ISSN: 2581-3404 (Online) **IF: 5.68 (SJIF)**

IJIRTM, Volume-9, Issue-3, June-2025

Protecting Applications on both the Server and Client Side from Cross-Site Scripting Attacks

[1]Bakul Dehariya, [2]Pradeep Pandey

[1]Research Scholar, [2]Assistant Professor

[1]Department of Computer Science & Engineering,

[1] SAM College of Engineering and Technology, Bhopal, India.

Abstract. *The rapid expansion of dynamic web applications has introduced significant security vulnerabilities, particularly Cross-Site Scripting (XSS), which remains among the top threats to modern websites. XSS attacks allow malicious actors to inject client-side scripts into trusted websites, leading to session hijacking, defacement, data theft, and phishing. This research focuses on identifying and mitigating XSS vulnerabilities in web applications, using the LMS (Learning Management System) platform as a case study. Initial evaluations using commercial scanners revealed only a subset of vulnerabilities, prompting the implementation of a thorough manual testing approach. This led to the discovery of critical flaws associated with sanitized input variables such as $_GET, $_POST, $_REQUEST, and direct output via echo. These vectors were found to be exploitable through stored, reflected, and DOM-based XSS attacks. To address the limitations of existing one-way filters and scanners, this thesis proposes a novel two-way filtering and detection mechanism, termed XSS_OBLITERATOR. Unlike conventional tools, XSS_OBLITERATOR is capable of detecting and neutralizing malicious code on both the client and server sides. It offers language-independent protection and is designed to be scalable and platform-agnostic, ensuring robustness across diverse web environments. The research outlines the design and implementation of this framework and evaluates its effectiveness through pilot experiments using various XSS vectors on the LMS platform. The results demonstrate that the proposed solution significantly improves the detection rate and minimizes false positives compared to existing methods. Furthermore, it addresses challenges in securing legacy codebases already compromised by persistent XSS payloads.*

Keywords: Static Taint analysis, symbolic execution, DOM based XSS Cross-Site Scripting (XSS), SQL Injection.

Introduction

This research was initiated on the website 'LMS' which is an open source PHP based website for student and teacher interaction, assignment submission, announcements, quiz and sharing some important files. It was curious to know about the security breaches, obligatory to manipulate in the early phase of web developing. This research started by examining the website for any of the vulnerability present in it. First we applied several commercial scanners which depicted a very few vulnerabilities in the website. Then finally we proceeded to go with the manual depiction of vulnerabilities in the websites. The manual testing performed was heuristic and in-depth and it brought me to the consequence that the website is vulnerable to the variables: echo $var, $_POST, $_GET, $_REQUEST. The variable “echo $” and “= $ any-variable” are responsible for fetching data from the database and include it into the webpage (front-end). $_GET is responsible to fetch or send data from the database using URL. $_POST is responsible to send data to the database using the input fields of the website such as search, form, email, comment etc. $_POST and $_GET are basically used to update, deleted, search, insert data to the database(server). Such variables, if not secured and validated properly, it will be vulnerable to scripting attacks. This scenario is universally functional in all the existing websites also. When LMS source code was inspected, there was ample of security holes present in the mentioned variables. The security holes that was present in LMS website was a typical Cross-Site-Scripting (XSS) vulnerability. The website was already loaded with many XSS vectors and it was found it impractical to first locate all the XSS vectors already existing in the websites and then eliminate them. The fact that existing filter and detector could not fix the already existing XSS vectors was a prime issue to be addressed [1]. Therefore, we developed a two-way-filter and detector mechanism named as “XSS_OBLITERATOR” to fix the bugs. The XSS_OBLITERATOR is an eliminator of malicious code and does a proper validation of the data at both, client and server end. The specialty of this guard is to stop the malicious activity from the attacker and eliminates already existing malicious code in the server.

A. XSS Exploits

XSS exploits are similar to SQL injection, an original form of code injection. This type of attack exploits an application’s output function that references poorly sanitized user input. However, SQL injection targets the query function that interacts with the database, whereas XSS exploits target the HTML output function that sends data to the browser. The basic idea of XSS injection is to use special characters to cause Web browser interpreters to switch from a data context to a code context. For example, when an HTML page references a user input as data, an attacker might include the tag <script>, which can invoke the Java-Script interpreter. If the application does not filter such special characters, XSS injection is successful, and the attacker can perform exploits such as account hijacking, cookie poisoning, denial of service (DoS), and Web content manipulation. Typical input sources that attackers manipulate include HTML forms, cookies, URLs, and external files. Attackers often favor JavaScript and also other kinds of client-side accessed user input in the outgoing web- page. This type of XSS exploit is common in error messages and search results.

The XSS project recently reported multiple reflected XSS holes in McAfee that attackers could exploit to trick users into downloading viruses. Stored or persistent XSS holes exist when a server program stores user input containing injected code in a persistent data store such as a database and then references it in a webpage. Attack against social networking sites commonly exploits this type of XSS flaw. An example is the Samy worm, which, within less than 24 hours after its release on 4 October 2005, caused an exponential growth of friend lists for 1 million Myspace users, effectively creating a DoS attack. Both reflected and stored XSS holes result from improper handling of user inputs in server-side scripts. In contrast, DOM-based XSS holes appear in the Web application when client-side scripts reference user inputs, dynamically obtained from the Document Object Model structure, without proper validation.

B. Example XSS Exploits

For a Web application that lets travelers share tips about the places they have visited. The program contains four input fields Action, ‖ Place, ‖ Tip, ‖ and User‖—that attacker can manipulate. An attacker could send a seemingly innocuous URL link to a victim via e-mail or a social networking site. The script in bold will execute on the victim's browser if the victim follows the link to traveling Forum.

C. XSS Defenses

XSS defenses can be broadly classified into four types: defensive coding practices, XSS testing, vulnerability detection.

1) **Defensive Coding**

Because XSS arises from the improper handling of inputs, using defensive coding practices that validate and sanitize inputs is the best way to eliminate XSS input validation as it ensures that user inputs conform to a required input format. There are four basic input sanitization options. Replacement and removal methods search for known bad characters (blacklist comparison); the former replaces them with non-malicious characters, whereas the latter simply removes them. Escaping methods search for characters that have special meanings for client-side interpreters and remove those meanings. Restriction techniques limit inputs to known good inputs (white list comparison). Checking blacklisted characters in the inputs is more scalable, but blacklist comparisons often fail as it is difficult to anticipate every attack signature variant. White list comparisons are considered more secure, but they can result in the rejection of many unlisted valid inputs. OWASP has issued rules that define proper escaping schemes for inputs referenced in different HTML output.

2) **XSS Testing**

Input validation testing could uncover XSS vulnerabilities in Web applications. Specification-based IVT methods generate test cases with the aim of exercising various combinations of valid/invalid input conditions stated, to avoid the sole dependency on specifications, Nao Li and colleagues attempted to infer valid input conditions by analyzing input fields and their surrounding texts in client-side scripts. Code-based IVT methods apply static analysis to extract valid/invalid input conditions from server-side scripts. In general, the effectiveness of both specifications, the code-based approaches rely largely on the completeness of specifications or the adequacy of generated test suites for discovering XSS vulnerabilities in source code. Only test cases containing adequate XSS attack vectors can induce original and mutated programs to behave differently. Hossain Shahryar and Mohammad Zulkernine developed MUTEC, a fault-based XSS testing tool that creates mutated programs by changing sensitive program statements, or sinks, with mutation operators. Cross site scripting (XSS) vulnerability is caused by the failure of web application in sanitizing user inputs embedded in HTML output pages. Through such inputs, there is a possibility that an attacker injects malicious scripts in the applications. Furthermore, the attacker's purpose may be served when a client subsequently visits an exploited web page causing the injected scripts to be executed by the client's browser. Thus, XSS attack is a type of code injection attack. In addition, injected scripts are written in any type of client-side scripts such as JavaScript, Action Script and VBScript. XSS has been ranked amongst the top two common and serious security laws. In the past, even giant web sites such as HSBC, Google Search Engine, Facebook, Myspace and Vodafone have been reported to contain this type of vulnerability. Code-based extraction of XSS defense artifacts in this concept, it presents the concepts on extracting the program artifacts, which serves the purpose for securing the program from input manipulation attacks. These concepts are built on modeling the possible code patterns of defensive coding methods. Through the empirical studies on many web applications, they observed that the following methods are generally implemented to prevent XSS: (a) input validation; (b) escaping; (c) filtering. These methods are also addressed as sanitization methods in the literature.

a) **Input validation**

It is a traditional approach for handling external data in web applications. This method could reject invalid input immediately. Originally, it is used to ensure input data correctness but in today's web applications, data accuracy could also ensure data security; therefore, nodes in a CFG which implement this defense method should be extracted and checked for adequacy in defending against XSS. Let G be a CFG of a given web program. Let k be a pv-out node in G and vk be a tainted variable referenced at k. There may be more than one tainted variable referenced at k. It is common that the program uses predicate nodes or exception nodes to allow vk to be operated at k only if the value of vk satisfies the user interface specification or some required conditions. It provides a definition that characterizes such node pattern.

b) **Characterizing XSS Defense through Escaping**

Although input validation may be used as a primary defense against all kinds of input manipulation attacks, validation methods may not defend against all XSS attacks. Therefore, for absolute prevention of code injection attacks such as XSS, escaping (also called encoding) is often used to complement input validation. Escaping is a technique that ensures any special characters significant to a certain interpreter are just treated as data not as code. To prevent XSS, proper escaping methods, such as HTML entity escaping, URL escaping, and JavaScript escaping, need to be used according to the context in which the tainted data are referenced (i.e. according to the type of client-side interpreter interpreting the tainted data) . Hence, this method will not work if the escaping scheme is inappropriate. For example, a developer may use SQL escaping scheme (i.e. special characters significant to SQL parser are escaped) on the tainted data assuming that the data are only to be referenced in SQL statements. However, if the data are also used in HTML outputs, the adopted SQL escaping scheme will not escape the special characters significant to the HTML interpreters, thus causing XSS vulnerability. Therefore, nodes implementing such defense method need to be extracted and examined for adequacy.

c) ***Characterizing XSS Defense through Filtering***

Although escaping could completely prevent XSS, it is required that the correct escaping method is applied depending on the context in which the tainted data are referenced. As the use of a standard escaping library is also required, some web applications may not prefer this method. Instead they may apply filtering method to prevent XSS. Filtering is a technique that either removes or replaces malicious characters with non-malicious ones. Among the discussed three defense methods the filtering method is equally important in many web applications as the flow of tainted data through the filtering methods and into the HTML outputs can be very seamless. Such filtering techniques had implemented in a web application is generally carried out by nodes in G that influence the tainted variables of pv-out node.

3) **Vulnerability Detection**

Cross-site scripting (XSS) vulnerabilities can be classified into two types:

- Non-persistent (or reflected) cross-site scripting is a most commonly exploited XSS vulnerability. In this type of attack, the malicious data is reflected immediately on the page by the server without proper sanitization.
- 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. For this reason, stored XSS is much more devastating than the reflected XSS. By exploiting the stored 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).

Literature Survey

Cross-Site Scripting (XSS) remains one of the most widespread and damaging vulnerabilities in modern web applications. Although many solutions have been proposed, XSS vulnerabilities persist largely due to developers' insufficient understanding of the problem and their unfamiliarity with available defensive techniques and their limitations. Existing approaches suffer from issues such as incomplete implementations, performance overhead, complex integration, and high manual effort (Kayson et al., 2019). To address these challenges, researchers have proposed solutions from both the development and detection perspectives. On the development side, improved methods for input validation and sanitization are recommended. For instance, Kayson et al. (2019) introduced an automated technique to generate inputs that expose SQL Injection and XSS vulnerabilities. Their tool, Ariella, symbolically tracks tainted data during execution and mutates it to produce real attack vectors. This method was later integrated into BLUEPRINT, which proved to be effective under stress tests, with minimal false positives and no runtime overhead. Scott and Sharp (2012) proposed a scalable structuring mechanism that abstracts security policies from application logic, alongside a set of tools for secure development and a signature-based misuse detection framework. Their work emphasized that web applications can be protected without being restructured every time new threats emerge. Another notable contribution involved the use of tainted information flow graphs to audit the effectiveness of XSS defense implementations. This method successfully identified all implemented XSS defenses across seven Java-based web applications and helped reduce false positives from existing vulnerability scanners (Author Anonymous, 2021). XSS vulnerabilities typically arise due to improper sanitization of user inputs dynamically rendered on web pages. While many defensive coding techniques exist, including input validation and escaping, their inconsistent use continues to leave applications exposed. As a result, a code-auditing approach was introduced to recover and evaluate the adequacy of defense models in source code, providing guidance on mitigating vulnerabilities (Author Anonymous, 2021). In another study, researchers analyzed several XSS detection methodologies and benchmarked them for performance. Their results demonstrated that JavaScript—while crucial for enhancing interactivity in web applications—also significantly contributes to XSS risks. Client-side scripting allows attackers to exploit browser behavior to execute malicious scripts (Author Anonymous, 2012). To combat this, a passive XSS detection system was developed. By analyzing HTTP request/response patterns in 20 popular web applications, and testing with both real and synthetic attack vectors, the system achieved zero false negatives and an excellent false positive rate in over 80% of cases (Author Anonymous, 2012). Client-side solutions have also been explored. A sandboxing technique was proposed to isolate and analyze suspicious scripts in the browser environment, protecting users without heavily degrading browsing performance (Author Anonymous, 2023). Another study introduced a Dynamic Hash Generation Technique, aimed at rendering cookies useless to attackers by hashing cookie attributes at the server level before storing them in the browser. This strategy mitigates XSS risks by ensuring that even if cookies are stolen, they cannot be reused (Author Anonymous, 2024). Further, parameter manipulation was identified as a common XSS attack vector. Attackers often exploit predictable variable names and parameter structures in web applications to inject malicious scripts. The study highlights the need for context-aware validation and consistent sanitization practices to prevent exploitation (Author Anonymous, 2025). A real-world example illustrating the severity of XSS is the universal XSS attack against the Adobe Acrobat PDF plugin, reported by Di Paola and Fedon in 2016.

This attack executed JavaScript payloads via manipulated links, demonstrating the persistent threat of XSS across even well-established platforms (Di Paola & Fedon, 2016). summary, despite significant research into XSS mitigation, no single solution is fully effective. A hybrid defense strategy—combining robust server-side validation, intelligent client-side filtering, automated detection, and regular code audits—offers the most comprehensive protection against this evolving threat landscape.

Problem Statement

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 client clicks 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 method 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.

Methodology

The introduction of XSS-obliterator is based on the need for a security mechanism to stop all three types of XSS attacks. The harmful code is detected and sanitized

(i) from the client-side and
(ii) from the server-side in this two-way filter-detector.

Figure 5 displays the architecture for identifying and eliminating harmful code (XSS vectors) that are received from the client/server side. Browser, XSS-obliterator, XSS-detector, XSS-filter, malicious-code handler, and database are the six modules that make up the framework. The following steps describe how these modules operate:

1. Initially, the browser module sends the input data via http-request to the application server's XSS-detector module. It might be harmful or valid data.
2. The "malicious code" pattern seen in XSS assaults and the "allowed characters" pattern found in the XSS-obliterator, which permits data to go straight to the server, are both included in the XSS-detector module.
2. The malicious code handler then compares the input data with both patterns. The handler sends the data to the XSS-filter module if the input data matches the harmful code pattern; if not, it sends the data straight to the database module.

Implementation Details & Result Analysis

When an attacker inserts malicious code into the server as user input and the server saves the code permanently, this is known as a persistent (or stored) cross-site scripting vulnerability. After that, it is always shown as a page of results to people who are on the webpage during normal surfing, thus everyone who visits the site may see it forever. Because of this, stored XSS is far more harmful than reflected XSS. An attacker may reproduce a significant quantity of harmful material for users by taking advantage of the stored XSS vulnerability. For instance, Myspace was impacted by the Samy XSS virus a few years ago. “The First 24 Hours of Propagation: Samy Sets a Record”. The basic payload used for testing this vulnerability is the <IMG SRC= “www.site.com/a.img> where the www.site.com is a link of attacker’s intermediate webpage which he links to the victim’s webpage, after the execution of IMG SRC tag the image gets embedded to the server permanently and anybody who views the webpage may see it permanently. Figure 6 shows the detailed scenario with the following steps:

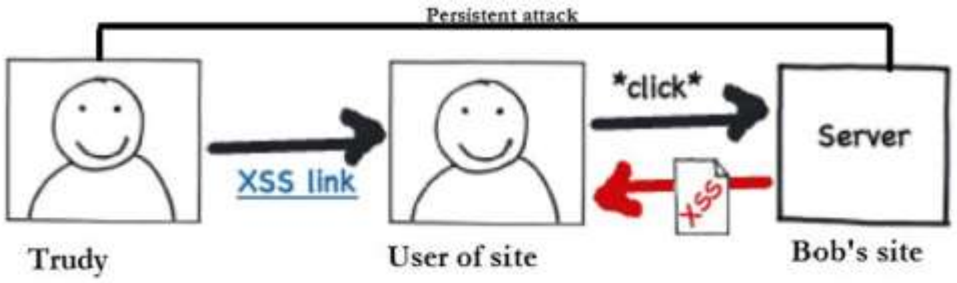

Figure 1: Persistent XSS attack scenario.

XSS-obliterator Recommendation

Table 21 is created to reflect the responsibility for the proposed XSS-obliterator in the sensitive regions based on the assessment of three scenarios. It is important to note from the experiment that various browsers execute XSS vectors differently. As a result, the browser-specific applications' XSS-obliterator recommendations are shown as HIGH and LOW. where low is between 1 and 29% and high is between 30% and 100%. Because every assault was carried out in both browsers in Scenario X, there is a great need for XSS-obliterator. The requirement for an XSS-obliterator is minimal in the case of I.E. and high in the case of G.C. in Scenario Y, which involves source-code injection; in Scenario Z, the opposite is true.

Table 1: 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

6. Result Analysis

Manual pen-testing is carried out on the local host to assess the effectiveness of the XSS-obliterator against XSS attacks. This involves mixing over 250 different XSS vector variations to illustrate the variables that are susceptible to XSS attacks, as indicated in table 22. $_GET, $_POST, and echo $var are the identified vulnerable variables that form the basis of the category I XSS vulnerability. 86/247 $_POST, 279/620 echo $var, and 6/61 $_GET variables were detected to execute XSS vectors in the absence of XSS-obliterator. However, all of the specified XSS vector variations were successfully destroyed when the XSS-obliterator was used. The category II reflective DOM-based XSS attack was successfully eliminated by the XSS-obliterator when it was used on the susceptible document-object "document.write()" in tables 16 and 17. It is important to note that XSS-obliterator is not dependent on

ISSN: 2581-3404 (Online) IF: 5.68 (SJIF)
IJIRTM, Volume-9, Issue-3, June-2025

browser-specific apps. In Section 5.2.3, it is discussed. Consequently, when XSS-obliterator was used, the vectors outlined in Scenarios X and Z did not execute on both browsers.

Table 2: Resultant table.

XSS VECTOR 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	$_POS T	echo $var	$_GE T	$_POST	echo $var	$_GET	$_POST	**echo $var**
>250	61	247	620	6	86	279	0	0	0

There is no defense against SQL-injection attacks in the suggested paradigm. Thus, the only way an attacker may exploit the XSS-obliterator is if he is able to obtain the admin-panel using SQL-injection. In this case, he can read the source code and insert malicious code, as seen in table 18 (scenario Y). Additionally, this technique may be used by websites that are already polluted by XSS vectors because the server-side evaluation eliminates the resident XSS vectors on the server. In conclusion, the performance evaluation demonstrates how well the suggested model fixes and sanitizes the contaminated XSS vectors and may help current automated solutions stop XSS assaults.

Conclusion

The current security methods are ineffective in identifying and eliminating XSS attacks, particularly those that are DOM-based. Therefore, in order to prevent XSS attacks, present solutions strongly need the suggested XSS-obliterator technique. Three susceptible locations—the input field, URL, and source code—are injected with a set of five XSS vectors in an experiment to assess the impact of an XSS attack. These points are classified as scenarios X, Y, and Z, respectively. Persistent XSS attacks are shown in scenarios X and Y, whereas reflected DOM-based XSS attacks are shown in scenario Z. This shows how, in the absence of XSS-obliterator, XSS vectors execute differently on two commercial browsers. These data indicate if XSS-obliterator is HIGHLY or LOWLY necessary for browser-specific apps. A six-module XSS-obliterator and algorithm framework is then suggested. The three components of the algorithm—XSS-obliterator, XSS-detector, and XSS-filter—are explained. Additionally, this work uses AJAX to present a platform-independent solution for the XSS-obliterator. Ultimately, the model's performance is assessed, demonstrating that it properly and successfully eliminates the persistent and reflected DOM-based XSS attack.

References

1. Amit Klein, “DOM Based Cross Site Scripting or XSS of the Third Kind A look at an overlooked flavor of XSS” , http://www.webappsec.org/projects/articles/071105.shtml
2. Jan Tudor, Context Information Security, “Web Application Vulnerability Statistics 2013”, June 2013.
3. http://msdn.microsoft.com/en-us/security/aa973814.aspx.
4. Jeremy Pullicino, “Preventing XSS Attacks”, Acunetix Featured Article MAR 22, 2011.

5. A.Kieyzun, P.J. Guo,K. Jayaraman, and M.D. Ernst, "Automatic Creation of SQL Injection And Cross-Site Scripting Attacks", ICSE '09 Proceedings of the 31st International Conference on Software Engineering, pp. 199-209, May 2019.
6. D. Scott and R. Sharp, "Specifying and enforcing application-level Web security policies", IEEE Transactions on Knowledge and Data Engineering, vol. 15, no.4, pp. 771-783, July-Aug 2003.
7. 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.
8. Tejinder Singh, "Detecting and Prevention Cross–Site Scripting Techniques", IOSR Journal of Engineering, Vol. 2(4) pp. 854-857, April-2012.
9. M. James Stephen, P.V.G.D. Prasad Reddy, Ch. Demudu Naidu and Ch. Rajesh, "Prevention of Cross Site Scripting with E-Guard Algorithm", International Journal of Computer Applications, Vol. 22(5), May-2011.
10. S. Shalini and S. Usha, "Prevention Of Cross-Site Scripting Attacks (XSS) On Web Applications In The Client Side", IJCSI International Journal of Computer Science Issues, Vol. 8(4), July-2011
11. Shashank Gupta and Lalitsen Sharma, "Exploitation of Cross-Site Scripting (XSS) Vulnerability on Real World Web Applications and its Defense", International Journal of Computer Applications, Vol. 60 (14), December-2012.
12. Shashank Gupta, Lalitsen Sharma, Manu Gupta and Simi Gupta, "Prevention of Cross-Site Scripting Vulnerabilities using Dynamic Hash Generation Technique on the Server Side", International Journal of Advanced Computer Research, Vol. 2(3), September-2012.
13. Vishwajit S. Patil, Dr. G. R. Bamnote and Sanil S. Nair, "Cross Site Scripting: An Overview", International Symposium on Devices MEMS, Intelligent Systems & Communication, Proceedings published by International Journal of Computer Applications (IJCA), 2011.
14. Ian Muscat,"The Chronicles of DOM-based XSS", Acunetix Featured Article FEB 26 2014.
15. http://www.acunetix.com/wp-content/uploads/2013/08/Diagram-Describing-Blind-
16. XSS-Attack.gif
17. Michelle E Ruse, Samik Basu, "Detecting Cross-Site Scripting Vulnerability using Concolic Testing", 10th International Conference on Information Technology: New Generations 2013.
18. Yousra Faisal Gad Mahgoup Elhakeem, Bazara I. A. Barry, "Developing a Security Model to Protect Websites from Cross-site Scripting Attacks Using Zend Framework Application", INTERNATIONAL CONFERENCE ON COMPUTING, ELECTRICAL AND ELECTRONIC ENGINEERING (ICCEEE) 2013.
19. Huangcun Zeng, "Research on Developing an Attack and Defense Lab Environment for Cross Site Scripting Education in Higher Vocational Colleges", International Conference on Computational and Information Sciences 2013.
20. HTML5 Security Cheatsheet, https://html5sec.org
21. http://www.w3schools.com/js/js_cookies.asp
22. ZHANG YuQing, LIU QiXu, LUO QiHan1,2 & WANG XiaLi, "XAS: Cross-API scripting attacks in social ecosystems", Science China Press and Springer- Verlag Berlin Heidelberg 2021.
23. Sonika and Yogesh Kumar, "PXSS: Framework to Prevent Cross Site Scripting Attacks", International Journal of Current Engineering and Technology ©2014 INPRESSCO Vol.4, No.5 (Oct 2022).