Received: 28 June 2021 | Revised: 07 August 2021 | Accepted: 10 August 2021
DOI: 10.37917/ijeee.17.2.8 Vol. 17| Issue 2| December 2021

Open Access

Iraqi Journal for Electrical and Electronic Engineering
Original Article

Server Side Method to Detect and Prevent Stored XSS Attack

Iman F. Khazal*, Mohammed A. Hussain
Department of Computer Science, Education College for Pure Science, University of Basrah, Basrah, Iraq

Correspondence
* Iman F. Khazal
Department of Computer Science,
Education College for Pure Science,
University of Basrah, Basrah, Iraq
Email: pgs2183@uobasrah.edu.iq

Abstract
Cross-Site Scripting (XSS) is one of the most common and dangerous attacks. The user is the target of an XSS attack, but the attacker gains access to the user by exploiting an XSS vulnerability in a web application as Bridge. There are three types of XSS attacks: Reflected, Stored, and Dom-based. This paper focuses on the Stored-XSS attack, which is the most dangerous of the three. In Stored-XSS, the attacker injects a malicious script into the web application and saves it in the website repository. The proposed method in this paper has been suggested to detect and prevent the Stored-XSS. The prevent Stored-XSS Server (PSS) was proposed as a server to test and sanitize the input to web applications before saving it in the database. Any user input must be checked to see if it contains a malicious script, and if so, the input must be sanitized and saved in the database instead of the harmful input. The PSS is tested using a vulnerable open-source web application and succeeds in detection by determining the harmful script within the input and prevent the attack by sterilized the input with an average time of 0.3 seconds.
KEYWORDS: Cross Site Scripting (XSS), Stored-XSS (persistent), Web application, Detecting XSS, Preventing XSS.

I. Introduction

Web applications are currently the best way to represent data and provide various services to users via the web. Banking or financial services, educational and news websites, and social media channels are among the services offered. Furthermore, the web application has become the primary means of gathering information on any topic. As a result, the use of web applications has increased, and it has become more appealing to hackers, not just users. This vast amount of sensitive data stored in web applications can be stolen by hackers for a variety of reasons, including monetary gain or spying [1]. The security issues are one of the main dangers that information technology faces and their application, Indicate the Measures Put in the place to maintain them information system capabilities and services from illegitimate access. malicious attack explore a computer and technology-based system companies [2]. The XSS (Cross-Site Script) attack is one of the most common security issues in web applications.

The injection attack, XSS, is one of the most common web application attacks. As a consequence of this attack, sensitive data, cookies, and sessions have been stolen. The injection attack is used, in which malicious scripts are injected into the web application's source code. This type of attack may occur in any web application that does not use the encryption method or verification of the validity of the input. Therefore, the attacker exploits a vulnerability in the application to launch XSS attacks by storing malicious scripts on the website or deceiving the user with the URL injected by malicious script [3].

This attack is aimed at the user rather than the application (the user is the victim). XSS attacks are considered one of the most dangerous approaches that exploit weaknesses in web applications and are ranked second of the most dangerous vulnerability and are considered critical with a rate of approximately 38%. What is concerning, however, is the low rate of solutions or treatment for this type of attack. Furthermore, according to the Open Web Application Security Project (OWASP) report, XSS attacks were ranked seventh out of ten major security risks, while XSS was ranked third in 2013. The severity of this attack is confirmed by Imperva data, which is associated with XSS attacks that exploited the greatest number of Web application vulnerabilities in 2017. Indeed, the number of vulnerabilities exploited by XSS has more than doubled since 2016. According to Imperva prediction, it will be one of the most common attacks in 2018 [4].

Cross-site script (XSS) attacks are carried out by injecting malicious code into various types of interpreters in the user's browser, such as JavaScript, Flash, ActiveX, HTML, VBScript, or any other client-side language. XSS is defined as an attack on a specific website's customers' privacy that involves three parties (the (victim) client, the attacker, and the web application). The goal is to steal or tamper with customer data, which results in:

- Customer identity theft (this is called impersonation)
- Account takeover by stealing authentication information
- Changing customer settings
- Rejection the service by distorting the site
- Include phishing links
- Cookie file theft [5].

There are three types of XSS attacks: Stored (persistent) XSS, Reflected (non-persistent) XSS, and Document Object Model (DOM) XSS [4]. A reflected XSS attack is the most common type. It is also known as a Type-1 XSS attack or a non-persistent XSS attack. When a user (victim) clicks on a link injected by a malicious script (which is commonly in HTTP query parameters), the victim's browser executes it [6].

The Stored XSS, in which the malicious script is injected inside the web application and saved in the database. When a victim visits this site, the web application retrieves information from the data set for display on the client's browser, and the malicious script is executed by the web browser [7].

Dom-XSS occurs when an attacker embeds the malicious script in a link and sends it to the client's machine, attempting to persuade him to click on it, resulting in the client's PC being hacked and the client becoming a victim. It occurs on the client-side rather than the server, as in (reflected and stored) XSS [8].

This paper focuses on stored XSS. It is regarded as the most dangerous because it is stored on the website and can harm any user who visits it. The most common is the Reflected-XSS attack, which affects only one user who clicks on the injected link. The DOM-XSS attack (also known as Type 0) is a more complex and less common type of XSS attack [4].

This paper proposes a method for detecting and preventing stored-XSS attacks. The main idea is to test and sanitize any user input to vulnerable web applications. First, examine the input to see if it contains any script. Second, if a script is detected, examine the inside script to see if it contains any malicious functions. If the malicious function in the script is detected in the user's input, the user is the attacker, and the input is the harmful input. Finally, prevent the malicious script from being executed by sanitizing the input and saving the sanitized input rather than the harmful input in the web application's database. The idea is distinguished by its clarity and simplicity.

The proposed method is a server-side method. It is a comprehensive method for protecting web application users that do not rely on the practices or tools of a specific server-side language. The method is not determined for specified content and reduces a load of work on the client-side so as not to affect the user's browser performance. The contributions are as follows:

- A proposed method for detecting and preventing stored XSS attacks, which are the most dangerous type.
- The proposal is a server-side method for reducing the load on the user's browser. All processing operations of the proposed method were completed without any additional action on the browser's work.
- PSS has been proposed to filter web application inputs to prevent an attacker from injecting XSS payloads. Users with PSS can enter web pages safely and the web applications can avoid stored XSS attacks.

The paper is organized as follows: Section 2 presents the background of stored XSS attacks and the scenario of it. Section 3 related work. Section 4 discusses the proposed method and provides a description of its architecture. Section 5 discusses experimental evaluation. Section 6 is for security analysis. Finally, section 7 discusses the conclusion and future work.

II. STORED XSS (PERSISTENT) ATTACK

This type of XSS attack is the most dangerous because the injection malicious script is stored in the web application's database and affects all users who visit the injection page [9]. Since this malicious script is injected directly into the vulnerable web page, it is referred to as direct XSS [4]. In general, XSS attacks have three main components: the website, the attacker, and the victim.

The website is the HTML page that the user has requested.

The attacker malicious user conducts the attack by exploiting a vulnerability in the website and executing it in the victim's browser.

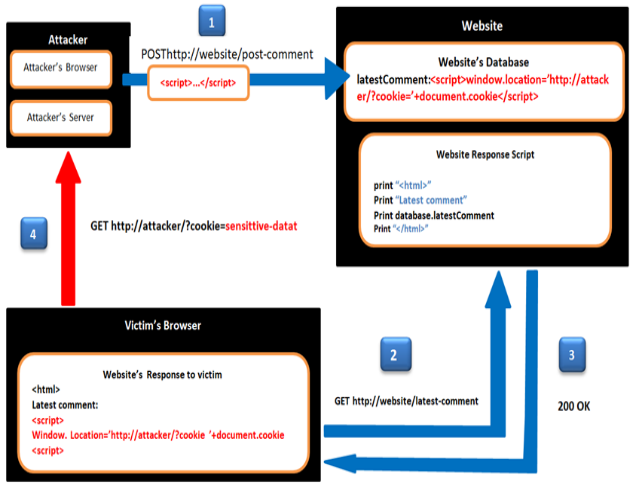
The victim is the ordinary user who visits the website and uses his/her browser to request the page [10]. The following is an example of a Stored XSS attack scenario:

1- The attacker exploits the vulnerability in the web application and injects the malicious Script into the database of the web application.

2- The victim accesses the web application and requests the vulnerable web page.

3- The web application combines the HTTP response with a malicious script and sends it to the victim's browser.

4- The malicious script is executed by the victim's browser, which sends the victim's sensitive information to the attacker [11]. The Stored XSS scenario is depicted in Fig. 1.



```
latestComment:<script>window.location='http://attacker/?cookie='+document.cookie</script>
```

```
print "<html>"
Print "Latest comment"
Print database.latestComment
Print "</html>"
```

```
<html>
Latest comment:
<script>
Window. Location='http://attacker/?cookie '+document.cookie
<script>
```

Fig. 1 Stored XSS attack Scenario [10].

III. Related Work

Parvez et al. (2015) [7] made several recommendations to improve the performance of black-box scanners in detecting stored XSS and SQLI vulnerabilities. The black-box technique is used for web vulnerability scanning. To analyze the two testbeds, the authors use three black-box scanners (WackoPicko and customize Scan-bed). In WackoPicko, there are two stored-XSS vulnerabilities in two pages, one of which requires login and the other which does not, and one in comment fields on a page that requires authentication in Scan-bed. Scanners can exploit the stored-XSS vulnerability by logging in and visiting the vulnerability pages. Then, inject the malicious code into the vulnerable field and submit it to preview and confirm the submission. Afterward, detect the stored-XSS by revisiting the injected pages and analyzing the server response.

The three scanners in WackoPicko without login succeed in all steps but are unable to analyze the response efficiently, resulting in the failure to detect the stored-XSS. The scanners failed to detect WackoPicko with login because the correct attack vectors were not used. Only scanner2 in the scan-bed can complete all scan steps and detect the stored-XSS. The recommendation presented is to improve scanner performance by increasing the use of the right attack vector in the right statute. Scanners can work more efficiently when they use login credentials. The limitation of this paper is that it only improves on previous scanners and focuses on detection rather than prevention of stored-XSS.

Rao et al. (2016) [12] present XBuster, a web browser extension (Mozilla Firefox). XBuster parsed the HTTP request to separate the JavaScript and HTML content and saved them separately as substrings called contexts J and H, respectively. The parameter is split into both contexts in two passes. The first pass, scan from left to right to recognize the first occurrence of "<" and then scan to the first occurrence of ">", all substrings between them define an H context. When found, XBuster continues to examine the rest of the parameter for the next occurrence of an H context. In the second pass, recognizes J contexts in the substring before the first occurrence of H context (if present), in the substring to the right of the last H context, and between two H contexts.

In this method, the browser components are: The Network Interface (NIC) is used to communicate with the server via HTTP requests and responses. The HTML document is analyzed by the Rendering Engine. For the XSS filter, there is a point numbered 1 between the Network Interface and the Rendering Engine. The JavaScript Interpreter is used to analyze and execute JavaScript code. Another 2 point between the Rendering Engine and the JavaScript Interpreter is also for the XSS filter. The user interface contains all of the browser's components. The Browser Engine screens the client's activities and passes them on to the Rendering Engine.

The HTTP response from the server is filtered by the XBuster component (Point 1) to see if there is any HTML injection. By attempting to find a match with each component of H. When a match is found, all special characters in the matching string are encoded. The modified response is sent to the rendering engine for further processing. (Point 2) examine for any JavaScript injection. Before executing it by JavaScript Interpreter, first, check if the response contains JavaScript 's'. If so, compare s with elements in J; if any match, the special character in s is encoded. This method reduces the user's browser's performance.

Kaur et al. (2018) [13] presented a method for scanning XSS attack vectors on cloud-based HTML5 web applications. The method is divided into two routines: "HTML5 Feature Injection" and "HTML5 Feature Comparison" in feature injection phase, The web parser parses the web application and sends the URL links to the DOM Generator, which generates the corresponding DOM tree, during feature injection. The JavaScript features are extracted from the corresponding Dom tree. These features are estimated by the Feature Estimation and Injection component and then injected into the source code. The injected features and source code are saved in the feature repository.

The feature comparison phase operates in such a way that the HTTP request is sent to the server and the browser receives the corresponding HTTP Response. The HTML Parser looks for hidden injection points and forwards them to the JavaScript extractor component, which extracts the necessary JavaScript code. This code is being sent to the components "Feature computation and JavaScript Decoder." Feature Computation computes and sends the features of this JavaScript code to the Feature comparator. These features are then compared to those in the Feature Repository. If the feature comparator or the Similarity indicator detects malicious code, it forwards it to the sanitizer. The complexity of this method is its many components, which include the DOM Generator, Context Locator (this component works with a corresponding specific algorithm to extract the HTML context during feature extraction processing), Similarity Indicator, Feature Estimator, and Sanitizer.

Taha et al. (2018) [14] proposed two methods for preventing XSS attacks using PHP functions. The first method is to use a regular expression to verify input entered by the user into web forms. The second method is another regular expression used to test and ensure that each input may contain a malicious script, so that if the attacker injects XSS script code in the information field, this malicious script is removed and not allowed to execute. To prevent an XSS attack, this method makes use of some built-in PHP language. The main strategy of the algorithm in this method is as follows: the AllowList regular expression list contains the trusted inputs for validation. Another regular expression list, DenyList, is used to determine whether the input contains invalid data and to remove any potentially suspicious characters, such as the start and end tags of HTML "<>" and each text within it. The proposed checking has the disadvantage of removing all executable code that contains a special character.

The German Rodrguez et al. (2018) [15] proposed Cookie Scout as an analytical method to prevent XSS attacks, which is used as a tool by the Browser Exploitation Framework (Beef). The idea is to examine the behaviour of the cookie that is created when a user visits a website, as well as the number of packets exchanged between the victim and the attacker. That is accomplished during the data collection phase. The collected data is analyzed during the data analysis phase. The collected data is saved as a variable in the cookie's parameter in the user's browser. These parameters include the name of the cookies, the created site (the web page the user visits), the creation date, the expiration date, command execution, and traffic between the attacker and the victims.

This method's algorithm consists of a set of operations and conditions. The operations are as follows: each website visited is scanned by the Site Explorer, any cookie created is analyzed by the cookie analyzer to determine the number of cookies created, Extract parameters, and Storage parameters. If a cookie is an execution, the conditions are the difference between the Creation and Expiration dates. If the cookie name ends in "* .js". If any of the conditions is not met, the cookie's reputation is reduced by 10. Finally, if the reputation value falls below 70, the site is blocked. All websites visited by the user are saved in the database, along with their cookies and reputation level. The limitation of this method reduces the browser's performance.

This paper develops a server-side method for detecting and preventing stored XSS attacks. It is a comprehensive method for protecting web application users who do not rely on the practices or tools of a specific server-side language. The method does not prevent all scripts, but only those that have been proven to be malicious, is simple, and reduces the amount of work on the client-side so that it does not impair the user's browser performance.

IV. The Proposed Method

This Section presents our method for detecting and preventing Stored Cross-Site Script (Stored-XSS).

A. Overview

The proposed method is made up of three major components: the user, the web application, and the Prevent stored XSS Server **PSS**.

The user: the person who accesses the web application and attempts to enter information into it. The malicious script may be present in the input; if so, the user is the attacker.

Web application: any application on the web that is presented to users and may contain the vulnerability of stored-XSS.

PSS: The server has been suggested to sterilize the web application's input.

In summary, the proposed method works as follows: when a user attempts to enter any input, whether normal or harmful, it is automatically forwarded to PSS. PSS check that input to see if it contains the malicious script. If the malicious script is found in the input, the PSS detects it and prevents it from being executed by sterilizing it. The sterilized input (new input) is sent to the web application and stored in the database. Refer to Fig. 2.

B. The proposed structure

When a user attempts to enter any input (such as his or her name or a comment) into any web application, the input is sent to PSS. The PSS examines the input to see if it contains any scripts. This is accomplished by comparing the input to a list of scripts (harmlist). When PSS detects a script, examine it to see if it contains the malicious command. This is accomplished by comparing the input to a set of keywords (the malicious commands) (harmlist2). If there is a keyword in the script, it is malicious, and PSS sanitizes the input by deleting the malicious script. PSS sends sanitized input to the web application as new input. Finally, the web application stores the new input in the database and displays the outcome to the user. The proposed structure is depicted in Fig. 3.

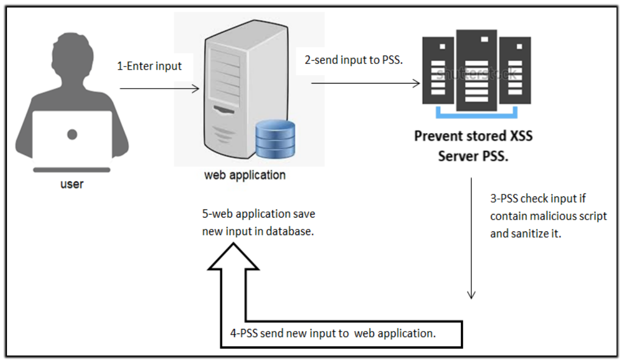

1-Enter input
2-send input to PSS.
user
web application
Prevent stored XSS Server PSS.
3-PSS check input if contain malicious script and sanitize it.
5-web application save new input in database.
4-PSS send new input to web application.

Fig. 2 the proposed method scenario.

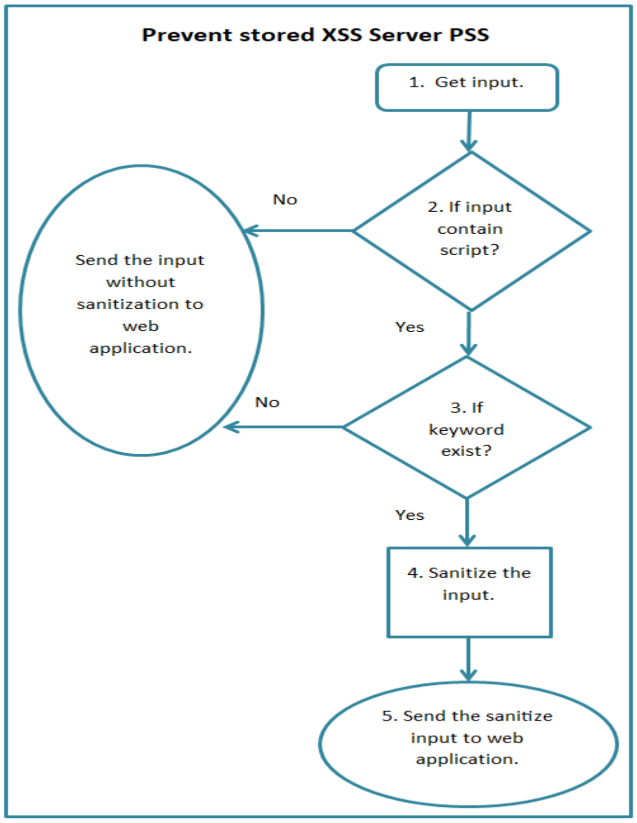

Prevent stored XSS Server PSS
1. Get input.
2. If input contain script?
No
Yes
Send the input without sanitization to web application.
3. If keyword exist?
No
Yes
4. Sanitize the input.
5. Send the sanitize input to web application.

Fig. 3 proposed server structure.

C. Prevent stored XSS Server PSS

The PSS is divided into two phases: setup phase and input verification phase.

First: Setup phase

During this phase, the management page was created. The setup phase consists of two steps:

1- *Administrate step*: include authentication and the creation of the necessary lists.

a- *Authentication*: This step determined which users were authorized to manage the database (harmlists) based on their username and password.

b- *Harmlists creation*: there are two lists

- Harmlist1 includes all scripts that the attacker may use
- Harmlist2 contains all keywords of malicious command (function) that the attacker can use with the script.

The harmlist1 is created in the creation step and contains two fields, the first for the script's start and the second for the script's end. Harmlist2 has two fields: one for keywords and another for the ASCII of these keywords. It should be noted that the attacker may use ASCII instead of the keyword.

2- *Operation step*: Create the update phase in this step to perform any database modifications such as delete, insert, or any other modifications.

Second: Testing Input phase

This phase depicts the work steps in PSS such as the following:-

1- Enter the input (Get input) when the user enters input into the web application and it is forwarded to PSS.

2- Check the input to see if it contains any scripts by comparing the input to harmlist1.

a. If the input contains the start of script (start tag of script function) such as <script>, <image>,….etc.

b. If there are script then find the end of that script (end tag of script function) such as </script>, </image>, >, …..etc.

c. If PSS finds the script, split the input from the beginning to the end of the script and check if the contents of the script contain any keywords (malicious function) by comparing the splitting part of the input by harmlist2.

d. If PSS finds a keyword then the detected script is malicious.

3- When PSS detects a malicious script, sanitize the input by deleting the script and continuing to check the input reminder.

4- If PSS detects no malicious script, send the input unchanged; otherwise, send the sanitizer input as a web application response.

5- Finally, the web application saves the response input in a database and displays the result to the user.

V. Experiential Evaluation

The evaluation is carried out with the assistance of a desktop system equipped with a 1.6 GHz processor, 4 GB RAM, and the Windows 10 operating system. The proposed method's detection and prevention of Stored XSS attacks were tested on a test open-source web application (DVWA). A PHP/MySQL web application that is damn vulnerable is known as a Damn Vulnerable Web Application (DVWA). The objective of DVWA is to test and analyze some of the most common web weaknesses, with varying levels of hardness (low, medium, and high levels), and with a simple, clear interface [16]. This web application is vulnerable to a variety of attacks, including SQL injection, XSS, and others. It was created to assess the vulnerability of these attacks in web applications. The localhost server-XAMPP was used for the evaluation.

Following the establishment, the next step was to configure the DVWA with low-level security and examine the interface of stored-XSS, which contains two text fields named "Name" and "message" that can be injected with an XSS payload. The XSS payload was obtained from (portswigger cheat-sheet) [17] and (Github XSS-payload-list) [18]. The injection was performed by injecting a malicious script into the input text and filtering the input with PSS, resulting in the sterilized input being saved in the DVWA database rather than the malicious input. The works are reiterated at the median and high levels, and the PSS was able to detect the injected script, sanitize the malicious input, and replace it with a sanitizer input. The amount of time required to check and clean the information has no bearing on the speed with which the response can be recovered. Table I shows the duration of the test for each level. The rate time required to check the information is 0.3 seconds.

TABLE I
TIME OF RESPONSE FOR EACH LEVEL

DVWA security levels	XSS-payloads	Response Time
Low level	Name*<script>alert(document.cookie) </script>	0.56 second
	Message*<script>alert("XSS")</script>	0.04 second
Median level	Name*<image src=xx onerror = alert(1) >	0.44 second
	Message*<script>alert(document.cookie) </script>	0.13 second
High level	Name* <ScRiPt> alert ("You have been hacked") </ ScRiPt >	0.46 second
	Message*<img src=nosource onerror = alert (document.cookie)>	0.15 second
average response time		0.3 second

VI. Security Analysis

Theorem 1: PSS prevents Stored XSS in HTML injection.

Proof: Most previous methods attempted to prevent XSS from exploiting vulnerabilities in web application pages and injected HTML. This is accomplished by preventing all special characters from being executed by encoding or deleting them. Hence, the proposed PSS searches for these characters as well as the special word that appeared with these characters in the input. The input is compared to the harmful list, which contains both the beginning and end of tags (special character or special word). Following the comparison, if any matching is found, an examination within a tag is performed to determine whether or not the damage has been proven, and the harmful tags are prevented from being executed. Table II shows a sub-list of HTML tags that can be used in XSS attacks [19]. These tags, as well as others, are used in PSS harm lists.

PSS compares the input to harmlist1, which includes the HTML tags. If the start of the tag exists, then look for the end tag; if it is found, this indicates the presence of a malicious script in the input. Then, as explained later in Theorem 2, cat the script part from the start tag to the end tag and examine that part. If the harm has been proven, then remove all predefined parts (detected malicious script). Deletion is the sterilization process that is used to prevent attacks.

Theorem2: PSS prevents Stored XSS against JavaScript injection.

Proof: PSS does not only find HTML tags, but also searches and tests within these tags, as stated in Theorem 1, because HTML alone does not prove the existence of XSS attacks, but must also search for the harmful code that executes the attacks. As a consequence, the attacker can use the text between the beginning and end of each HTML tag in comparison to another harmful list that contains the keywords of all JavaScript Functions. When PSS finds a match, the input is sterilized by removing all harmful code, ensuring that the entry is free of malicious JavaScript. Table III contains a list of JavaScript functions that may be used with the HTML script for an XSS attack, as well as other functions listed in the PSS harm list [20, 21, 22]. PRS and PSS are both on the harm list.

PSS compares the cut-out part of the input (the text between the start and end HTML tags) to the harmlist, first with keywords and then with ASCII. Since an attacker may use ASCII rather than keywords in XSS payloads, the ASCII field in the harm list is critical. If any keywords or ASCII are found in the script is harmful, PSS sanitizes the script by deleting it.

Theorem 3: Proposed servers (PSS) detect Stored XSS attacks.

Proof: Searching for HTML Tags and JavaScript Functions is a technique for detecting the presence of XSS attacks. PSS is unable to prevent the attack before conducting a script search. The presence of the script does not prove the entry's riskiness. As a result, the PSS checks the script to see if there are any potentially harmful functions. The processing of search for HTML Tags and JavaScript Functions is a method for detecting the presence of Stored-XSS attacks.

In addition to Theorems 1 and 2 already mentioned. First, to prevent stored XSS attacks must be detected. The detection method of stored XSS attacks is a search process for malicious script (HTML tags and keywords). JavaScript functions and HTML events can both be denoted by keywords. As a result, the detection step is as follows:

1. If Input contains the script (searching in harmlist).
2. If the script is a detection, then check to see if it contains keywords (searching in harmlist2).

Keywords could be JavaScript functions or HTML events, as previously stated. Table III contains information on Malicious JavaScript. Table IV shows HTML events [23]. The proposed server in harmlist2 uses these and other events as keywords.

Theorem 4: the proposed method reduces the load on the user's browser.

Proof: To prevent stored XSS attacks from executing in the user's browser, the browser must perform some practice or procedures to prevent these attacks. These procedures include verifying each HTML page before allowing the user to visit it. Also, check for vulnerabilities in web applications to avoid entry with XSS payloads. PSS eliminates the need for the browser to perform any verification or searching methods. The verification corresponding to the proposed method necessitates a collection of comparisons with many lists; all of these lists and comparisons may take time, space, and affect the web browser's performance. Therefore, PSS is given list storage space and performs comparisons away from the user's browser. The proposed servers required time to complete the test and sterilize the input if it is confirmed its injection and noted that it is not affected by the speed of response as mentioned earlier in the experimental evaluation section in Table I. As a result, there is no process in the proposed method that affects browser performance, and the browser does not perform any action or additional work that affects its performance.

TABLE II
HTML TAGS [19].

HTML tags	Meaning	Start script	End script	Encode end in the input
<audio >	embedded sound content	Audio	Audio	Audio
<body>	document's body	Body		
<div>	section in a document	div		
<img>	Defines an image	Img		
<object >	container for an external application	Object	>	&gt;
<svg>	container for SVG graphics	Svg		
<iframe >	An inline frame	Iframe		
<script >	client-side script	Script	Script	Script

TABLE III
LIST OF JAVASCRIPT FUNCTIONS [20, 21, 22].

JavaScript functions	Meaning	Keywords in PSS
getCookie()	Cookie access	Cookie
setCookie()		
location.assign()	Redirect to attacker site	Location
location.replace()		
location.herf		
AppName()	Access to web browser	AppName
getUserAgent()		UserAgent
document.write()	Document content	
document.getElementById,	Access to id	Document
document.getElementByName	Access to name	

TABLE IV
HTML EVENTS [23].

Windows events	Mouse and keyboard events	Other
onload onunload	Onkeydown onkeypress onkeyup onclick ondbclick onmousedown onmousemove onmouseup	Onerror Onload

VII. Conclusion and Future Work

In stored-XSS (persistent) attacks, also known as (direct XSS), the malicious script is presented within the attacker's input in the vulnerable input field of a web application. The malicious script is executed in the victim's web browser, resulting in damage outcomes such as the steal of session data, access to sensitive data, or cookie theft. The injected script is stored inside the vulnerable web page and harms all users who visit the injection web page, making it one of the most dangerous attacks. A method for detecting and preventing the reflected-XSS attack was proposed.

The proposed method for resolving web application vulnerabilities and filtering all user input by deleting all scripts may cause harm. It also considers the time and cost of storage and processing in relation to the user's device. For testing and sanitizing input, a general server called PSS has been suggested. PSS is not intended for a specific type of web service; rather, it can be used by any application to filter its input. PSS has been proven to be successful by evaluating it with a local server (XAMPP) and an open-source application (DVWA), and various XSS payloads have been used to inject the input. The PSS successfully detected the damage in the input, sterilized it, and saved the safe input rather than the harmful input.

Future plans include adding a second factor of authentication to PSS. The possibility of facing the replay is avoided by adding a random number to each request to PSS, such as the date of sending. Merge the PSS with the previously suggested server PRS for detecting and preventing reflected XSS.

Conflict of Interest

The authors have no conflict of relevant interest to this article.

References

[1] A. Marashdiha, Z. Zaabaa, K. Suwaisb, N. Moda "Web Application Security: An Investigation on Static Analysis with other Algorithms to Detect Cross Site Scripting", Procedia Computer Science, Vol. 161, pp. 1173-1181, 2019.

[2] Mustafa H. Alzuwaini, and Ali A. Yassin, "An Efficient Mechanism to Prevent the Phishing Attacks", Iraqi Journal for Electrical and Electronic Engineering, Vol. 17, Issue 1, pp. 125-135, 2021.

[3] A. Marashdih, and Z. Zaaba. "Cross site scripting: removing approaches in web application", Procedia Computer Science, Vol. 124, pp. 647-655, 2017.

[4] Germán E. Rodríguez , J. Torres , P. Flores , D. Benavides. "Cross-site scripting (XSS) attacks and mitigation: A survey", Computer Networks, Vol. 166, 106960, 2020.

[5] S. Gupta & B. Gupta. "XSS-secure as a service for the platforms of online social network-based multimedia web applications in cloud", Multimedia Tools and Applications, Vol. 77, No. 4, pp. 4829-4861, 2018.

[6] C. Lv, L. Zhang, F. Zeng, and J. Zhang, "Adaptive random testing for XSS vulnerability", 2019 26th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2019.

[7] Parvez, Muhammad, P. Zavarsky, and N. Khoury. "Analysis of effectiveness of black-box web application scanners in detection of stored SQL injection and stored XSS vulnerabilities", 2015 10th International Conference for Internet Technology and Secured Transactions (ICITST). IEEE, 2015.

[8] S. Mahmoud, Marco Alfonse, M. Roushdy, A. Salem "Detection of Cross Site Scripting Attacks Model with Deep Transfer Learning", 2020.

[9] Manaa, M. Ebady, and R. Hussein. "Preventing cross site scripting attacks in websites", Asian Journal of Information Technology, Vol. 15, No. 6, pp. 797-804, 2018.

[10] XSS, A comprehensive tutorial on cross-site scripting, Created by Jakob Kallin and Irene Lobo Valbuena, July 9th, 2016. Available from: https://excess-xss.com/ .

[11] S. Gupta, B. Gupta. "Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art", International Journal of System Assurance Engineering and Management, Vol. 8, No.1, pp. 512-530, 2017.

[12] K. Rao, N. Jain, N. Limaje, A. Gupta, M. Jain, and B. Menezes. "Two for the price of one: A combined browser defense against XSS and clickjacking", 2016 International Conference on Computing, Networking and Communications (ICNC). IEEE, 2016.

[13] G. Kaur, B. Pande, A. Bhardwaj, G. Bhagat, and S. Gupta "Efficient yet robust elimination of XSS attack vectors from HTML5 web applications hosted on OSN-based cloud platforms", Procedia Computer Science, Vol. 125, pp. 669-675, 2018.

[14] Taha, T. Assad, and M.t Karabatak. "A proposed approach for preventing cross-site scripting", 2018 6th International Symposium on Digital Forensic and Security (ISDFS). IEEE, 2018.

[15] G. Rodrıguez, D. Benavides, J. Torres, P. Flores, and W. Fuertes. "Cookie scout: An analytic model for prevention of cross-site scripting (XSS) using a cookie classifier", International Conference on Information Technology & Systems. Springer, Cham, 2018.

[16] GitHub. About damn vulnerable web application (dvwa). Jun 3, 2021; Available from: https://github.com/digininja/DVWA.

[17] GitHub. Cross Site Scripting (XSS) Vulnerability Payload List Fep 10, 2021; Available from: https://github.com/payloadbox/xss-payload-list.

[18] PortSwigger. Cross-site scripting (XSS) cheat sheet. 19 Jan 2021; Available from: https://portswigger.net/web-security/cross-site-scripting/cheat-sheet

[19] W3school, HTML Element Reference, 1999-2021, available in: https://www.w3schools.com/html/html_lists.asp

[20] S. Gupta, and B. Gupta. "CSSXC: Context-sensitive sanitization framework for Web applications against XSS vulnerabilities in cloud environments", Procedia Computer Science, Vol. 85, pp. 198-205, 2016.

[21] S. Gupta, and B. Gupta. "Enhanced XSS defensive framework for web applications deployed in the virtual machines of cloud computing environment", Procedia Technology, Vol. 24, pp. 1595-1602, 2016.

[22] A. Sivanesan, A. Mathur, and A. Javaid. "A Google chromium browser extension for detecting XSS attack in html5 based websites", 2018 IEEE International Conference on Electro/Information Technology (EIT). IEEE, 2018.

[23] P. Chen, C. Min, J. Wang "Research and Implementation of Cross-site Scripting Defense Method Based on Moving Target Defense Technology", 2018 5th International Conference on Systems and Informatics (ICSAI). IEEE, 2018.