Journal of Basrah Researches ((Sciences)) Vol. (47). No. 1 (2021)

Available online at: https://www.iasj.net/iasj/journal/260/issues

ISSN –1817 –2695

Received date: 6-5-2021 Accepted date: 18-8-2021 Available online date: 31-12-2021

Proposed Method to Detect and Prevent Reflected Cross-Site Script Attack

Iman Fareed Khazal[1], Mohammed Abdulridha Hussain[2]

[1,2] *Computer Science Department, College of Education for Pure Science, University of Basrah, Basrah, Iraq*

E-mail: imanfared88@gmail.com, mohsubber@gmail.com

Abstract

Due to the widespread use of web applications and the dramatic increase in the number of application users, most web applications contain flaws that make them vulnerable to a variety of attacks. One of the most common attacks is Cross-Site Script (XSS). In an XSS attack, the attacker exploits an XSS vulnerability in a web application and injects a malicious script into it. The majority of preventive measures are client-side, reducing the performance of the web browser. It has been suggested as a server side method in this paper. The Prevent Reflected-XSS Server (PRS) is a suggestion server that checks the domain name of a link's Uniform Resource Locator (URL) to see if it is on the untrusted list of malicious sites. If it does not exist, Check that link to see if it contains any malicious script. If the URL is injected, the malicious URL is replaced with a sterilized URL. This method was tested using an open-source application and was successful in determining the harmful code within the URL and sterilizing it in an average of 0.33 seconds.

Keywords: cross site script(XSS), Reflected(non-persistent) XSS, Web application, sterilized URL, preventing XSS.

1. Introduction

Since the first case of Covid-19 (Coronavirus) disease was reported in China (Wuhan District) in December 2019, the disease has spread throughout the world [1]. The COVID-19 epidemic has prompted a call for the use of novel techniques to mitigate the epidemic's impact. Telemedicine, online education, and telework are becoming increasingly important in assisting society in slowing the spread of the virus [2]. COVID-19 prevalence posed a significant threat to the "technology-driven society," and internet criminals took advantage of this opportunity to expand their attacks [3]. Internet

criminals targeted Hammersmith Medicines Research, which was preparing to test vaccines in the United Kingdom [4].

According to the 2007 report of the open web application security project (OWASP), Cross-site script (XSS) is one of the most dangerous types of web attacks, ranking seventh among the most dangerous types of web attacks. Despite the presence of several technologies to prevent XSS attacks, this attack continues to endanger the security of web applications. As a consequence, it is critical to search for technical XSS attacks and improve web applications[5]. The name cross-site comes from the fact that the attacker uses the interference between two web applications to carry out the attack [6]. XSS attacks exploit vulnerabilities in web applications to inject malicious code into them. This type of attack was chosen because it is regarded as one of the most serious threats to the web application, potentially leading to other types of session hijacking attacks that harm both the user and the web application service provider[7].

There are three types of XSS attacks: stored (persistent), Document Object Model-based (DOM-based), and reflected (non-persistent). The Stored XSS, in this type of attack, malicious code is stored on the server by injecting it into the web application and storing it in the database. When a user visits this website, the web application retrieves data from the database and displays it on the user's browser, after which the web browser executes malicious code [8].

Stored XSS attacks occur on the server side, whereas Dom-based XSS attacks occur on the client side. It happens when the attacker injects malicious code into a link (URL) and sends it to the user's machine "document. url of victim's machine" and tries to convince him to click on it, causing the user's computer to be hacked and the user to become a victim[9].

Reflected-XSS attacks also called (type-1), the malicious script is presented within the victim's request, and then embedded in the return response from the web application that does not use the validation or encoding. The malicious script is executed in the web browser of the victim that leads to damage outcomes, such as steal the session data, access to sensitive data, or theft the cookie. The attacker must persuade the user to click on a link that leads to the Reflected XSS attack in order for this type of attack to succeed. This can be done via e-mail to persuade the user to click on the malicious link or by visiting a website that contains malicious links[6].

This paper focuses on reflected XSS. It is the most common, whereas the Stored XSS attack is the most dangerous because it remains on the website and can harm any user who visits it. The DOM-based XSS attack (Type 0) is a more complicated but less well-known type of XSS attack[10].

This paper proposes a method for detecting and preventing reflected XSS. The proposed method attempts to analyse and test URLs. The goal was to reduce the load on the user machine and provide easy and clear method, so a server was proposed that performs the analysis, testing, and sterilisation. The steps for checking URLs are as follows: first, determine whether the name of the site in the URL is on a list of untrusted sites. Second, check the URL to see if it contains the malicious script, which is accomplished by passing the URL through two levels of verification. In the first level, look for the script tag in the URL's path. If any script is found in the URL path, the second level would then look for keywords within that script. Finally, if the keyword is found, the URL is harmful and contains malicious script; therefore, prevent it by sanitizing the URL from the malicious script and replacing it with a sterilised URL. This approach is straightforward and simple.

The proposed method is a server-side method. It is a comprehensive method for protecting web application users and it is not depend 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 following is how this paper's content is organised: Section 2 discusses the background of reflected XSS attacks as well as the scenario. Section 3 related works. 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 contains the conclusion and future work.

2. Reflected XSS (Non-persistent) attack

This type of attack is also known as a reflected attack because it is carried out as a response to a previous request (URL) that was injected with malicious code, so is not stored inside the web application but is carried out directly in the user's browser. Because reflected XSS may redirect the victim to the attacker's application, it is referred to as indirect[10]. XSS attacks, in general, involve 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[11].

The reflected-XSS scenario is depicted in Figure 1 below:

1. The attacker creates the URL and injects it with a malicious code then sends it to the victim.
2. The attacker deceives the target victim into requesting the URL from the website.
3. The web application or website that returns a response to a victim that includes malicious code.
4. Finally, the victim's browser executes the malicious code script and sends the target victim's cookies to the attacker[12].

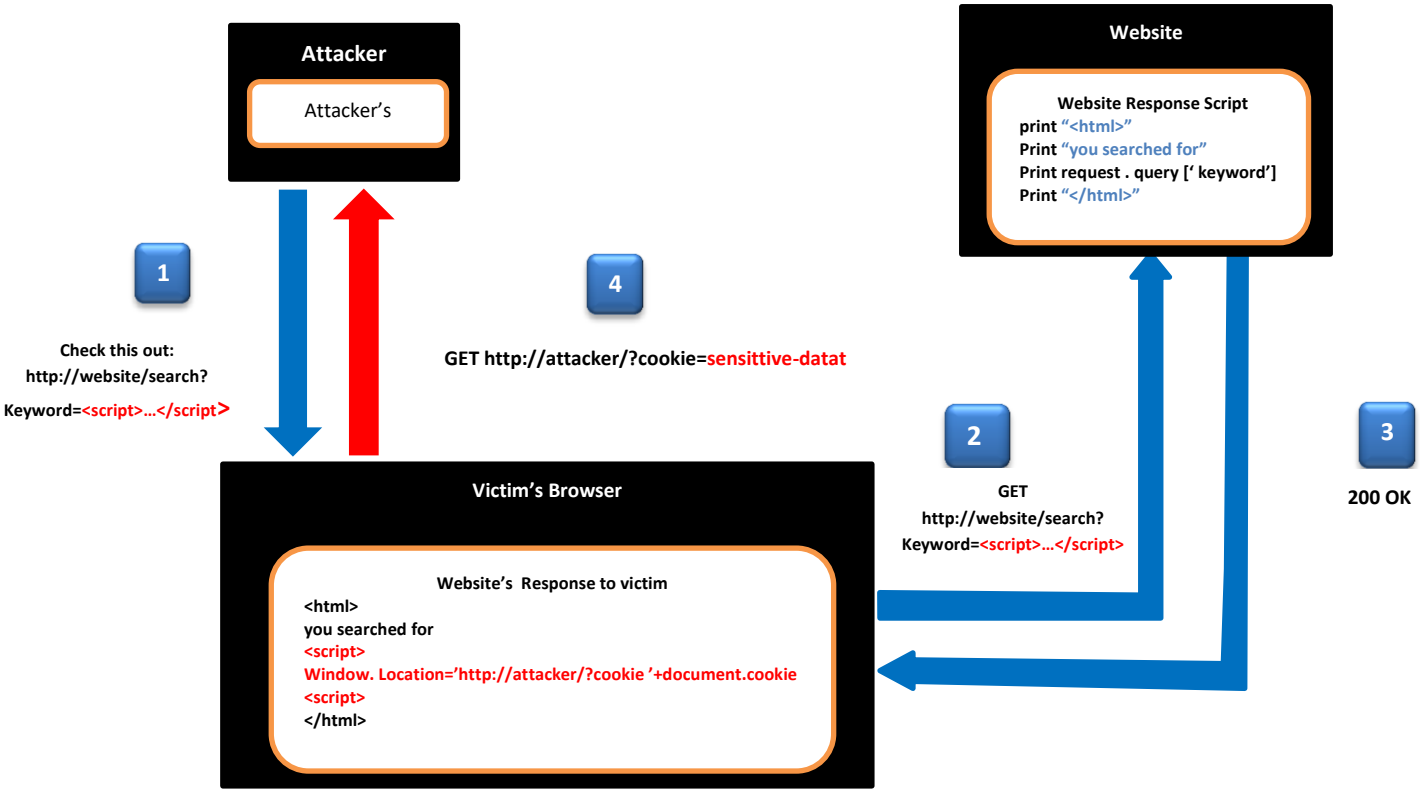

Fig.1: The reflected-XSS attacks scenario [11]

3. Related work

Bukhari and et al. 2018[6] (server-side) concentrated their research on type 1 XSS (reflected XSS). The suggestion is to use technologies and practices for a secure development life cycle (SDL) to reduce the impact of XSS attacks in web applications. The researchers ensured input validation, which was accomplished through a number of practices including check the input format range, length, type, and source of the input. Using encryption on user input to prevent any input from containing executable code and being treated as data only. The significance of built-in validation request technology for developers who use Microsoft ASP.NET. Use the ValidateRequest option to protect against attacks. The web.config file settings can be used to configure it. When this

option is set to “True,” the web application’s inputs are checked for potentially dangerous entries, then the HttpRequestValidationException option is activated and the attack is terminated. Other security techniques, such as AntiXSS and The NET Framework, demonstrated additional encoding options. The disadvantage of this method is that it only works with Windows SDL tools.

Zubarev et al. 2019[13](Server-side) provided a general description of the security problems associated with XSS for web applications related to graphic content by the practical and treatment of XML using PHP framework Larval version 4.2, MySQL, and PHP 7. A set of recommendations and rules were proposed to protect the graphic content on websites. It was stated in the following rules: protect the input and output by using built-in functions in the processor language; it yields the lowest percentage of errors, works faster, and is constantly improved, for example, strip-tags, which is a built-in function in PHP language that removes all PHP and HTML tags.

The researchers also emphasised the use of a white list, which includes safe sources from which to download programmes, images, and styles. In this case, even if the attacker was successful in including the malicious script by redirecting the user to a third-party source, it was not implemented because it will check whether the source is on the white list. Also, coding for each page so that the browser does not detect this coding, which aids in bypassing the filtering phase. One of the researchers' suggestions is to use the HTTP only feature, which prevents cookie access. The disadvantage of this method is that it only targets graphic content, whereas XSS attacks can target vulnerabilities in a variety of ways, not just graphic content.

Shashank et al. 2016[14] (client-side) suggests a method for the cloud environment and called it a context-sensitive sanitization for XSS defensive (shortly called cssxc). This method tries to detect all hidden points for injection XSS in HTML5-based web applications and then sanitizes XSS attacks injected in these points by context sensitive method. The farmworker proposal consists of three parts: Web Application Server (WAS) It is the "injection point locator (IPL)", User It is a cloud user who can access any web application services in the cloud environment, and Detection Server It is a “routine sterilisation injector (SRI) “ Attempts to access JavaScript attack vectors through free XSS attack repositories (RSnake, 2008) then automatically inject sanitizers on the harmful XSS payloads.

This method contains four units, Extraction Units that extract links, forms, parameters, and functions (POST and GET) from an http request. Then it extracts all web links corresponding to that request and refers to all possible injection points. After that, save all information in the repository. Identification Unit responsopel is in charge of discovering all extracted web application links in order to verify injection points and determine which vulnerable JavaScript functions may be injected at this point. The XSS

attack vectors are then automatically sanitized by context-sensitive sanitizers placed in the web source code by the Sanitization Unit. Sanitizers are applied using a list of escape codes (&# followed by several characters). Finally, the Test Unit is in charge of injecting and then executing the sanitized XSS attack at each extracted point of injection. This method is solely for the cloud.

Panja et al. 2015[15] (server-client sides) presented a Buffer Based Cache Check that detects and prevents XSS by utilising both sides (client and server). The server keeps a cache with a trusted sample of the last time the page was rendered, which can be compared to the requested page if they are incompatible. When a user requests a document from a web application, the server searches the cache for it. If the page is found, compare the current state of the page's code to the cached version. If any node does not exactly match that in the cache node, it is isolated with the untrusted nodes. X and Z are code functions that are embedded in the page that is sent to the Mobile Browser. The mobile browser parses the code beginning with function X to see if there are any untrusted nodes, and if so, it compares the text of that node and the DOM location to the content of the whitelist within function X. The whitelist contains all of the code and keywords that are trustworthy and secure (allowed to be executed).

the code is not found in the whitelist, The code is then appended with the .innerHTML property and displayed on the screen as a string. The InnerHTML attribute yieldstThe HTML code's content. Function X sends to the server the following data: the string rendered to the screen, the text of the untrusted node, and the DOM location of the node in the document. The server now searches for the text of that node in the document using DOM location, deletes it, and replaces the old page in the cache with this page. The limitation is oriented only for mobile browsers.

Cookie Scout, developed by the German Rodrguez et al. 2018[16] (client-side), prevents XSS attacks by categorising cookies based on their parameters. The tool made use of the Browser Exploitation Framework (Beef). There are three parts to this method. The first part is data collection, in which variables are defined to be used later in the algorithm. The goal is to analyse the number of sent and received packets between attacker and victim, as well as cookie behaviour. The second section is data analysis, which analyses the collected variables. The values of variables are saved in the cookie parameter on the user's computer. These parameters are the name of the cookies, the created site (the web page the user visits), the creation date, the expiration date, the execution of commands, and the traffic between the attacker and the victims.

The third component is the analytical model (the algorithm); in this component, any website visited and any cookie created are subjected to a series of operations. The operations are as follows: Site Explorer scans all websites visited by the user, Cookie Analyser analyses the cookies for each website visited to determine the number of

cookies created, Parameters for extraction and storage. Finally, a set of conditions was executed to determine the suspicious website. These conditions are the difference between the creation date and the expiration date, if a cookie is an execution, and if the end of the cookie name is "*.js". if each condition is not met, the value of the cookie's reputation is deducted by ten points, and the site is stored in the [BlockSite] if the cookie's reputation value is less than 70. As a result of this algorithm forming a knowledge base, all websites visited can be located in the database along with their cookies and level of reputation. Because it is dependent on the values of the cookie's parameters stored on the client's computer, this method reduces browser performance.

This paper develops a server-side method for detecting and preventing reflected 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 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.

4. The proposed Method

The proposed method for detecting and preventing reflected XSS attacks in web applications and preventing their execution would be presented. The proposed method tests the URL by classifying it as untrusted sites or not, then checks the URL to sterilise it from harmful code before allowing the user to enter the URL site, and finally, the sterilised URL is sent to the user. Figure 2 depicts the overall concept of the proposed method.

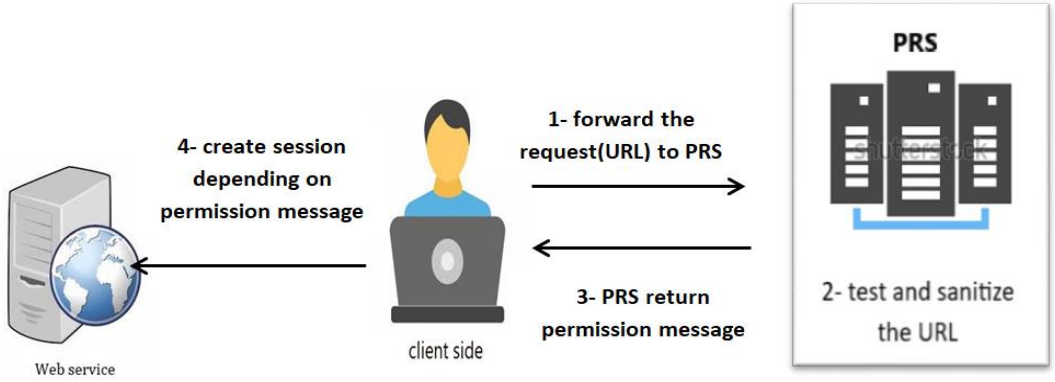

Fig.2 scenario of the proposed method.

There are three entities in this method: the Client (the user browser), the Prevent Reflected-XSS Server (PRS), and the Web Service.

Client: The user of a web application who can send a request (URL) to any web application for a specific purpose.

Prevent Reflected-XSS Server (PRS): This server is a model (web application) used to test the URL requested by the user to determine if the URL Site is untrusted or not and then sanitize it from malicious code if exists.

Web Service: It is the web application (URL Site) that the user has requested.

4.1 Proposed Architecture

The user sends the request (URL) to any web application. The URL was forwarded to **(PRS)**. The PRS firstly checks it by (**classifier**) to determine if it is untrusted sites. By comparing the URL's domain to a list of untrusted sites (Untrusted List). If it is on the untrusted list, the user would be presented with a page containing a message allowing him/her to access the sanitizer URL; this message is referred to as a permission message. The sanitizer URL is taken from the Untrusted List's "sterilised URL" field, but if the URL is not in the Untrusted List, it is sent to (Sanitizer). The URL would be tested in the Sanitizer to see if it contains malicious code. This is accomplished by comparing the URL to a list of malicious code (Harmful list). If any script code is found, examine it to see if it contains any malicious command is inside that script. If a malicious script is discovered, the domain of the URL should be added to the untrusted list in the "malicious URL" field. The malicious script are then removed from the URL and added to the untrusted list's sterilised URL field. The URL has been sterilised. Finally, the user is permitted to access the sanitizer URL via a permission message. The proposed architecture is depicted in Figure 3.

4.2 Prevent Reflected-XSS Server (PRS)

The PRS is divided into two phases (Setup phase and URL Checking phase), as shown below:

First: The setup phase

A Prevent Reflected-XSS Server (PRS) administration page would be developed for this phase, which requires two steps:

1- Manage step: this step includes the following:

a- Authentication step: this is accomplished by entering the authorised users' username and password, allowing them to perform any maintenance or management on the untrusted and harmful lists.

- Untrusted list: it is a list of insecure sites that is initially empty and can be filled with the sites of all malicious URLs detected by PRS.
- Harmful lists: two lists were included the first list contains all of the script functions that an attacker can use, and the second list contains all of the keywords that can be used within the script functions.

b- Creation of untrusted List: Here a list of insecure sites created, which is made up of several fields, such that the first field dedicated to the name (name of the site) and the second field to the site's IP, due to the presence of some attack methods that use the site's IP instead of the explicit name, the third field for the sanitizer URL.

c- Creation of Harmful List: the first list includes two fields one specify to all the keywords the second field for ASCII to the keyword because sometimes the attacker uses ASCII in the malicious code instead of the keyword. The second lists contain also contains two fields, one for the word that indicates the beginning of the malicious script, and the other for the end of that script.

2- Operation step

a- The update page: any modification, such as changing information, adding or removing a field from one of the databases, or any other modification, can be made on this page.

b- Adding a user: here a new user is added who has been approved to be allowed to make any administrative changes.

c- Finally, logout page.

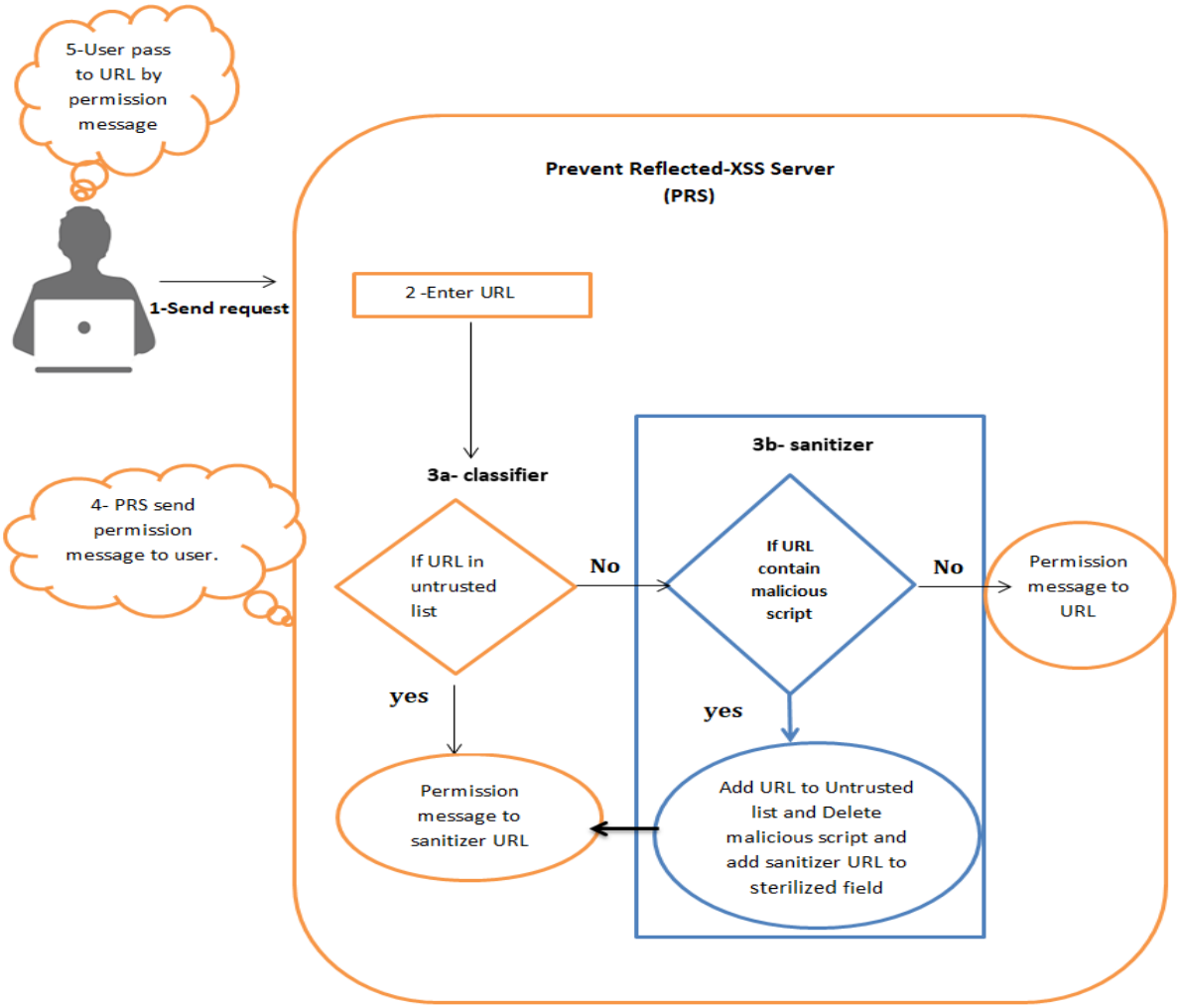

Fig.3 the proposed architecture.

Second: URL Checking phase

This phase describes how PRS checks URLs using the following steps:-

1- Enter the URL (web page). When a user enters a request (URL), it is forwarded to PRS.

2- PRS test the URL.

3- Proposal

- Classifier step

a. Extract website name (the domain of URL) or IP.

b. Check that website with the untrusted list.

i. If exist within the untrusted list: obtain the sanitizer URL from the sterilised field and send it to the user via a permission message.

ii. If none exist, proceed to the sanitizer step.

- Sanitizer step

i. Separate the URL path; the URL path begins after the website name.

j. Compare the URL path to the harmful list (by testing the path for the presence of a script tag (script function) and a keyword within that script).

k. When determine whether or not the URL contains a script:

i. If any malicious script detects in the URL insert the domain name of the URL into the untrusted list and prevent the user to accesses that URL by sanitizing the URL, then send a permission message to the sterilised URL to the user.

ii. Otherwise, send a URL permission message to the user.

5. Experimental Evaluation

The evaluation is carried out using a desktop system equipped with a 1.6 GHz processor, 4 GB of RAM, and the Windows 10 operating system. On a test open-source web application, the proposed method's detection and prevention of Reflected-XSS attacks were evaluated. A Damn Vulnerable Web Application (DVWA) is a vulnerable PHP/MySQ L web application. The goal of DVWA is to exercise some of the most common web weaknesses at various levels of difficulty (low, medium, and high), using a simple and straightforward interface. This was accomplished through the use of localhost server-XAMPP [17].

Following the installation, the DVWA was configured with low-level security and a malicious XSS script was injected into it. The malicious XSS script was obtained from the (portswigger cheat-sheet)[18], and the (Github XSS-payload-list)[19]. Then, PRS checked the injection URL of DVWA, and the result was the permission message to the sanitizer URL.Then, with a medium and high-security level, the PRS detects the malicious script and prevents execution by sanitizing the injection URL and replacing it with a sanitizer URL. The time required to test and sanitize the URL is minimal and has no effect on the speed with which the site page is retrieved. Table 1 shows the duration of the test for each level. The test URL takes 0.33 seconds on average.

Table1 the response time of three security levels

DVWA security levels	XSS-payloads	Time of test
Low	<script>alert(document.cookie)</script>	0.27 second
	<image src=xx onerror=alert(1)>	0.22 second
Median	<ScRiPt>alert("You have been hacked")</ ScRiPt >	0.51 second
	<SCRIPT>document.location.href="https://google.com/"</SCRIPT>	0.3 second
High	<img src=nosource onerror=alert(document.cookie)>	0.49 second
	<body onload= alert(document.cookie)>	0.21 second
The average time		0.33 second

The proposed method has been validated by using it in the DVWA application rather than comparing it to the other studies. All previous methods aimed at protecting the user worked on the client-side, whereas the proposed method in this paper works on the server-side. As a result, comparing the various methods is difficulit.

6. Security analysis

Theorem 1: PRS prevent reflected-XSS in HTML injection.
Proof: The majority of 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 executing by encoding or deleting them. Thus, the proposed PRS search is based on these characters and the special word associated with these characters in the URL. The URL is compared to a list of harmful that includes both the beginning and ending of tags (special character or special word). If any matches are found after the comparison, then the damage is determined and the harmful tags are prevented from being executed.

Theorem 2: PRS prevent reflected-XSS against JavaScript injection.
Proof: Because HTML tags alone does not prove the existence of the reflected-XSS attack, PRS searches and tests within these tags, as it was described in the Theorem 1. However, it is likely that PRS is looking for the malicious code that's used to carry out

the attack. So that PRS compare the text between the beginning and end of each HTML tag with another harmful list that contains the keywords of all JavaScript Functions may be used by the attacker. If PRS detects a match, the URL is sterilised by removing all potentially harmful code, ensuring that the link is free of malicious JavaScript.

Theorem 3: PRS detect reflected-XSS.
Proof: The detection of Reflected-XSS attacks is accomplished by searching for HTML Tags and JavaScript functions. In addition, if any harmful code is discovered within the link, it is added to the untrusted list. That is to warn the user if he/she attempts to access this link again in the future and to redirect the user to the sterilise link. This demonstrated the PRS's capability of detecting reflected-XSS inside links and storing these links in a list containing all sites containing vulnerabilities that can be exploited by the attacker to carry out the Reflected-XSS attack.

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

Proof: To prevent a Reflected-XSS attack from executing in the user's browser, the browser must follow some practices or procedures. These procedures include verifying each link before allowing the user to click on it. The verification corresponding to the proposed method necessitates a collection of comparisons with many lists, all of which take time, space, and affect the user browser's performance.Therefore, the PRS that provides storage space for lists and performs the comparisons away from the user's browser. PRS required time to complete the test and sterilise the link is measured, and it is noted that it is unaffected by the speed of response as mentioned earlier in Experimental Evaluation.

7. Conclusion and Future Work

The malicious script is presented within the victim's request (URL) in reflected-XSS (Non-persistent)attacks, also known as (type1). The malicious script is executed in the victim's web browser, resulting in damage outcomes such as session data theft, access to sensitive data, cookie theft, or victim redirection. This is therefore one of the most dangerous and most frequent attacks. A proposed method for detecting and preventing the reflected-XSS attack was proposed.

The proposed method focuses on protecting the user rather than covering the vulnerability in the web application. It also considers the time and cost of storage and processing in relation to the device of the user. Because the method involves many comparisons and database connections, a server called PRS has been proposed to perform all of these comparisons and tests in order to reduce effort while not affecting the power of the user's device.

PRS examines each user request (URL) to see if it falls into the untrusted list, then checks to see if it contains a malicious script before sterilising the URL and sending it to the user. PRS is evaluating using a local server (XAMPP) and an open-source application (DVWA), and various XSS payloads have been used to inject the URL. PRS was successful in detecting the damage in the URL, sterilising it, and redirecting the user to a safe URL that was free of any malicious script.

Future work aims to provide a second factor for PRS authentication. By adding a random number to each request to PRS, such as the date of sending, the possibility of facing a replay is eliminated. The other method has been added to detect and prevent stored XSS.

References.

1. S. Lalmuanawma, J. Hussain, and L. Chhakchhuak. "Applications of machine learning and artificial intelligence for Covid-19 (SARS-CoV-2) pandemic: A review." Chaos, Solitons & Fractals 139 (2020): 110059. 2020.

2. Wu He, Z. Zhang, and W. Li. "Information technology solutions, challenges, and suggestions for tackling the COVID-19 pandemic." International journal of information management 57 (2021): 102287. 2021.

3. H. Lallie, L.Shepherd, J. Nurse, A. Erola, G. Epiphaniou, C. Maple, and X. Bellekens. "Cyber security in the age of covid-19: A timeline and analysis of cyber-crime and cyber-attacks during the pandemic." Computers & Security 105 (2021): 102248. 2021.

4. M. Milanovic, and M. Schmitt. "Cyber attacks and cyber (mis) information operations during a pandemic." J. Nat'l Sec. L. & Pol'y 11 (2020): 247. 2020.

5. P. Chen, C. Zhao, C. Yu, C. 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.

6. S. Bukhari, M. Dar, and U. Iqbal. "Reducing attack surface corresponding to Type 1 cross-site scripting attacks using secure development life cycle practices." 2018 fourth international conference on advances in electrical, electronics, information, communication and bio-informatics (AEEICB). IEEE, 2018.

7. A. Marashdih, Z. Zaaba, and H. Omer. "Web security: detection of cross site scripting in PHP web application using genetic algorithm." International Journal of Advanced Computer Science and Applications (IJACSA) 8.5 (2017). 2017.

8. M. Parvez, 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.

9. S. Mahmoud, M. Alfonse, M. Roushdy, and A. Salem, "Detection of Cross Site Scripting Attacks Model with Deep Transfer Learning". 2020.

10. G. Rodrıguez, J. Torres, P. Flores, D. Benavides. "Cross-site scripting (XSS) attacks and mitigation: A survey." Computer Networks 166 (2020): 106960. 2020.

11. XSS, E., A comprehensive tutorial on cross-site scripting, July 9th, available Available from: https://excess-xss.com/, 2016.

12. 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 8.1 (2017): 512-530. 2017

13. D. Zubarev, and I. Skarga-Bandurova. "Cross-Site Scripting for Graphic Data: Vulnerabilities and Prevention." 2019 10th International Conference on Dependable Systems, Services and Technologies (DESSERT). IEEE, 2019.

14. S. Gupta, and B. Gupta. "CSSXC: Context-sensitive sanitization framework for Web applications against XSS vulnerabilities in cloud environments." Procedia Computer Science 85 (2016): 198-205.

15. P. Biswajit, T. Gennarelli, and P. Meharia. "Handling cross site scripting attacks using cache check to reduce webpage rendering time with elimination of sanitization and filtering in light weight mobile web browser." 2015 First Conference on Mobile and Secure Services (MOBISECSERV). IEEE, 2015.

16. G. Rodrıguez, D. Benavides, J. Torres, P. Flores, and W. Fuertes I. "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.

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

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

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