ISSN:0975-9646

Securing Server/Client side Applications against XSS attack via XSS-Obliterator

Amit Singh[#1], Suraj Singh Tomer[*2]
Mtech Computer Science Engineering, RGPV
Airport Bypass Road, Gandhi Nagar, Bhopal, Madhya Pradesh 462036

Abstract— **In the modern technological epoch, the internet advancement is at its peak and the web services are emerging more towards dynamic than static web pages. In order to serve the demands, websites holds many applications that extensively opens door for several script-languages. Contrarily, Cross-site-scripting (XSS) attack exploits wide variety of Script-languages and various programming techniques that can easily breach the security of the website. This paper presents a model of XSS-obliterator which supplements the security at client/server side with the mechanism of two-way filter and delivers a platform-independent elucidation to cater security against enormous variants of XSS attack. To address the security issues, an open-source PHP based website is evaluated to render threat against XSS-vectors injected in input fields, URL and source-code using two commercial browsers. As a result of evaluation, the vulnerable sections of the website are declared as high/low recommendation for the proposed model. Considering the extracted artifacts, an experiment has been conducted on the website using the proposed model for detecting and sanitizing all the variants of XSS vectors.**

Keywords— **Cross-site-scripting (XSS), Input validation, Escaping, Sanitization, Document –object-model (DOM), Reflection point.**

I. Introduction And Motivation

Cross site scripting (XSS) is conceivably one of the most devastating web-attack. It is ranked in top 3 vulnerabilities in “OWASP Top Ten report 2013” **[1]** and forth in “CWE/SANS Top 25 Most Dangerous Software Errors” **[2]**. For a cyber-attacker to acquire the control of the website he doesn’t need to break the security of the web server and then upload or alter the files on that server, indeed the attacker can execute XSS attack vectors at the client-side to acquire the control and upload/alter files on that server. The modern web pages fetch data from many different sources and to meet the demand of higher graphics and ever changing content in the web pages they are designed dynamic. Different sources of data contain simple text, images, flash, HTML tags, graphics, videos, music etc. In 2013, the IBM X-Force research team affirmed in “Mid-Year Trend and Risk Report 2013” a detailed analysis of current security landscape, including data on main cyber threats and information on mitigation techniques. The report targets social media as a prime parameter for depicting cyber criminal activities and the users of such websites are unaware of the ongoing menaces. The cyber-attackers also take advantage of the low level security or the accidental flaw left at the time of development of web applications. Majority of web applications continues to be vulnerable to basic SQL injection or cross-site scripting attacks, despite the high level of awareness on these categories of menaces **[3]**.

XSS is an attack in which attacker manipulates the input fields and URL links or source-code of the website by injecting malicious scripts into it. If the input parameters are not validated or sanitized properly, it eases the attacking likelihood of the attacker. The attacker most often use the JavaScript language but in rare cases VBScript, ActiveX, HTML or flash may also be used by the attacker to explicitly inject the malicious code at the client-side. Once the attack is formulated, attacker can perform activities like changing the admin and user settings, gain access of admin panel through Shell, cookie grabbing and poisoning or uploading unwanted contents into the website. Cross-Site Scripting (XSS) attack has shown a drastic growth in breaching the security of websites. Even the tool kiddies are successful in breaching security of many popular websites in the past few years. Cross-Site Scripting (XSS) has unavoidably become the prime security issue in the modern websites. The security analysts prefer to go with fully automated commercial scanners to ensure the vulnerabilities of the website but recent research **[4]** rendered that such scanners shows imperfect results when concerned with persistent XSS and DOM-based XSS. This is because of the dynamic nature of boundless combination of XSS vectors to manipulate the vulnerabilities of the websites. The software developers and security professionals must focus on some of the vulnerabilities usually found in web-sites which may lead to severe damage of web applications.

It may become easy to prevent XSS vulnerabilities during code development by focusing on the identification of some least secure coding. As once they are identified, it becomes very easy to resolve them. XSS basically targets scripts that are injected in a page and executed on the client-side (user’s web browser) than on the server-side. 'XSS' allows the attacker to insert malicious code (client side script) and provide an opportunity for the attacker to bypass access controls from the admin and deface the website , such an internet security weaknesses of client-side scripting acts as the prime culprits for this exploit, which in-turn make XSS attack a big threat. XSS aims to analyze and execute as client-side scripts of a web application in a manner desired by the malicious user. In this exploitation script is embedded in a webpage and executed every time the page is processed. Despite of the fact that these attacks are perilous, developers still lack in updating themselves with the latest exploits and vulnerabilities. Moreover, Security policies are often deemed as an additional endeavor to be considered at the end of the software project. That is why routine security tests should be the part of an effective website development process. Also the effective use of automated tools and scanners at particular time intervals is firmly required. But unfortunately, these tools alone cannot detect and eliminate all kinds of XSS vulnerabilities, mainly because of the unpredictable attributes of malicious code. To protect the client's environment from malicious code, there must be a filter that can detect and prevent malicious activities by the third party. An attacker can lure the client to render the page containing the URL (the location and/or the referrer) partly controlled by the attacker. When the user hits the link and the data is processed by the page (typically by a client side HTML-embedded script such as JavaScript), the malicious JavaScript payload gets embedded into the page at runtime. Input validation using HTTP POST submission method and HTTP GET submission methods are vulnerable to XSS attacks as it allows the user to manipulate the website controls using code implementation. Although the HTTP GET can be exploited more easily by an attacker because all it needs is to change the URL. The exploitation may change according to submission methods used by different web-browsers. To rely completely on tools and scanners can disgrace the developers and clients from the inevitable results. As the attacking techniques are getting advance and vigorous attempts made by attackers to manipulate the malicious code are getting more rapid, different methodology to develop for various platforms are costly and time consuming. Therefore it is intended to propose a methodology to develop a language independent filter and detector that will be updated by the latest threats and will secure web-pages written in various languages in the same manner.

This study intends a security approach in which a model of XSS- obliterator (a 'two way filter') is introduced for detecting and sanitizing the malicious code at the client-side and the server side also. It also classifies the XSS attacks into two categories, in which the second category defines the DOM-based XSS attack which is still the matter of concern for the security analysts. Recently websites like Facebook, Yahoo, Google™, Microsoft Dynamics Canadian **[12]** were observed vulnerable to DOM-based XSS attack as it is much harder to detect due to the fact that it executes at runtime. The model utilizes the enhanced defense mechanisms of input validation, filtering and escaping special characters by exploiting updated features of PHP functions. Furthermore, a platform-independent solution is also proposed that allows the XSS- obliterator to operate on different platforms, i.e. various combinations of web servers, database servers, operating system. The proposed model also addresses the problem of category II DOM-based XSS attack and overcomes the limitation of "L.K Shar and H.B.K Tan", analyzing the client-side script **[9]**. Finally the performance and accuracy of the proposed model is evaluated.

The structure of the paper is as follows. Section II and section III presents preliminary concepts and defense mechanisms. Section IV explains discovering the category I and II of XSS attack and section V demonstrates an experiment that states vulnerable scenarios covering both the categories of XSS attacks. Section VI describes the proposed approach to detect & sanitize XSS vectors. Section VII provides the performance evaluation of proposed model on XSS attacks. Finally Section VIII concludes the paper and addresses the future work.

II. Meticulous Aspect Of Cross-Site-Scripting Vulnerabilities:

In this section the relevant possibilities of XSS attack and the circumstances that may lead to XSS vulnerabilities is interpreted. In order to explore the potential aspects of XSS, a HTML form that accepts the user input and forwards it to the server for processing is analyzed. An excerpt from a query form that uses HTML form to acquire the reply for the query is given below:

```
<form action="query.php" method="GET">
Enter your query:
<input name="id" type="text" size ="25">
<input type="submit">
</form >
```

When the user submits the form, browser creates the "http request" and sends it to the server. It is processed through the "GET" command with the parameter "id" which is responsible for retrieving data from the database. The web application of the server receives the request and start search process for the appropriate term in its database. After the search process is complete, the "http response" is composed by the server which is sent back to the browser as the HTML content.

```
// Initialize
$queryitem = $_GET['id'];
//query process
echo "You have queried for<b>";
echo $queryitem;
echo "</b>.";
//List of Possible answers for the query
```

The above excerpt is a server-side script which receives the term submitted by the user as a http request using PHP's automatically generated global array $_GET['id']. The variable $queryitem stores the http request temporarily and after obtaining the result from the database it echoes the outcome within the page. For example if the user enters "HELP ME" the resultant web page contains the following HTML:

```
< >
You have queried for <b> HELP ME </b>.
< >
```

The GET method with parameter 'id' is echoed uninterruptedly in the resulting HTML page. The GET method retrieves data from the database in context with the http request without validating the input data from the front-end. Therefore it insures that the excerpt is highly vulnerable to the XSS attack. If a JavaScript or a HTML markup is entered in the input field, it will be included in the webpage and displayed to all the users visiting the webpage. For instance, if someone enters a JavaScript "<script> alert("XSS ATTACK ! !")</script>" into the input field or into the URL "id?= <script> alert("XSS ATTACK ! !")</script>", it will validate the input data and the script is injected into the web application permanently. Every time the page loads it will display a popup window prompting the message "XSS ATTACK". The malicious code injection techniques can be either HTML injection or Script injection **[13]**

III.Deliberations For Xss Exploits:

A. Discover XSS attack:

The XSS vulnerability is present in those Web applications that accepts data from users and dynamically include it in servers then on web-pages without properly validating the data. In this process, if the XSS vulnerability is exploited then it allows an attacker to execute arbitrary commands and display content in a victim's browser. They tend to bypass the normal security restrictions and execute XSS vectors to steal victim's session-id and hijack their session **[5]**. While considering the black-box testing approach, the following sections of website is vital to operate:

1) Client-side URL XSS attack: The First type of Attack is 'URL XSS' attack in which the access-point for the malicious code is through the URL of the browser. After inserting the code to the URL of the site, it is submitted to the server to discover the exploit. In this type the URL of the website is tampered, for example if the website link is http://site.com?p=image then it implies that the website has a search engine and currently the web-page 'image' has been requested by the user. This URL can be overwritten with malicious script which in-turn is submitted to the server to discover the vulnerability. This is done by overwriting the 'image' in the URL with the malicious script "<script>alert(1)</script>". The vulnerability assessment of URL-based code injection deliberately prompts the server to verify the URL-based vulnerability **[10]**.

2) Client-side Input field attack: The Second Attack is on input fields. Input fields in the web-page are the text fields, radio-buttons, submit buttons, forms etc. These fields are used to pass data to a server. For example , if we talk about a site with a search engine and suppose we enter 'word', after page loads, it will show 'Found 100 Results For word', which indicates it is displaying data from the linked server, now what if some malicious code is executed? But some scripts never stays on the server, hence these scripts are displayed just once as a page of result.

3) Server-side XSS attack: The Third Attack is the type of attack where the malicious code is injected into the server through the vulnerable input fields. These fields are used as a gateway that sends input-data to the server. If the webpage is vulnerable, malicious code is injected through these fields and server unknowingly includes it into the webpage. The malicious code can be either in HTML or PHP. The HTML code is interpreted by the browser but the PHP code is directly interpreted by the server. For this reason, PHP code is less preferable for the security analysts for the purpose of vulnerability check. Acunetix elaborates the input field vulnerability assessment on an Acunetix test-website with its practical mitigation techniques **[16].** The malicious code injected can either be a PHP or HTML code. HTML is slightly different then PHP, it first downloads the code into the pc in the form of cookies (session), then makes it available to the browser where the browser interpret the code. But in PHP the code is interpreted directly on server (where the script is posted) then the output is returned to the browser. For this reason, PHP is less vulnerable to XSS. In this type of attack, the script injected by the attacker gets stored in the server and is displayed every time as a page of result to all the users of that site. For this reason this attack is more devastating variant of a cross-site scripting flaw.

B. Defense against XSS attack:

To ensure best security policies following defense mechanism is recommended:

1) Differentiating defense mechanism via Input Validation: The primary goal of input validation is to provide better defense mechanism by properly detecting, analyzing, manipulating and validating the unauthorized data arriving before being processed by the application. From the statics of three years (2010-12) "Percentage of Web Applications Containing Input Validation Vulnerabilities" **[11]**, a gradual decline in input validation vulnerabilities present in web applications was observed. If the validation pattern of a particular website is well structured with predefined parameters such as regular expressions, drop down list, radio button, then it becomes easy for the developers to design a secure validation patterns. Input validation method analyzes the data received from URL and input fields of website. The data received from input-fields (text box, search box, forms etc) of the websites requires analyzing POST method at server-side. Similarly data received from URL requires analyzing GET method at the server-side. The GET and POST method is the key focus of the developers at testing-phase. More secure will be the GET and POST method, less will be the chance of XSS attack. An appropriate Input validation requires defined set of rules (section 6B):

- To validate input data of the client side. It includes the filter to check the data length, type and syntax before the server process the data.
- Reject the suspicious pattern from the third party or browser.
- Allow the valid HTML tags that may not be used to execute XSS attack.
- Block all the scripts and HTML attributes.

2) Differentiating defense mechanism via Escaping: The primary goal of escaping is to interpret input data from the browser as regular data not as code. But some web applications avoid this method because it requires use of a standard escaping library like AntiXSS **[14]**. If proper escaping mechanism is used, the evil scripts will not affect other users of the website as it will not allow scripts to execute and treat these scripts as a regular data. For example in HTML, PHP and JAVASCRIPT, dangerous characters such as <, > can be escaped by using &#60 and &#61 respectively. The escaping of HTML, JavaScript, CSS characters are given in Acunetix **[15]**. But attackers find ways to bypass these security barriers by using advance bypassing techniques.

3) In combination with escaping method, Sanitization method is also crucial to consider for better security measures.

4) Differentiating defense mechanism via Sanitization: Sanitization is a technique that either catches or scans the malicious characters from the user inputs and concurrently replaces it with non-malicious ones. The XSS codes could enter through external source such as <FORM> or through complex route such as JSON script **[16]**. Therefore it can't be trusted and proper filtration is required for external sources. The best defense practice for the developers would be to design a filter that will obstruct the special characters ('' # & ( ) / ;) in JavaScript, CSS styles and XML codes containing event handlers. In this technique the suspected codes containing the evil characters is searched by the filters and replaced with the null characters or empty strings (section 6.B). However some intruders comfortably manage to bypass these filters also by using the encoding techniques such as hex encoding, Unicode character variations and null characters. This bypassing technique is known Advance XSS bypassing technique. In order to make these filters more efficient, developers should regularly update and maintain their library by a reliable source like ESAPI, AntiXss and Acunetix. The use of HTML markdown library is good for eliminating HTML markup such as bold, underline, colors as it converts the user inputs into standard XHTML that minimizes the use of HTML markup **[17].**

IV. XSS CATEGORIES WITH THE EQUIVALENT VULNERABLE MODULE

In this paper we have categorized Cross-site-scripting into two categories:
Category I: persistent XSS attack
Category II: Reflective DOM-based XSS attack:

Here we have combined two forms of XSS attack (Non-persistent and DOM based) into a single category: Category II Reflective DOM-based attack. The Non-persistent and DOM based attack works in the same fashion and executed with the aid of a specially crafted link containing malicious code, or a malicious web page laced with a web form, when posted to the vulnerable site will commence the attack. These specially crafted links are the intermediate links of an attacker. If any vulnerable resources accept only the HTTP POST method, the attacker might implant a malicious form. The attacker entices the user to submit the malicious form or click the intermediate link and when the user get trapped, the XSS payload is provoked and executed by the user's browser.

The Persistent XSS attack is different from the other variants of XSS attack because here the code is stored permanently to the database. The attacker generally targets message board, web mail message, web chat, query posts, search engine, login forms or server-side code directly, if admin panel is accessed.

To determine the vulnerability source of both the categories of XSS attack, an open source PHP based website 'LMS' is evaluated on the Bitnami framework. Both the categories of XSS attack with their equivalent vulnerable codes are listed below:

(a) Category-I Persistent XSS vulnerable code at server-side where the visitors seek website's information and (b) Category-II XSS vulnerable script executes at client-side.
snippet (a) <?php $mission_query = mysql_query("select * from content where title = 'mission' ")or die(mysql_error()); $mission_row = mysql_fetch_array($mission_query); echo $mission_row['content']; ?> <hr> <?php $mission_query = mysql_query("select * from content where title = 'vision' ")or die(mysql_error()); $mission_row = mysql_fetch_array($mission_query); echo $mission_row['content']; ?>

snippet (b)
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>

TABLE 1: XSS vulnerable snippets

A. *Analyzing Cross-site-scripting Category I at server-side:*

Persistent (or stored) cross-site scripting vulnerability occur when the attacker injects the malicious code as user input into the server and the code is saved permanently by the server. Thereafter, it is displayed every time as a page of result to the users visiting the webpage in the course of regular browsing, so anyone who views the site may see it permanently. For this reason stored XSS is much more devastating than the reflected XSS. By exploiting the persistent XSS vulnerability, attacker may replicate large amount of malicious data to the users. For example the Samy XSS worm that affected Myspace a few years ago. "The First 24 Hours of Propagation: Samy Sets a Record**"[6].** The vulnerable module in table 1 (a) is a snippet of the page about.php at server-side, which allows the users to seek the website's information. In this case the server-side code is evaluated to determine the persistent XSS vulnerability. Though accessing the admin-panel is the only way to inject XSS payload which not an easy task for an attacker, but if the page contains input-fields such as message and comment, the attacker can exploit the vulnerable code without acquiring access to the admin-panel. The input field penetration is illustrated in table 2.

The vulnerable code $mission_query and $mission_row variables displays and fetch data from the table named 'content'. The echo $mission_row['content'] is directly displaying vulnerable data on webpage. When the malicious JavaScript <script>alert("PERSISTENT ATTACK DEMONSTRATION !!")</script> injected in the about.php from the vulnerable admin panel, it validated the script and the script get permanently stored into the database at line 70 that allowed it to execute on the browser showing a pop-up-window with the message "PERSISTENT ATTACK DEMONSTRAATION !!". Whenever a user visits the page about.php, the pop-up-window will appear each time as a page of result. The attack performed was a passive attack and therefore it did not debilitate the system. But what would happen if an attacker discovers persistent XSS vulnerability before the security analysts? He may apply an active attack to deface the site or steal cookies of the legitimate users. Therefore the study illustrates that it is intended to apply a filter and detector to the vulnerable variables like echo $var (where var can be any variable) that will suitably validate and fetch the data from the database and echo it to the user.

B. *Analyzing Cross-Site-Scripting Category II at client-side:*

The XSS category-II vulnerability deal with the reflective DOM-based XSS attack script. In this category, the attacker performs a runtime embedding of the malicious code in the client side, from within a page served by the web server .Unlike XSS category-I, the reflective DOM-based XSS does not require the web server to receive the malicious code. An attacker don't need to access the source-code at server-side and then inject the XSS vector or check for the vulnerable code that validates the XSS vector and then store it into the database of the server. Instead the attacker seeks for the reflection point (error message, search bar, URL point) in the webpage and then embeds the vector. The XSS vector doesn't get stored into the database and neither processed by the server, indeed it executes at the browser. For instance, In December 2006, Stefano Di Paola and Giorgio Fedon described a universal XSS attack against the Acrobat PDF plugin **[7]**. The attack applied to the PDF documents served to the browser. When rendered by the Acrobat plugin, the javascript get executed **[8]**. The existence of reflective DOM-based XSS attack is typically in the static HTML page. The precondition of a HTML page that processes a DOM attack is the exploitation of the data from the different DOM objects such as document.location, document.URL, document.referrer etc. The LMS's vulnerable module exploits document.URL object that could be a possible target for an attacker. Note that the document object is the representation of the parsed HTML, also these objects are not extracted from the HTML body and do not appear in the webpage.

In the case of LMS, the reflection point is the URL:http://localhost/lms/ajax_xss_obli/Welcome.php?name=amit. The URL is reading the data from the input field "Please type your name" directly to change the page content. When the malicious JavaScript vector :<script>alert(1)</script> injected into the URL, the vector alerted the pop-up-window with message "1" assuring the XSS attack. This vulnerability is due to the object present in JavaScript document.URL.indexOf("name=") that reads the data from the input field "name" and document.write(data) writes the data received from the input field directly to the webpage. The attack worked due to the fact that when the script is injected, the browser received the link and sends an HTTP request to the database to receive the static HTML page. After this, the browser starts parsing the HTML into Document object model. The object used in the HTML page is 'document' with the property of URL and the due to this property the malicious code as a JavaScript executed. When the parser arrives to the JavaScript code in the HTML page, JavaScript is executed and the page content is modified. The script code then refers to the document.URL which is then immediately parsed and the malicious code is executed at the runtime without being processed by the server.

V. EXPERIMENT TO DEMONSTRATE VULNERABLE SCENARIOS:

Based on the extracted artifacts on XSS category I and II, the extent of XSS vulnerabilities are evaluated in the same website. This website is designed for student-teacher interaction to submit assignments, communicate via message, share files and make announcements. The experiment is performed using two commercial browsers Internet Explorer 9 (I.E) and google chrome v31 (G.C) in the bitnami framework with the following specifications:

- Database Server type: MySQL (5.5.34) Community Server (GPL)
- Web Server type: Apache
- phpMyAdmin version 4.1.4

The experiment shows equal set of XSS vectors applied on input fields, source-code & URL points. The case where vector executed is set to 1 else 0.

A. *Scenario X:*

Here the XSS vectors are injected on the input fields: message, add/submit assignment, admin-panel's content. A set of five XSS vectors are taken in which 1st vector is passive and rests are active. The 1st and 2nd vectors are injected on 'message' of the module 'send_message_teacher_to_student' & 'send_message_student_to_teacher'. The 3rd vector is injected on 'admin-panel's content' module and 4th, 5th vectors are injected on 'add/submit assignment' module. It is observed that all the vectors applied on these fields executed on both the browsers and which assures a Category-I XSS vulnerability. The XSS attack executed due to the existence of unsecured sensitive variables $_POST, $_GET & echo $var (where 'var' is a variable) which validated these vectors. These variables are stated as sensitive variables in the table 2. For every vector injected, the corresponding sensitive variable is depicted within the module.

B. *Scenario Y:*

In this scenario Category I XSS attack is illustrated in which a set of five active XSS vectors are injected within the vulnerable statements in the source-code as shown in table 3. The target area for injection is the access-point of XSS vectors in HTML elements like tags, attributes and event handlers present in different modules of the website.

TABLE 2: Scenario X (XSS Category I attack executed on Input-fields)

	XSS vectors	Sensitive variables	Impact without XSS OBLITERATOR	
			I.E	G.C
1	<script>alert(1)</script>	$_POST	1	1
2	<img src="http://localhost/lms/lms/admin/uploads/girl.png">	$_POST	1	1
3	<script>alert(document.cookie)</script>	echo $var	1	1
4	<EMBED SRC="http://localhost/lms/ajax_xss_obli/test.swf" AllowScriptAccess="always"></EMBED>	$_POST	1	1
5	<iframe src=http://localhost/lms/ajax_xss_obli/nonPersistant_v.php>	$_GET	1	1

TABLE 3: Scenario Y (XSS Category I attack executed on Source-Code)

	XSS vectors	Vectors Injected on vulnerable statements	Impact without XSS OBLITERATOR	
			I.E	G.C
1	<!--[if]><script>alert(1)</script --> <!--[if<img src=x onerror=alert(2)//]> -->	within <body>tag	1	0
2	<button formaction="javascript:alert(1)">X</button>	'formaction' attribute	0	1
3	<body oninput=alert(1)><input autofocus>	'oninput' event handler	0	1
4	<iframe srcdoc="&lt;img src&equals;x:x onerror&equals;alert&lpar;1&rpar;&gt;"/>	'srcdoc' attribute	0	1
5	<object allowscriptaccess="always" data="test.swf"></object>	<object> tag directly including flash file	0	1

TABLE 4: Scenario Z (XSS Category II attack executed on URL)

	XSS vectors	Impact without XSS OBLITERATOR	
		I.E	G.C
1	<script>alert(1)</script>	1	0
2	<img src="http://localhost/lms/lms/admin/uploads/girl.png">	1	0
3	<script>alert(document.cookie)</script>	1	0
4	<title onpropertychange=alert(1)></title><title title=></title>	1	0
5	<SCRIPT FOR=document EVENT=onreadystatechange>alert(1)</SCRIP>	1	0

It is observed that XSS vectors executed on both the browser diverged from the previous scenario. Only the 1st vector is executed in I.E browser and except for the 1st vector, rest vectors are executed in G.C browser.

C. Scenario Z:

This scenario of XSS attack entirely represents the Category II Reflective DOM-based XSS attack. A set of five XSS vectors are injected into the URL of the browser. The vulnerable reflective-point evaluated in 4(ii) is taken as an access-point for the set of XSS vectors illustrated in table 4. It is observed that all the vectors are executed in I.E browser and neither of the vectors executed in G.C browser, as shown in table 4. The experiment was aimed to determine the vulnerable areas present in the website where the XSS attack are most likely to occur. The vulnerability that covers both the categories of XSS attack is divided into three Scenarios, Scenario X & Y for the Category I XSS attack and Scenario Z for Category II XSS attack.

TABLE 5: Recommendation for XSS-obliterator

Scenario	XSS Attack Successful in I.E (%)	XSS Attack Successful in G.C (%)	I.E specific applications	G.C specific applications
X	100	100	HIGH	HIGH
Y	20	80	LOW	HIGH
Z	100	0	HIGH	LOW

Based on the evaluation of three Scenarios, Table 5 is formulated to report the liability for the proposed XSS-obliterator in the vulnerable areas. From the experiment it is worth noticeable that the execution of XSS vectors varies in different browsers. Therefore, recommendation for XSS-obliterator in the browser-specific applications is depicted as HIGH and LOW. Where HIGH is ranges between 30-100% and low ranges between 1-29 %. In the Scenario X, necessitate of XSS-obliterator is very high because all the attacks executed in both the browsers. While in Scenario Y, that is in source-code injection, necessitate of XSS-obliterator is low in case of I.E and high in case of G.C and vice-versa in Scenario Z.

VI. PROPOSED MODEL OF XSS-OBLITERATOR:

A. XSS-obliterator framework:

On the basis of the necessity for security mechanism to prevent Category I and II XSS attack, XSS-obliterator is introduced. It is a two way filter-detector that comprises of two phases: detect and sanitize the malicious code (i) from client-side and (ii) from server-side.

The framework for detection and sanitization of the malicious code (XSS vectors) received from client/server-side is shown in figure 1. The framework has six modules: browser, XSS-obliterator, XSS-detector, XSS-filter, malicious-code handler and database. These modules functions in the following steps:

1) First the XSS-detector module in the Application server receives the input data from the browser module via http-request. It may be a legitimate or malicious data.

2) The XSS-detector module contains a pattern of 'malicious code' (used in XSS attacks) and a pattern of 'allowed characters' defined in the XSS-obliterator (allows the data to pass to the server directly).

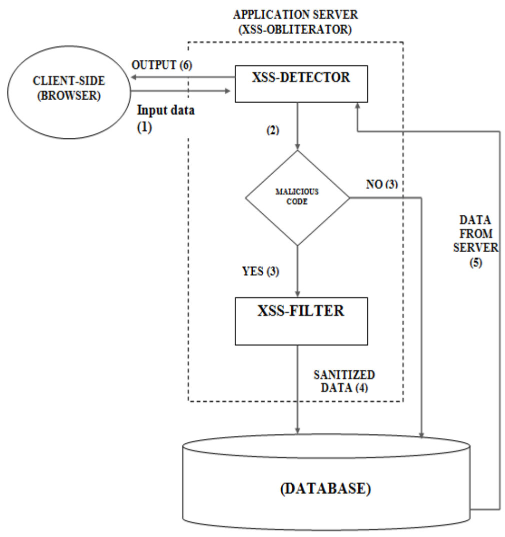

Fig 1: XSS-obliterator framework

3) The input data is then matched with both the patterns in the malicious code handler. If the input-data match the malicious code pattern, the handler passes the data to the XSS-filter module, else the data is passed directly to the database module.
4) The XSS-filter receives the malicious data and sanitizes the malicious characters present in the data. Then the sanitized data is passed to the database module of the server.
5) In response, the database module sends requested data via http-response to the XSS-detector module again. If the response data is malicious, then step 3 is repeated.
6) Finally the browser module receives the sanitized data as response from the server.

B. Algorithm:

Algorithm for XSS-obliterator:

1. Declare allowed HTML TAGS.
2. Detect malicious code using XSS-detector algorithm.
3. If any detection found pass data to XSS-filter algorithm, else display/store the data.

Algorithm for XSS-detector:

1. Convert input data to lower case.
2. Load malicious code patterns.
3. Match patterns in input data.
4. If pattern found in input data set Result to true else false ($result = true).
5. Return the result (return $result).

Algorithm for XSS-filter:

1. Remove any null characters i.e. '\0' and replace it with (‘’) null character.
2. Remove any JavaScript and replace it with null character (‘’).
3. Normalize data using html special symbol code.
4. Convert data to lowercase.
5. Remove all slashes.
6. Match HTML tags in data.
7. Trim and remove malicious HTML tags and protocol data.
8. Accept allowed attributes only.

The XSS-obliterator contains the string ‘$allowed’ that contains the array of HTML tags (i, a, p, img, font, br, pre, ul, li, table, td, tr, th) to be bypassed by the filter and declare it in “allowed characters” pattern. This function classifies the data received from client/server as ‘Rich data’ (set as 0) containing scripts or tags and ‘Legitimate data’ (set as 1) without any special characters. It then classifies the data from server-side/client-side as ‘Rich data’ (set as 0) containing scripts or tags and ‘Normal data’ (set as 1) without any special characters. If the data is a malicious code with a defined malicious pattern in the XSS-detector, The malicious code will be sanitized by the filter and If the data is not malicious then it will be passed to the database (in case of input data from client) or to the browser (in case of data arriving from the server-side) without being sanitized. The malicious pattern is defined in the set of arrays as shown below:

```
$pattern[0] = "(.*)<script>(.*)</script>(.*)";
$pattern[1] = "(.*)<!--(.*)-->(.*)";
$pattern[2] = "(.*)<div (.*)</div>(.*)";
$pattern[3] = "(.*)<(.*)href=(.*)/>(.*)";
$pattern[4] = "(.*)<style>(.*)</style>(.*)";
$pattern[5] = "(.*)&lt;(.*)&gt;(.*)";
$pattern[6] = "(.*)&gt;(.*)&lt;(.*)&lt;(.*)";
```

C. Platform Independent Solution:

The platform –independent solution is obtained by using Ajax given below:

```
var request = $.ajax(
{
url: "http://localhost/lms/ajax_xss_obli/ajax_xss_obli.php",
type: "POST",
dataType: "html",

data: {vdata: text}
 });
```

In this case the XSS-obliterator (ajax_xss_obli.php) is maintained in the centralized php server and the Ajax code is included into the servers seeking security via XSS –obliterator. Thereafter web applications developed in different languages can execute the XSS-obliterator at any operating system.

TABLE 6: Resultant table

XSS VECTORS Variants	Total Number of Variables			XSS Vectors Executed on Vulnerable Variables(XSS OBLITERATOR not Applied)			XSS Vectors Successful on Vulnerable Variables (XSS OBLITERATOR Applied)		
	$_GET	$_POST	echo $var	$_GET	$_POST	echo $var	$_GET	$_POST	echo $var
>250	61	247	620	4	76	219	0	0	0

VII. PERFORMANCE EVALUATION:

To evaluate the performance of the XSS-obliterator against XSS attack manual pen-testing is accomplished on the local- host in which the blending of more than 250 variants of XSS
vector were injected to depict the variables that are vulnerable to XSS attack as shown in table 6. The foundation of category I XSS vulnerability lies within the detected vulnerable variables $_GET, $_POST, echo $var. It is observed that 6/61 $_GET, 86/247 $_POST and 279/620 echo $var variables were found to execute XSS vectors when XSS-obliterator has not applied. But when the XSS-obliterator is applied, it successfully obliterated all the given variants of XSS vectors. The XSS-obliterator also found successful to obliterate the category II reflective DOM-based XSS attack when applied in the vulnerable document-object 'document.write()' in table 1(b).It is worth noticeable XSS-obliterator is also independent of browser-specific applications discussed in section 5. As a result vectors described in Scenario X and Z did not executed on both the browsers when XSS-obliterator is applied. The proposed model does not provide protection against SQL-injection attack. Therefore if an attacker manages to acquire the admin-panel using SQL-injection, he may access the source-code and inject malicious code as shown in table 3 (scenario Y), which leaves the only possibility where an attacker can take advantage over the XSS-obliterator. Furthermore, the fact that the server-side assessment obliterates the resident XSS vectors in the server, this model can also be utilized by those websites that are already contaminated by XSS vectors. In summary, the performance evaluation shows the accuracy of the proposed model to rectify and sanitize the defiled XSS vectors and could be useful in aiding existing automated tools to prevent XSS attacks.

VIII. CONCLUSION AND FUTURE WORK:

The Existing security approaches are inefficient in detecting and sanitizing XSS attacks especially DOM-based attack. Therefore current solutions ardently require the proposed approach of XSS-obliterator to avoid XSS attack. To determine the impact of XSS attack an experiment is conducted in which set of 5 XSS vectors are injected on three vulnerable points: Input-field, URL and source-code categorized into scenarios X, Y, Z respectively. Scenario X & Y demonstrate persistent XSS attack and Z demonstrate reflective DOM-based XSS attack. This illustrates the varying execution of XSS vectors on two commercial browsers when XSS-obliterator has not been applied. Based on these facts,
HIGH or LOW necessity of XSS-obliterator on browser-specific applications is recommended. Thereafter a 6-module-framework of XSS-obliterator and algorithm is proposed. The
algorithm is described in three parts: XSS-obliterator, XSS-detector and XSS-filter. Moreover this paper also proposes a platform-independent solution for the XSS-obliterator by the application of AJAX. Finally the performance of the model is evaluated in which it is proved that the model accurately and
effectively obliterates the persistent and reflective DOM-based XSS attack. For future work the model can be further extended with more complex pattern of malicious code and also avoid centralized server to get overloaded due to huge number of requests to sanitize i/o data.

REFERENCES:

[1] https://www.owasp.org/index.php/Top_10_2013-A3-Cross - Site_Scripting_(XSS)

[2] http://cwe.mitre.org/top25/#CWE-79

[3] http://securityaffairs.co/wordpress/18230/cyber-crime/ibm-x-force-2013-mid-year-trend-risk-report.html.

[4] http://www.acunetix.com/blog/news/acunetix-comparison-web-application-scanners

[5] Cross-Site Scripting (XSS) Tutorial: Learn About XSS Vulnerabilities, Injections and How to Prevent Attacks Example 1, http://www.veracode.com/security/xss.

[6] Cross-Site Scripting Worms & Viruses, The Impending Threat & the Best Defense.

[7] "Subverting Ajax" (23C3 presentation), Stefano Di Paola and Giorgio Fedon, December 2006

[8] https://www.owasp.org/index.php/DOM_Based_XSS .

[9] L.K. Shar and H.B.K. Tan, "Auditing the XSS defence features implemented in web application programs", Software, IET, Vol. 6, Iss. 4, pp. 377–390, 2012

[10] Lwin Khin Shar and Hee Beng Kuan Tan, "Defending against Cross-Site Scripting Attacks", IEEE Computer, Vol. 45(3), pp 55-62, March-2012.

[11] Web Application Vulnerability Statistics 2013 Jan Tudor , June 2013

[12] http://www.acunetix.com/blog/web-security-zone/chronicles-dom-based-xss/

[13] https://www.owasp.org/index.php/HTML_Injection

[14] http://msdn.microsoft.com/en-us/security/aa973814.aspx

[15] http://www.acunetix.com/blog/web-security-zone/preventing-xss-attacks

[16] https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet (RULE #3.1)

[17] http://phpsecurity.readthedocs.org/en/latest/Cross-Site-Scripting-(XSS).html (HTML sanitization).