INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2025, VOL. 71, NO. 1, PP. 101-106
Manuscript received March 14, 2025; revised March, 2025. doi: 10.24425/ijet.2025.153548

Detection of XSS vulnerabilities in OJS

Serhii Buchyk, Ruslana Ziubina, Tetiana Yuzhakova, and Anastasiia Shabanova

***Abstract*—This article analyzes XSS vulnerabilities in OJS (Open Journal Systems) and develops a model for protecting against these attacks. It discusses different types of XSS attacks, vulnerabilities in OJS, methods of detecting them, and potential consequences for system security. The article describes a specific vulnerability that can be exploited to inject malicious code through user input of specially generated data. Based on the analysis, a protection model is developed, which includes the introduction of restrictions for vulnerable fields, encoding, and filtering of data before displaying it on the page. This article is essential for OJS administrators and developers to ensure high security and protection against potential XSS attacks.**

***Keywords*—OJS (Open Journal System); vulnerabilities; XSS; website; web applications; XSS attacks; detection of vulnerabilities**

I. Introduction

XSS (Cross-Site Scripting) attacks pose a significant cybersecurity threat due to their widespread use and serious consequences. Unlike phishing, which manipulates individuals to disclose sensitive information through deception, XSS attacks target web application vulnerabilities. These attacks involve the injection of malicious scripts into legitimate websites, which are then executed in users' browsers, resulting in a variety of malicious activities such as cookie theft, session hijacking, corruption of web pages, and the spread of malware.

The prevalence of XSS attacks is evident when attackers exploit weaknesses in web forms, URLs, and input fields to deliver malicious scripts. These scripts can manipulate user sessions, steal sensitive data, or compromise the functionality of the affected website. XSS attacks are particularly insidious because they target unsuspecting users who visit compromised or maliciously crafted web pages, often without their knowledge.

The seriousness of XSS attacks is underscored by their potential impact on user privacy, financial security, and the reputation of affected websites. For companies, organizations, and individuals, the consequences can range from financial losses and regulatory sanctions to loss of brand credibility and deterioration of customer relationships [1].

URL spoofing and following redirects to phishing pages are particularly dangerous. The authors describe the detection of phishing attacks in [2], and possible methods for determining the degree of suspicion of a phishing address in [3], which is a separate area but interrelated to the article's subject.

Eliminating XSS vulnerabilities requires a multifaceted approach, including strict input validation, source coding, secure coding methods, and regular security checks. By implementing these measures, web developers and administrators can reduce the risk of XSS attacks and protect their systems and users from malicious exploitation.

II. Research Objectives

This study aims to analyze and identify cross-site scripting (XSS) vulnerabilities in the Open Journal Systems (OJS) platform.

The research focuses on the process of detecting XSS vulnerabilities, with particular emphasis on methodologies employed for their identification. Accordingly, the primary objective is to uncover previously unreported XSS vulnerabilities within OJS and develop a comprehensive protection model to mitigate XSS attacks.

III. Experimental Evaluation

XSS attacks are divided into three main types. The first is Stored XSS (Figure 1). This attack occurs when an attacker injects malicious code into form fields on a website, and then the malicious data is stored on the server. For example, the comment form fields do not have validation, and the attacker enters the script code into the field and submits the form. The comment is then stored in the database and can be displayed on the blog page in the list of all comments. When other users browse the blog page, the malicious code contained in the comment will run, and sensitive user data can be sent to the attacker.

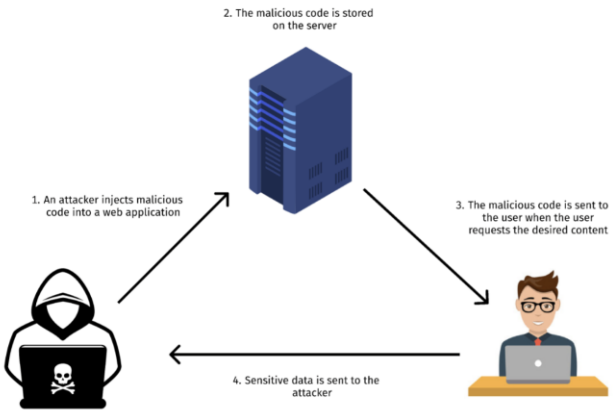

Fig. 1. Stored XSS

The following type of attack – Reflected XSS (Figure 2).

First Author, Third Autor, and Forth Author are with the Taras Shevchenko National University of Kyiv, Kyiv, Ukraine (e-mail: buchyk@knu.ua, tetiana.yuzhakova@knu.ua, shabanovaa@knu.ua).

The Second Author is with the University of Bielsko-Biala, Bielsko-Biala, Poland (e-mail: rziubina@ubb.edu.pl).

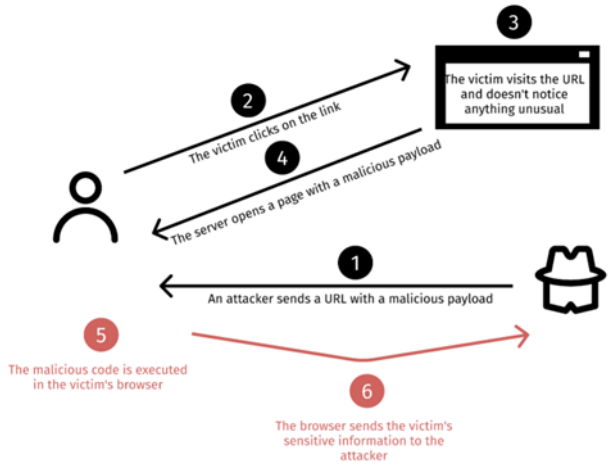

Fig.2. Reflected XSS

Such an attack occurs when an attacker sends a link to a user that contains additional data in the URL that will allow malicious actions to be performed as soon as the user opens the link. For example, a website has a search form and when entering data into the search field, a search parameter (?search=example_text) is added to the page URL. Instead of the search text, an attacker can add a script to perform a specific malicious action when the user clicks on the link. Attackers usually add such links on social networks, forums, and emails, so do not click on unknown links.

The next type of attack is DOM XSS (Figure 3). This attack occurs when an attacker uses existing layout elements on a page to change the DOM structure of the page, while simultaneously executing malicious code. For example, an attacker analyses the code of a web page and notices that when entering text into a search, the text "You are searching" is displayed, as well as the search text in the form of code. Thus, the user fills out the search form on the page, adds a script containing malicious actions, and when clicking on the search, the search text is displayed in the form of code, and the malicious script is launched.

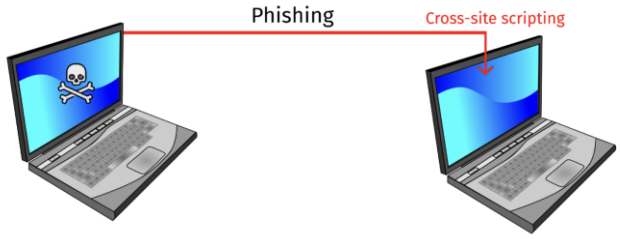

Fig. 3. DOM XSS

In recent years, many tools have been developed to detect XSS vulnerabilities in web applications — for example, XSS Hunter, XSStrike, XSSER, and others. However, DOM XSS vulnerabilities remain the most difficult to detect, as the DOM structure of a website is usually not static, and some elements can be changed, added, or deleted dynamically. JavaScript code can be very large, which means that it takes a lot of time to analyze it. Also, DOM XSS is difficult to detect because these attacks are performed only on the client side, and the results of malicious code execution are not transmitted to the server.

Let's look at some examples of large-scale XSS attacks and the consequences they caused.

An XSS worm (cross-site scripting virus) infected more than 1 million MySpace user profiles in just 1 day. The worm spread exponentially. In 2018, British Airways was subjected to an XSS attack from the Megacart hacker group. Using the website, hackers tried to steal confidential customer data. In 2008, during Barack Obama's election campaign, a hacker found an XSS vulnerability on the politician's website. Using this vulnerability, he made sure that everyone who visited the website was redirected to Hillary Clinton's page. However, the politician's team eliminated this vulnerability, which was contained in one of the forms on the website, in a few hours [4].

In 2011, the CIA (US Central Intelligence Agency) suffered an XSS attack on their website. The attack was carried out by an Indian hacker who penetrated the website and damaged it. In 2020, Amazon Alexa was attacked due to incorrect CORS configurations. When the attacker exploited this vulnerability, he gained access to CSRF tokens, which allowed him to use these tokens as user accounts and act on their behalf in the system [4].

In 2019, the North Carolina healthcare system detected a customer data leak due to an XSS attack. White hackers have discovered XSS vulnerabilities in many Internet companies, such as Google and Amazon, but the companies have already fixed them. Also, Tesla had a Stored XSS vulnerability, which was discovered by a white hacker who received a $10k reward. He used the XSS Hunter tool, which proves that if hackers use similar tools, they can quickly find vulnerabilities in the right web applications and use them for their own purposes [4].

The next few paragraphs of the article will focus on discussing methods for detecting XSS vulnerabilities in websites that contain a large amount of JavaScript code. The methods shown in Figure 4 can simplify the process of vulnerability detection.

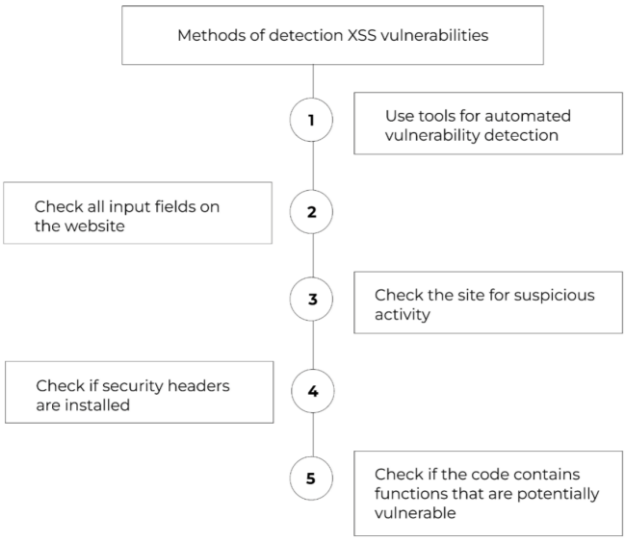

Fig. 4. Methods of detection XSS vulnerabilities

Many studies, for example [5], focus on XSS detection using artificial intelligence, but this work focuses on manual XSS vulnerability detection, which will allow website developers to easily identify vulnerabilities in a web application.

Let's start with XSS vulnerability scanners. As mentioned earlier, there are many programs for scanning a web application for XSS vulnerabilities. XSS Hunter [6] is a tool that can be used either in a web version or installed on Linux-like systems to run through a console. To use this tool, you need to specify the domain name of your web application, specify the email address to which you will receive notifications about the vulnerabilities found, and wait for the scan results. What are the features of this utility? In the control panel, you can see all the payloads that are sent to your website for testing. Also, when vulnerabilities are detected, a notification is sent to the email about the page where the vulnerability was detected, the HTTP request that was sent, the entire HTML code of the page, and a screenshot of the web page where the XSS attack was performed.

The next tool is XSStrike. Unlike the previous scanner, this scanner can only be run on Linux-like systems via the console. This scanner can show the status of the WAF (Web application firewall), the entire page scan, along the parameters that were checked. If the WAF is enabled, information is additionally added about whether the firewall has blocked the sending of payloads to a form field or page URL. If you do not have a WAF configured on your website, it is worth enabling it, as it will additionally protect your site from some XSS attacks. For example, if your website has a search field and an attacker enters a script into the search field and sends a request to search for such data, the WAF will block this request, and the attacker will not be able to use the search field to perform an XSS attack.

At the very least, if web application developers scan their applications with scanners, they will receive information about some of the XSS vulnerabilities found. However, you shouldn't assume that the site has only these vulnerabilities found by the scanners, as scanners cannot find all vulnerabilities, especially if it is DOM XSS. Scanning is only the first method for detecting vulnerabilities.

The next method is to check all input fields on the website. Input fields include contact form fields (name, phone number, email, etc.), search field, registration and login form fields, feedback or comment field, and other various forms that a user may fill out. According to OWASP [7], a tester should go through three stages of field testing. The first one is to search for all the places where the user can enter data, in our case, these are the fields of all the forms on the website. The input data can include various HTTP request parameters, POST data, and various hidden form fields that are invisible to the user but pass this data to other places. Usually, Developer Tools are used for this stage, which are available in every browser. Thanks to this tool, the tester can see the entire existing DOM structure, and determine how a particular form is processed on the site. In the second stage, the tester must check the vulnerability of the web application to common XSS attack vectors by filling in the form fields with a special code that will show immediately whether the field is vulnerable. Special code is code that is not malicious, for example, <script>alert('test')</script>. Vulnerability testing code can be generated by online resources (web application fuzzer). OWASP describes how to test and what code to use for testing. In the third stage, it is necessary to assess the impact of the found vulnerabilities on the security of the web application. To do this, the tester checks the HTML code for incorrectly coded, modified special characters. It is very important that all special characters, such as < > " &, are encoded with sequences that are written in the HTML documentation In JavaScript code, line feeds, apostrophes, double quotes, backslashes, and others must be encoded and escaped.

To test fields, you usually use JavaScript code that includes the alert() function to see the result immediately. That is, if you enter a script with the alert function in the field and send this data to the form, then if this data is displayed immediately on the page and a pop-up window with the alert message is launched, it means that the field is vulnerable and validation is not applied to it to remove all tags that indicate scripts.

Examples of payloads for validating input fields are described in [8]. This paper describes that three variants of the input data need to be tested. The first is the basic option, which involves entering the script tag with the code to be executed, for example, <script>alert(1)</script>. The next option is to analyse the HTML code for attributes that can be used to exploit vulnerabilities. For example, if HTML markup is allowed in the comment field, then for tags that display headings or paragraphs, you can add the onclick attribute and add code that should be executed when you click on this text, for example, alert('xss'). When the comment is published, an alert window will be displayed when you click on the text for which the onclick attribute has been set.

The next step is to check all attributes for links to third-party services. For example, if you add a third-party script connection to a comment using the script tag, a script from a third-party service can run when you click on the text where the script was connected. That is, it is worth checking the entire site for links to unknown services.

Also, [8] discusses potentially dangerous constructions in HTML code. These include the href, src, content, data attributes, and attributes that allow you to run javascript code (onclick, onerror, and others). These attributes are dangerous, especially when the data entered by the user is automatically transferred to the site. Thus, if the data is not validated, the user's malicious code can be executed.

In addition to potentially dangerous attributes, you need to test as many variants of entering code into form fields on your website as possible. For example, if the developer sets up a check for the <script> tag, then if an attacker enters the same tag but with uppercase and lowercase letters, for example, <ScRiPt>, the code that filters the entered data may not remove this tag, and thus the malicious code may be executed. Therefore, you should pay attention to such variants of the entered data.

Let's consider the next method of detecting XSS attacks - detecting suspicious activity on the website. If a WAF is installed, it can block all suspicious activity, which includes the introduction of potentially dangerous constructs on the site. A WAF is usually installed on a server and contains a file with logs that can be used to track the activity that has been blocked. Therefore, if a WAF is installed, it is worth reviewing how often

and what kind of activity it blocks, so you can find out whether this activity is widespread and which elements of the site it is directed at.

If you don't have a WAF that automatically blocks suspicious activity, you should periodically check the database for html and script tags that may be stored after users fill out forms on the site, especially if these forms are not properly validated. If you find tags in the database, you should pay attention to the validation of the data entry fields in the form, as well as check the functions that will display this data on the user's screen so that you do not accidentally run a script that is already stored in the database. That is, you need to filter and validate data when sending form data to the database, as well as when displaying data on the user's screen.

The next method of detecting XSS vulnerabilities is to check the security headers set. XSS attacks are often aimed at stealing user session cookies. To get this data, you need to execute the document.cookie JavaScript code. However, since access to cookies via JavaScript is not always necessary for a website, a security method has been developed to prevent cookie hijacking. To do this, you need to check the HttpOnly box. Content-Type header - used as an indicator of the original type of media file, before any encoding is applied to it. If you set the header value incorrectly, for example, an image can be interpreted as HTML code, which will make your web resource vulnerable to an XSS attack [9, 10]. That is why it is important to check whether these headers are configured on the server to prevent some XSS attacks.

The last method is to check the code for potentially dangerous functions that can be used by an attacker to inject his code. Such JavaScript functions include the innerHTML, alert, document.write, document.location, and other functions that can display the HTML layout on the page or redirect to the page that is passed to the function as a parameter. It's also important to check the HTML markup, which may contain attributes that allow you to execute a script when you click on an element or when an error occurs. For example, the onclick, onerror, and onfocus attributes. It is advisable not to use these attributes in the layout of the site, it is better to run functions through a separate JavaScript code, so an attacker will have fewer ways to inject malicious code [11].

To search for vulnerabilities in OJS, we installed Ubuntu 22.04 LTS and installed OJS (version 3.3.0-14). The OJS settings set a list of tags that are not allowed to be used (a[href|target|title], em, strong, cite, code, ul, ol, li[class], dl, dd, b, i, u, img[src|alt], sup, sub, br, p), but not all fields are checked for these tags. Only the fields with an editor are checked, i.e. those that allow you to add additional text formatting. This was found out by entering tags in the fields of various forms in the web application.

While checking the web application for XSS vulnerabilities, we found the following vulnerability in the Issue section, which was not mentioned on the OJS website. The essence of the XSS vulnerability is that when creating an Issue in OJS, you can enter any data into the fields, so for testing purposes, the text <script>alert();</script> was entered into almost all fields where there was no validation, and then the Issue was saved (Figure 5).

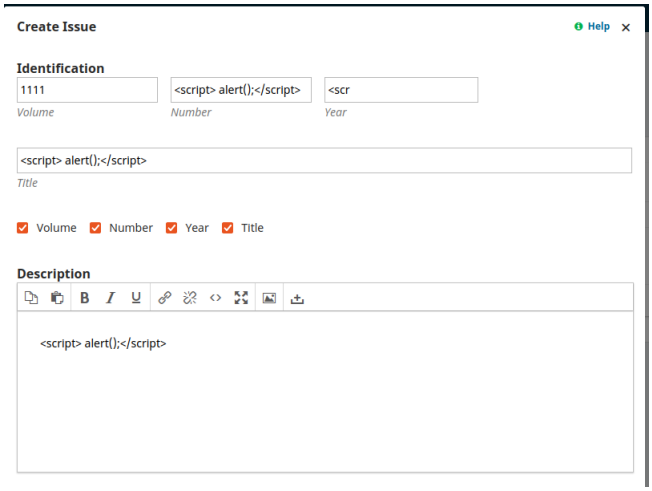

Fig. 5. Filling the fields with the text <script>alert();</script>

After saving, the created Issue was displayed on the page with other created records (Figure 6).

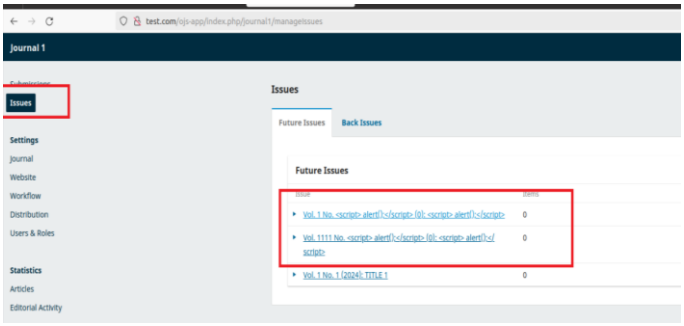

Fig. 6. List of created Issues

If you click on the name of the created Issue, the alert() script is run several times, indicating that the system has a vulnerability stored XSS (Figure 7).

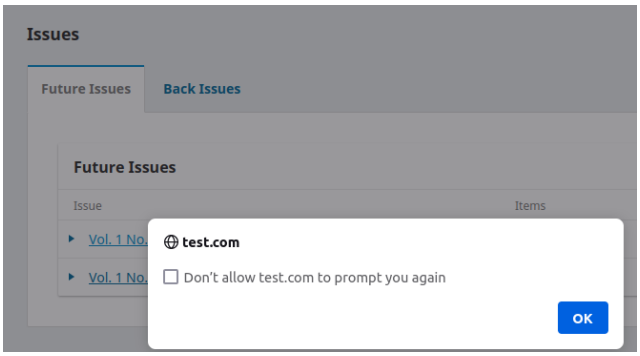

Fig. 7. Executing the alert() script

To check which fields have a vulnerability, we added text to the alert() function for each field, for example, for the number field we added alert('number'), and for the title field we added alert('title'). Accordingly, after creating a new Issue and clicking on the Issue name, a pop-up window with the text 'number' was launched, and then a pop-up window with the text 'title' was launched, but no other alert pop-ups were displayed. That is, it

is the Number and Title fields that have an XSS vulnerability.

The Number field has a character limit of 40 characters. However, although this field must contain a number, you can write both letters and numbers into it. An error message is not displayed if you enter the characters > < and others. That is, you can enter the text of the script, as long as it is up to 40 characters in size. For the Title field, there are no restrictions on the number of characters or the type of data entered, and there is no check for scripts or prohibited characters.

After clicking the Save button, the data is transferred via a POST request. If you look at the form in which the data is transmitted in the POST request (Figure 8), you can see that in some fields the < > characters are encoded, and in some, they are not. These characters are encoded only in the Description field, which is made in the form of an iframe and has validation of the entered data. If the characters were encoded in all fields, there would be no XSS vulnerabilities.

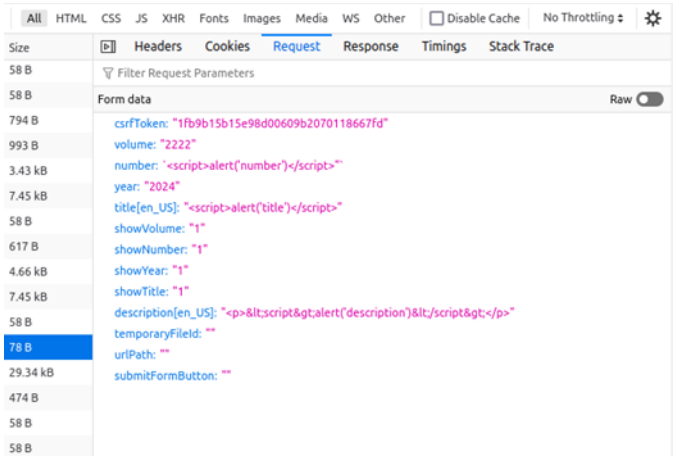

Fig. 8. The content of the POST request

In addition to the Stored XSS vulnerability, a DOM XSS vulnerability was also found. If you enter the text "><script>alert('in number')</script> in the Number field, it will lead to another XSS attack, since the value attribute records the value of the field, using double brackets you can close the value and the input tag, and insert the script outside the input field so that the input field code and script will look like this:

<input type="text" maxlength="40" class="field text" name="number" value=""><script>alert('in number')</script>">

In this case, the script that was inserted outside the input tag will execute first and open the alert window with the text "in number", and the next script will execute alert('title'), which will open the alert window with the text "title".

The main problem, in addition to the lack of validation of the entered data, is the dynamic output of the Number and Title fields in the form of HTML code. That is, when you click on the name of the created Issue, a new element is dynamically added (a pop-up window in which you can edit the Issue), in which the Number and Title data are inserted in the form of HTML code. If the output of this data in the form of text had been implemented, or at least filtering of this data and removal of HTML tags had been performed, these vulnerabilities would not have existed.

According to the testing, the following OJS flaws were identified:

- DOM XSS vulnerability;
- Stored XSS vulnerability;
- lack of validation of tags and text entered into the input fields, although the OJS settings specify the allowed tags.

Security measures that are available in OJS:

- password hashing (sha1), but you need to add a secret key;
- most input fields have validation of the entered data and are displayed on the page in the form of text, not in the form of code, which blocks possible attacks;
- ordinary authors do not have access to the admin panel;
- you can add an SSL certificate.

To fix the found vulnerabilities and prevent possible XSS attacks, you need to build a security model. A security model is a set of measures and methods that can be used to build reliable OJS system protection.

For the "Number" field, you need to enter the following checks/restrictions (Figure 9):

- enter a check for the presence of the characters "< " ">", which are not usually used to indicate a number;
- introduce the encoding of the characters ">" "<" to avoid the inclusion of script tags in the page code if it is impossible to completely remove these characters from the title;
- prohibit entering letters, as the "Number" field should contain only numbers and possibly separating characters such as hyphens and dashes;
- if you want to leave the permission to enter letters in this field, then you need to introduce a check for dangerous JavaScript constructs, such as "alert()", "script", onclick and others, which can cause malicious code to run when you click on the Issue number.

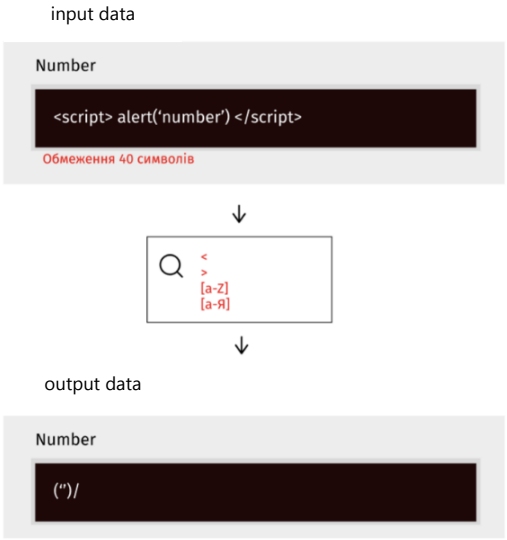

Fig. 9. General protection model of the “Number” field

For the "Title" field, you need to enter the following restrictions/checks (Figure 10):

- introduce a check for malicious JavaScript code constructs that may contain "script" tags, alert, innerHTML, and other JavaScript functions;
- introduce the encoding of the characters ">" "<" to avoid the introduction of script tags into the page code if it is impossible to completely remove these characters from the header.

Fig. 10. General protection model of the “Title” field

CONCLUSIONS

Protection against XSS attacks is extremely important for both businesses in general and OJS systems in particular. It ensures that the reputation and trust of users are preserved, as XSS vulnerabilities can lead to the leakage of confidential information and disruption of the system. In addition, XSS protection helps to ensure data security, which is a critical aspect in today's digital world, where data loss can have serious business consequences. Legal requirements and security standards also emphasize the importance of protecting against XSS attacks, as breaches of these requirements can lead to fines and legal issues for organizations.

For the OJS system, which publishes scientific materials, XSS protection is particularly important as it affects the trust of authors and readers in the platform, as well as the overall status and professionalism of the journal. Therefore, the development and implementation of effective XSS protection measures is a critical task to ensure the security and stability of the OJS system.

The XSS protection model for the OJS system focuses on preventing vulnerabilities associated with dynamic data output in the HTML code format. For the "Number" field, we suggest checking for the presence of the characters "<", ">", encoding the characters ">", "<", prohibiting the input of letters, and checking for unsafe JavaScript constructs. For the "Title" field, it is recommended to introduce a check for malicious JavaScript constructs and the encoding of the ">" and "<" characters. Such measures will help prevent DOM XSS and Stored XSS vulnerabilities while maintaining data security in the OJS system.

REFERENCES

[1] A survey of detection methods for XSS attacks. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S1084804518302042

[2] 35+ Cross-Site Scripting Statistics That Will Baffle You. [Online]. Available: https://securityescape.com/cross-site-scripting-statistics/

[3] B. Gogoi, T. Ahmed, and H. K. Saikia, "Detection of XSS Attacks in Web Applications: A Machine Learning Approach." [Online]. Available: https://www.ijircst.org/DOC/1-detection-of-xss-attacks-in-web-applications-a-machine-learning-approach.pdf

[4] What is XSS Hunter? [Online]. Available: https://www.hispa.eu/features

[5] Testing for Reflected Cross Site Scripting. [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/01-Testing_for_Reflected_Cross_Site_Scripting

[6] C. R. Pardomuan, A. Kurniawan, M. Y. Darus, M. A. M. Ariffin, and Y. Muliono, "Server-Side Cross-Site Scripting Detection Powered by HTML Semantic Parsing Inspired by XSS Auditor." [Online]. Available: http://www.pertanika.upm.edu.my/resources/files/Pertanika%20PAPERS/JST%20Vol.%2031%20(3)%20Apr.%202023/14%20JST-3458-2022.pdf

[7] HTTP Security Response Headers Cheat Sheet. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Headers_Cheat_Sheet.html

[8] The HttpOnly Flag – Protecting Cookies against XSS. [Online]. Available: https://www.acunetix.com/blog/web-security-zone/httponly-flag-protecting-cookies/

[9] DOM Based Cross Site Scripting or XSS of the Third Kind. [Online]. Available: http://www.webappsec.org/projects/articles/071105.shtml

[10] S. Buchyk, D. Shutenko, and S. Toliupa, "Phishing Attacks Detection," in *IX International Scientific Conference “Information Technology and Implementation" (IT&I-2022),* Workshop Proceedings, Kyiv, Ukraine, Nov. 30 - Dec. 02, 2022, pp. 193–201.

[11] S. Toliupa, S. Buchyk, A. Shabanova, and O. Buchyk, "The Method for Determining the Degree of Suspiciousness of a Phishing URL," in *X International Scientific Conference "Information Technology and Implementation" (IT&I-2023),* Workshop Proceedings (IT&I-WS 2023), Kyiv, Ukraine, Nov. 20-21, 2023, pp. 239-247.