which we are going to combine to propose new approach among which, we have completed implementation of first approach successfully.

**Behavior-based anomaly detection on the server side to reduce the effectiveness of Cross Site Scripting vulnerabilities**

by Jayamsakthi Shanmugam, M.Ponnavaikko [4]

Author’s research aims to use the positive security model to reduce the processing time and by introducing the application level attributes. Their proposed solution comprises of four components namely analyzer, parser, verifier and white listed tag cluster and the interactions between them.

**Proposed Solution Procedure:**

Following components and their interaction shown in figure addressing the XSS vulnerability at server side the components are:

1. Analyzer
2. Parser
3. Verifier
4. Tag Cluster (White Listed Clusters)

The following definitions are made to define the tags with respect to the group of tag clusters and are used to form the rules to identify the vulnerability.

Let I= {I1, I2, I3… In} be a finite set of tags in the input. Let W = {W1, W2, W3… Wn} be the finite set of white listed tags. Ii is the tag content to determine whether the input provided is malicious.
Rules to conclude an input as malicious input is defined as follows:

- If the input is Malicious, only if it is a subset of \{W_1, W_2, W_3, ..., W_n\} where \(I_i\) is the tag in the input.
- If web request attributes (e.g., userName, Address, etc.) i.e., all input parameters of an web application available in the web request are extending character length greater than 20 then there may possible the XSS attack.
- If web request contain any special symbols (which are mostly available in vulnerable script).

If there is no vulnerability detected then the verifier returns the status to parser. The parser then returns the status to analyzer. Based on the status returned, analyzer either redirects the request to the error page or forwards the request to the web application.

The following are the advantages of this approach:

1. This approach allows tags to be entered in the web application and at the same time provide security for the web application.
2. The research work uses the positive security model to reduce server load for processing of XSS detection at server side.

The following are the Disadvantages of this approach:

1. This approach needs an updation in the white listed cluster XML data, when a new tag needs to be permitted.
2. As of now the Behavior-based anomaly detection on the server side approach does not address all the encoding patterns.

**Injecting Comments to Detect JavaScript Code Injection Attacks By, Hossain Shahriar and Mohammad Zulkernine[7]**

In this review, Author’s addresses the issues like an injection of malicious scripts by third parties in the form of overriding the available methods but with different implementations, injection of bad inline functions, injection of additional methods which may be malicious or not by developing a server side JavaScript code injection detection approach. They pre and postponed each legitimate JavaScript code block with comment statements that include identical random token then they identify the expected features of a JavaScript code block (e.g., method call, method definition), save the features in policies, and embed the policy information into comments. During the deployment phase, they performed a number of checks to detect injection attacks. These include (i) code without comment, (ii) code with correct and duplicate comment, and (iii) code with correct and non-duplicate comment; however, the actual code features are not matching with the intended features specified in a policy.

They apply the proposed approach for server side programs implemented in Java Server Pages (JSP). They developed a prototype tool in Java to inject JavaScript comments and generate policies based on legitimate code features then they had deployed the injected code detector as a server side filter. They evaluated their approach with three real world JSP programs.

**Figure 3: Flow of input through the Web Application [4]**

Figure 3 describes the flow of the system. The execution sequence is numbered in the above diagram for better understanding of the process. Analyzer reads application level parameters first, and checks whether the input meets the maximum character rule and encoding rule. Then the input is checked for the special character existence in the input and if it exists then it forwards the request to the parser. The parser splits input to tokens and sends it to the verifier. The verifier accesses the white listed cluster and checks for its vulnerability.
Their evaluation indicates that the proposed approach can mitigate many types of injected JavaScript code that might contain arbitrary and legitimate method call injection, and method definition overriding. The result showed that the approach suffers from zero false negative rates.

III. PROBLEM STATEMENT

The XSS (Cross Site Script) attack is most resent attack, today many web applications of all kinds mostly E-Commerce is suffering through this top 3rd vulnerability [1]. The existing methods are detecting cross site scripting either in web request or in dynamic content generated by server as a response but, there is no system reviewed which can perform detection of the cross site scripting in web request as well as in server response. So we came to define system which initially validate web request for cross site script vulnerability and will take a decision about forwarding the request next to the web server or to reject, and in the dynamic response page contents generated by the web server which is essential to check. If XSS is detected in the response, system will alert the user about XSS vulnerability in the response. If XSS is not being detected, the response can be considered as valid responses and forwarded to the client safely.

Scope:
The scope of the proposed system is the detection of the craft and malicious XSS of type JavaScript only.

IV. PROPOSED SYSTEM

After the reviews [2][3][4][5][6][7] we proposed a system which will combine two approaches [4][7] to detect an XSS attack in web request and in server response. The proposed system is trying to provide combine solution to detect XSS in both the web request while forwarding it to web server by validating and server side by validating response generated by vulnerable server.

The proposed system is divided in two parts:

1. XSS Detection of client’s web request
2. XSS Detection of dynamic response generated by Server

Followings are the components of each,

XSS Detection of client’s web request:

- Input Validator (Request parameter validator)

XSS Detection in response generated by vulnerable Server:

- Comment Injector
- Feature Extractor
- Policy storage
- Feature Comparator
- XSS Handler

The following figure 4 gives the proposed system for detection of cross site scripting in web request and server response.

Figure 4: Workflow Of XSS Detection System

XSS Detection in web request:
The XSS detection in web request is done by performing following steps. The input request will be analyzed by Analyzer to check request inputs contain malicious links or malicious script as a request parameter in crafting form. The crafting is done by using alternate character set for e.g. UTF 7, UTF 8 or by inserting some special symbols. This is because the script functions can only be executed when it is embedded using the tags and special characters. For example ‘<’, ‘>’, ‘%', '&’, ‘\’, ‘&#’ are few of the special characters used to embed JavaScript functions in the tags. If special characters exist in the input, then the input is passed to parser. The parser will extract all tag information from input if exist and pass it to verifier; verifier will access the white listed tags i.e. safe tags and compare it with the tags extracted by parser.
If the tags exist in input among the tags not listed in white list are considered as malicious tags and the verifier will pass the comparison status back to analyzer to take further decision about to forward the request or to put an alert message of XSS vulnerability.

Let \( I = \{I_1, I_2, I_3 \ldots I_n\} \) be a finite set of tags in the input. Let \( W = \{W_1, W_2, W_3 \ldots W_n\} \) be the finite set of white listed tags. \( \{MS_1, MS_2, MS_3 \ldots MS_n\} \) be the corresponding set of security classes for the tag \( W_i \) to identify the attribute or the value of the tag content to determine whether the input provided is malicious.

Rules to conclude an input as untainted input is defined as follows:

If \( I_i \) is not as per the application level parameters set. \( I_i \) is untainted, only if it is a subset of \( \{W_1, W_2, W_3 \ldots W_n\} \) where \( I_i \) is the tag in the input and if security classes identify the attribute’s value as untainted

**Algorithm:**

1. INPUT “HTTP Request” (Web Request)
2. Analyzer reads the request and analyzes the behavior of the request parameters
3. IF(request parameters length > 20)
   3.1 Alert(“Authentication parameter character length exceeds ”);
4. IF(request parameters length < 20)
   4.1 IF (special symbols exist)
      Invoke Parser to extract all available tags in request and call Verifier
5. Verifier will access white listed tags
   5.1 IF (Tags exist in request! = White Listed Tags)
      Alert(“XSS vulnerability message ”);
   5.2 ELSE
      Forward Request to server

The following figure 5 shows the workflow of cross site scripting detection in web request.

**XSS Detection In Web Request:**

![Workflow of XSS Detection in Web Request](image)

**XSS Detection in response generated by vulnerable Server**

In the dynamic response page contents generated by the web server which is essential to check for XSS. If XSS is detected in the response, system will alert the user about XSS vulnerability in the response. If XSS is not being detected, the response can be considered as valid responses and forwarded to the client safely. This detection will be processed with following steps

**Comment Injector:**

The Comment Injector uses a prototype tool in Java that parses source files with Jericho [10] and injects comments in JavaScript code blocks present. To insert comments, we modify appropriate program locations (e.g., event handler method call embedded with comments). The Output Document object allows us to save the modified program back to a source file.

For example following table I shows the example of comment injector.
### Table I

<table>
<thead>
<tr>
<th>Type</th>
<th>Example Code</th>
<th>Code with injected comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inline</td>
<td><code>&lt;script&gt;</code> function foo(){...};</td>
<td><code>&lt;script&gt;/!*t1*/</code> function foo(){...};</td>
</tr>
<tr>
<td>Script inclusion</td>
<td><code>&lt;script&gt;</code> <code>/*t1*/</code></td>
<td><code>&lt;script&gt;</code> <code>/*t1*/</code></td>
</tr>
<tr>
<td>(local)</td>
<td><code>&lt;script&gt;</code> src=&quot;a.js&quot;&gt;</td>
<td><code>&lt;script&gt;</code> src=&quot;a.js&quot;&gt;</td>
</tr>
<tr>
<td>Script inclusion</td>
<td><code>&lt;script&gt;</code> <code>/*t1*/</code></td>
<td><code>&lt;script&gt;</code> <code>/*t1*/</code></td>
</tr>
<tr>
<td>Event handler</td>
<td><code>&lt;input onclick=&quot;foo();&quot;/&gt;</code></td>
<td>`&lt;input onclick=&quot;/!*t1*/foo();/*t1*/&quot;/&gt;</td>
</tr>
<tr>
<td>URL attribute value</td>
<td><code>&lt;a href=&quot;javascript:window.history.goback()&quot;&gt;</code></td>
<td><code>&lt;a href=&quot;/!*t1*/javascript:window.history.goback()/*t1*/&quot;&gt;</code></td>
</tr>
</tbody>
</table>

### Table II

<table>
<thead>
<tr>
<th>Type</th>
<th>Example Code</th>
<th>Expected Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>User defined method (named)</td>
<td>function foo (x, y){...}</td>
<td>[foo, 2, x, y]</td>
</tr>
<tr>
<td>User defined method (anonymous)</td>
<td>var foo = function (x, y){...}</td>
<td>[foo, 2, x, y]</td>
</tr>
<tr>
<td>Host object method (override)</td>
<td>document.write = function (arg1){...}</td>
<td>[document.write, 1, arg1]</td>
</tr>
<tr>
<td>Simple method call</td>
<td>foo(2,3)</td>
<td>[foo, 2, 2, 3]</td>
</tr>
<tr>
<td>Nested method call</td>
<td>foo(2, foo(2,3))</td>
<td>[foo, 2, 2, [foo, 2, 2, 3]]</td>
</tr>
</tbody>
</table>

### Feature Extractor:

The Feature Extractor uses Rhino [9] to parse the JavaScript code and extract the features based on the abstract syntax tree. The random token is generated before inserting comments. And finally the extracted features store the generated policies in policy storage. The following table II showing an example of extracting features.

### Policy Storage:

The extracted code features are stored in policy files so that JavaScript code present in response pages can be compared against these known features to detect XSS. The policy storage stores the feature information in XML format. Moreover, comments include the policy information.

Table III shows example code, policy information, and modified comments with policy information in the first, second, and third columns, respectively.
Table III
Example of policy generation [7]

<table>
<thead>
<tr>
<th>Example code</th>
<th>Example policy</th>
<th>Modified comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;script&gt;function foo(x, y){...};&lt;/script&gt;</td>
<td>&lt;policyID&gt;1&lt;/policyID &gt; &lt;type&gt;def&lt;/type&gt; &lt;name&gt;foo&lt;/name&gt; &lt;paramCount&gt;2&lt;/paramCount&gt; &lt;param&gt;x&lt;/param&gt; &lt;param&gt;y&lt;/param&gt;</td>
<td>/<em>t1:1</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/<em>t1:1</em>/</td>
</tr>
<tr>
<td>&lt;input onclick = “foo (2, 3);” ... /&gt;</td>
<td>&lt;policyID&gt;2&lt;/policyID &gt; &lt;type&gt;call&lt;/type&gt; &lt;name&gt;foo&lt;/name&gt; &lt;argCount&gt;2&lt;/argCount&gt; &lt;arg&gt;2&lt;/arg&gt; &lt;arg&gt;3&lt;/arg&gt;</td>
<td>/<em>t1:2</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>/<em>t1:2</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>... /&gt;</td>
</tr>
</tbody>
</table>

The following figure 6 shows the workflow of cross site scripting detection in server response.

Figure 6: Workflow of XSS Detection In Server Response

Feature Comparator:
Feature Comparator will compare the response against with the stored features and any mismatch occurs i.e. additional script or change at valid script then it’s an availability of XSS hence XSS is detected and it is passed to XSS handler.

XSS Handler:
XSS handler will alert about XSS to user if response has been modified otherwise it will forward it to client.

Advantages of Proposed System:
1. XSS detection without modification in script engine and browser
2. Detects XSS at in both i.e. request and response
3. The proposed approach does not require executing actual JavaScript code. Rather, it detects injected code and nullifies any potential damages of website users early.

V. PERFORMANCE MEASURE AND EVALUATION
To perform and evaluate the first approach the samples of web request tested against XSS through reference websites [15][16] and following results has been obtained.
The table IV showing the inputs and tests with respect to web request. The output V refers to valid request where as M refers to malicious request.

Table IV
Request Samples tested against XSS

<table>
<thead>
<tr>
<th>Sr. No</th>
<th>Input: Web Request</th>
<th>O / P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;a href=<a href="http://www.google.com?t1=aa&amp;%3Eabc">http://www.google.com?t1=aa&amp;&gt;abc</a>&lt;/a&gt;</td>
<td>V</td>
</tr>
<tr>
<td>2</td>
<td><a href="http://www.mysite.com/page.html?parameters=">www.mysite.com/page.html?parameters=</a>&lt;script&gt;alert(document.cookie)&lt;/script&gt;</td>
<td>M</td>
</tr>
<tr>
<td>3</td>
<td><a href="http://www.mysite.com/confirmaddress?address=">www.mysite.com/confirmaddress?address=</a>&lt;script&gt;alert(document.cookie)&lt;/script&gt;</td>
<td>M</td>
</tr>
<tr>
<td>4</td>
<td>&lt;a href=<a href="http://www.safesite.com/myFirstName?first_name=">http://www.safesite.com/myFirstName?first_name=</a>&lt;SCRIPT&gt;alert(document.cookie)&lt;/SCRIPT&gt; &quot;&gt; This is a great site&lt;/a&gt;</td>
<td>M</td>
</tr>
<tr>
<td>6</td>
<td>&lt;script&gt;src=’data:text/javascript;base64,dG9wLmxvY2F0aW9uPSJodHRwOi8vd3d3Lmdvb2dsZS5jb20iOw==’&gt;&lt;/script&gt;</td>
<td>M</td>
</tr>
<tr>
<td>7</td>
<td>&lt;object data='data:text/html;base64,PHNjcmlwdD50b3AubG9jYXRpb249Imh0dHA6Ly93d3cuZ29vZ2xlLmNvbSI7PC9zY3JpcHQ+’&gt; &lt;/object&gt;</td>
<td>M</td>
</tr>
</tbody>
</table>

1131
To perform and evaluate the second approach, the examples of dynamic HTML responses programs are considered randomly from the available sources. Table V shows the parameters of the JavaScript code that will be present in these programs: inline, script source inclusion (local), script source inclusion (remote), event handler, and URL attribute value. Most of the JavaScript code is present as inline code or event handler attribute value. These programs employ very few numbers of local JavaScript source file inclusion. However, none of the programs include remote JavaScript file source. Table VI shows the parameters of the JavaScript code features and the number of policy generated for the programs. The code features include the number of the method definition, parameter count, method call, argument count, user defined method call, and host object method call.

### Table V
JavaScript code parameters

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Inline</th>
<th>Source Inclusion (Local)</th>
<th>Source Inclusion (Remote)</th>
<th>Event Handler</th>
<th>URL Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Program2</td>
<td>21</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>Program3</td>
<td>15</td>
<td>1</td>
<td>0</td>
<td>13</td>
<td>21</td>
</tr>
</tbody>
</table>

### Table VI
Feature and policy Generation

<table>
<thead>
<tr>
<th>Program Name</th>
<th>M. D</th>
<th>P. C</th>
<th>M. C</th>
<th>U.M.C</th>
<th>H.M.C</th>
<th>T.P</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program1</td>
<td>6</td>
<td>17</td>
<td>13</td>
<td>11</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>Program2</td>
<td>45</td>
<td>82</td>
<td>84</td>
<td>81</td>
<td>3</td>
<td>44</td>
</tr>
<tr>
<td>Program3</td>
<td>26</td>
<td>38</td>
<td>36</td>
<td>13</td>
<td>23</td>
<td>52</td>
</tr>
</tbody>
</table>

### Evaluation:
The evaluation applies eight types of JavaScript code injection attack. Table VII shows an example of these attacks (denoted as a1-a8). The first four attacks (a1-a4) inject no injection of comments. However, the remaining attacks include injection of incorrect (a5 and a6) and correct (a7-a8) comments. Moreover, the attacks include injection of arbitrary (a1 and a2) and legitimate code injection that is present in programs (a3-a8). The code injection includes both method call (a1, a3, a5, and a7) and method definition overriding (a2, a4, a6, and a8).

### Table VII
Attacks with their description

<table>
<thead>
<tr>
<th>Attack</th>
<th>Attack Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>Inject a method call dissimilar to legitimate method call (no comment).</td>
</tr>
<tr>
<td>a2</td>
<td>Inject a method definition and a call dissimilar to a legitimate method call (no comment).</td>
</tr>
<tr>
<td>a3</td>
<td>Inject a method call similar to a legitimate method call (no comment).</td>
</tr>
<tr>
<td>a4</td>
<td>Override a method definition (no comment).</td>
</tr>
<tr>
<td>a5</td>
<td>Inject a method call similar to a legitimate method call (incorrect comment).</td>
</tr>
<tr>
<td>a6</td>
<td>Override a method definition (incorrect comment).</td>
</tr>
<tr>
<td>a7</td>
<td>Inject a method call similar to a legitimate method call (correct comment).</td>
</tr>
<tr>
<td>a8</td>
<td>Override a method definition (correct comment).</td>
</tr>
</tbody>
</table>

The Table VIII shows an example of the injected JavaScript code (columns 3-10) for the three programs: Program1, Program2, Program3 (Fn. [Function], Inj. [Injected scripts], Det. [Detected]).
**False positive Rate:**

To evaluate false positive rate, the injection of legal inputs, deploy instrumented programs, enable comparator, and visit related programs that allow displaying the legal inputs. Then after the checking if any of the input is missing in the response page to identify false positive warnings. The false positive rate is computed by the ratio between the total number of false warnings to the total number of policies checked [8].

\[
FPR = \frac{\text{Total No. Of False Warnings}}{\text{Total No Of Policy Checks}}
\]

**Their proposed approach has several advantages.**

1. No information needs to be sent to the browsers, and no modification of browser or server script engine is required.
2. A program suffering from weak client or server side input validation can be safe against JavaScript code injection attacks i.e. XSS.
3. The proposed approach does not require executing actual JavaScript code. Rather, it detects injected code and nullifies any potential damages of website users early.

**VI. CONCLUSION**

The XSS is the top vulnerability today which perform malicious activities by injecting JavaScript method calls and method definition overriding. The proposed method in this report addresses this vulnerability challenge by injecting comments for legitimate JavaScript code that encode legitimate code features in terms of method definition and call signatures. This approach makes an attacker very difficult to inject arbitrary and legitimate JavaScript code. During response page generation, the proposed approach performs a number of checks to detect injected JavaScript code. The approach detects a wide range of attacks including arbitrary and legitimate method call injection and overriding legitimate method definition.

The Implementation completed for detection of XSS in web request tested for number of input web request collected from various reference sites and successfully detected XSS. The implementation of second approach is also completed and also it’s successfully detecting XSS vulnerability if any the dynamic contents have been added in valid response.