American Research Journal of Computer Science and Information Technology

Volume 7, Issue 1, 37-41 Pages
Research Article | Open Access
ISSN (Online)- 2572-2921
DOI : 10.21694/2572-2921.24009

AMERICAN RESEARCH JOURNALS
An Academic Publishing House

Methods of Protection Against XSS Attacks at the Client Side Level

Okhonko Pylyp
Application Security Engineer, Tential, Rockville, Maryland, United States.

ABSTRACT

Protection methods against XSS attacks on the client side are key approaches to ensuring the security of web applications that prevent the introduction and execution of malicious code in users' browsers. The main methods include data validation and filtering, using the Content Security Policy (CSP) protocol, encoding user input, as well as using mechanisms to detect and prevent the execution of suspicious scripts. These measures are aimed at reducing the risks of both reflected and persistent XSS attacks, providing a higher level of protection for confidential information. Special attention is paid to an integrated approach that includes the interaction of server and client protection methods, which minimizes vulnerabilities and increases the resistance of web applications to various types of attacks.

KEYWORDS: *XSS attacks, data validation, data filtering, Content Security Policy, data encoding, web application protection, browser security, attack prevention.*

INTRODUCTION

With the development of web technologies and the widespread use of web applications, information security issues are becoming increasingly relevant. One of the most common threats faced by modern web applications is cross-site scripting (XSS attacks). XSS attacks are a type of vulnerability that allows attackers to inject malicious scripts into web pages, which are then executed on the client side. These attacks can lead to serious consequences, such as the theft of confidential data, user account compromise, and the execution of arbitrary code in the browser.

The relevance of this topic is driven by the growing complexity and diversity of modern web applications, as well as the increasing number of cyberattacks aimed at exploiting XSS vulnerabilities. Despite the advancement of protection methods, XSS attacks continue to pose a serious security threat, necessitating the development and implementation of reliable client-side protection methods. This is especially important as attacks become more sophisticated, and the consequences of their successful execution can be extremely destructive.

The aim of this paper is to analyze existing methods of protection against XSS attacks at the client-side level and identify the most effective approaches to ensuring the security of web applications.

XSS ATTACKS: DEFINITION AND TYPES

Cross-site scripting (XSS) is an internet security threat that occurs when malicious JavaScript code is injected into input fields and subsequently executed on the side of other users. This can lead to the theft of session data or redirection to phishing websites. One of the key protection mechanisms is the principle of domain interaction restriction. Web scripts operating within a single site can function without limitations, but their impact is confined to that domain and does not extend to other resources [1]. Reflected XSS attacks occur when the data entered by a user is immediately returned to them on the same page. An example would be a search bar that displays results based on the user's query. An attacker could create a malicious search query containing JavaScript code, which would execute when the search results are displayed, leading to the execution of harmful code in the user's browser [2].

There are three main types of XSS attacks: stored, reflected, and DOM-based attacks. Although there are more types, these three are the most common and deserve detailed consideration.

Stored XSS occurs when a malicious script is saved in the application's database and then executed when the page is generated for the client. An example might be an online store where users can leave reviews. If developers have not implemented adequate data filtering, an attacker could embed a script in a review, and anyone who opens the page with that review would be subjected to the attack.

Reflected XSS does not require the script to be stored on the application server. Here, the attack occurs via a URL

containing malicious code, which is sent to the server and returned in the response. The script then executes on the client side. This type of attack requires the attacker to individually target each user by sending them a specially crafted link.

DOM-based XSS is characterized by the script being injected into the DOM structure of the page on the client side when JavaScript processes data passed in the URL. Unlike other types of XSS, in this case, the HTML returned from the server is safe, but the script is activated during the data processing phase in the browser.

In addition to the above, there are other, less common types of XSS attacks. For instance, mutating XSS (mXSS) exploits the way browsers clean and process user data. Blind XSS takes advantage of vulnerabilities in an application where the script is triggered later and may affect another application or system. Self-XSS is an attack where the user enters malicious code into the web application interface themselves, falling prey to social engineering tactics. These types of attacks highlight the importance of continually improving security measures and awareness of various vulnerabilities, even if they are less common [3].

Reflected XSS is also one of the types of XSS attacks, where the malicious script is not stored on the application's servers but is delivered to the victim through a link. This process is implemented via a URL containing malicious code, which the server displays on the web page generated for the user. As a result, the user sees a page with a script already embedded in it, which executes on the client side. Unlike stored XSS vulnerabilities, reflected attacks only affect users who follow specially crafted links. This vulnerability is more prone to detection at the Web Application Firewall level since the malicious code is transmitted in the URL, which is logged everywhere. Thus, a victim can be targeted without even realizing it.

DATA VALIDATION AND FILTERING ON THE CLIENT SIDE

To ensure the security of frontend applications, comprehensive measures must be taken to minimize risks and vulnerabilities. However, it is important to understand that security is not a final goal but an ongoing process that requires regular monitoring and adaptation depending on the project's context.

Data recording should be done using encoding immediately before storing information obtained from the user on the web page. This is necessary because the choice of the appropriate encoding type depends on the context in which the recording occurs. For example, embedding values in JavaScript requires one type of escaping, while HTML requires a different approach.

Data validation is the process of checking the correctness and security of information entered by the user. If a web page lacks a validation mechanism, users can enter incorrect or incomplete data, which may complicate further processing and create a security threat.

Client-side validation occurs before the form is submitted to the server. It allows the user to promptly correct errors that may arise when filling out fields. For example, input data can be checked against specific criteria, such as the format of an email address or phone number. HTML attributes can be used to set rules for each input field. For instance, the type attribute specifies the type of data being entered, while required makes the field mandatory. The pattern attribute allows for the use of regular expressions to validate data, such as a phone number.

Despite the usefulness of client-side validation, relying on it alone is not advisable, as JavaScript code may fail to load due to internet connection issues or other factors. In such cases, HTML attributes play an important role in preventing incorrect data entry.

Server-side validation occurs after data is submitted. This method is more reliable because it does not depend on the client device and can detect errors that may be missed on the client side. Server-side validation is typically implemented using programming languages like PHP, Python, or Java. The main stages include receiving the data, checking it against established rules, and then processing the valid data, including storing it in a database or using it in further operations.

It is important to note that server-side validation also plays a key role in ensuring security by preventing XSS attacks through escaping user input [4].

When working with HTML content, any values not included in the approved list must be converted into HTML entities:

- The "<" symbol should be replaced with: `<`

- The ">" symbol should be replaced with: `>`

For data placed in a JavaScript string, non-alphanumeric characters should be converted to Unicode format:

- The "<" symbol should be replaced with: `\u003c`

- The ">" symbol should be replaced with: `\u003e`

Sometimes it may be necessary to use multiple levels of encoding, and this must be done in a specific sequence. For example, if you need to safely embed user input into an event handler, both JavaScript and HTML contexts must be considered. In this case, the input should first be converted to Unicode format, and then to HTML entities.

Data encoding is a critically important tool for protecting against XSS attacks, but it does not provide full security in every context. In addition to this, strict input validation should be carried out when data is received from the user.

Examples of input validation include:

- Ensuring that a URL provided by the user begins with a secure protocol, such as HTTP or HTTPS. This prevents the use of unsafe protocols like `javascript:` or `data:`.

- Checking that numerical values match the expected data type, for instance, ensuring that the provided value is an integer.

- Restricting input to allowed characters only.

The ideal approach is to block invalid input, while attempts to clean incorrect input to make it valid often lead to errors and should only be used as a last resort.

Modern web applications often use server-side templating engines, such as Twig, Blade, or Freemarker, to embed dynamic content into HTML. These engines typically have built-in methods for escaping data. For example, Twig uses the `e()` filter with an argument that defines the context:

```
{{ user.firstname | e('html') }}
```

Some templating engines, such as Jinja or React, by default prevent the execution of dynamic content, significantly reducing the risk of XSS attacks.

Before using a templating engine, it is important to carefully review its data-escaping capabilities. It is crucial to remember that if user input is directly inserted into template strings, it can lead to server-side template injection, which is often more dangerous than XSS.

PHP provides a built-in `htmlentities` function designed for encoding entities. This function should be used to escape user input in the context of HTML. It requires three arguments: the input string, the `ENT_QUOTES` flag, which indicates the need to encode all quotes, and the character set, usually UTF-8.

Example usage:

```
<?php echo htmlentities($input, ENT_QUOTES, 'UTF-8');?>
```

For the context of JavaScript strings in PHP, Unicode format escaping must be used. Although PHP does not have a built-in API for this, a custom function can be implemented:

```
<?php
function jsEscape($str) {
  $output = '';
  $str = str_split($str);
  for($i=0; $i<count($str); $i++) {
    $chrNum = ord($str[$i]);
    $chr = $str[$i];
    switch($chr) {
      case "'":
      case '"':
      case "\n";
      case "\r";
      case "&";
      case "\\";
      case "<":
      case ">":
        $output .= sprintf("\\u%04x", $chrNum);
        break;
      default:
        $output .= $str[$i];
        break;
    }
  }
  return $output;
}
?>
```

This function can be used to escape a string before it is used in JavaScript:

```
<script>x = '<?php echo jsEscape($_GET['x'])?>';</script>
```

An alternative is to use a server-side templating engine. For safe data escaping in the context of HTML within JavaScript, a custom HTML encoder is required, as JavaScript does not provide built-in tools for this. Here is an example of a function that converts a string into HTML entities:

```
function htmlEncode(str) {
  return String(str).replace(/[^\w. ]/gi, function(c) {
    return '&#' + c.charCodeAt(0) + ';';
  });
}
```

This function can be applied to escape data before rendering it. If user input is used inside a JavaScript string, Unicode escaping must be performed:

```
<script>document.body.innerHTML =
htmlEncode(untrustedValue)</script>
function jsEscape(str) {
  return String(str).replace(/[^\w. ]/gi, function(c) {
    return '\\u' + ('0000' + c.charCodeAt(0).toString(16)).
slice(-4);
  });
}
```

An example usage of this function [5]:

```
<script>document.write('<script>x="'+jsEscape(untrusted
Value)+'";<\/script>')</script>
```

To ensure the security of data input in an application, careful validation of input data must be implemented. The primary goal of such validation is to prevent injection attacks, where attackers may insert malicious code by exploiting vulnerabilities in input fields.

Maintaining up-to-date project dependencies is an important aspect of ensuring security. Outdated libraries may contain vulnerabilities that attackers can exploit to inject malicious code or carry out other attacks, such as denial of service. Regular updates and vulnerability checks for packages help reduce risks and protect the application from potential threats [6].

Thus, creating a secure frontend application requires a comprehensive approach that includes the use of proven encryption methods, thorough data validation, proper security policy configuration, and continuous updating of dependencies. Only such an approach can provide reliable protection against various types of attacks and minimize potential risks.

USE OF CONTENT SECURITY POLICY (CSP)

The Content Security Policy (CSP) concept is a standard developed by the W3C consortium aimed at reducing the risks of injection attacks, such as XSS. CSP gives developers the ability to control which resources can be loaded and

executed on a web page, thereby minimizing potential threats. This technology allows the establishment of rules governing the loading of resources (such as images, JavaScript, fonts, etc.) in the user's browser. These rules can permit or restrict the loading of specific resources, set conditions for resources from certain domains, and block loading from other sources.

For example, it is possible to configure a policy so that browsers only load images from the domain example.com, ignoring all other sources. This is particularly important when ensuring the security of web applications by preventing the loading of unverified or potentially malicious data.

CSP can be implemented via HTTP headers or using META tags. HTTP headers include directives that define where different types of resources can be loaded from. Examples of such directives include `default-src`, which sets general loading rules, and `script-src`, which restricts JavaScript loading sources. Directives can be combined to create more complex policies. For instance, the command `Content-Security-Policy: default-src 'self'; img-src example.com;` will allow any resources to be loaded only from the current domain, while images can only be loaded from example.com.

CSP also supports a reporting mechanism that logs violations of the established policies. This is achieved using the `report-uri` directive, which sends violation data to a specified server. If violations are detected, the information is transmitted in JSON format.

Before implementing CSP, it is important to carefully analyze all the resources required for the proper functioning of your web application. Incorrectly configured policies can block critical components of the site, leading to system errors. If there is uncertainty about the necessary set of resources, CSP can be activated in report-only mode. In this mode, violations will be logged, but resources will not be blocked, allowing you to refine the policy before fully enforcing it [7].

One approach to implementing CSP involves using the `Content-Security-Policy` header, which allows strict rules to be set for acceptable content on web pages. In modern applications, developers can define the security policy header at the application code level, allowing greater control and flexibility in ensuring security. Developers can directly implement CSP in the application to protect against attacks such as cross-site scripting (XSS). It is essential to test and monitor the effectiveness of these policies to avoid potential disruptions to site functionality [8].

DETECTION AND PREVENTION OF SUSPICIOUS SCRIPT EXECUTION

To enhance the protection of web applications, it is recommended to use Content Security Policy (CSP), which restricts the browser from loading resources from untrusted sources. This significantly reduces the risk of successful XSS attacks, even if there are vulnerabilities in the site's code. The `report-uri` directive is a mechanism for transmitting data on security policy violations. When deviations from the established security rules are detected, the browser sends reports with information about the violation in JSON format to a specified server, the address of which is indicated in the content security policy. This can be any local or external URI, for example:

```
Content-Security-Policy-Report-Only: default-src 'self'; ...;
report-uri /your_csp_report_parser;
```

It is worth noting that it is possible to use both the `Content-Security-Policy-Report-Only` and `Content-Security-Policy` headers simultaneously to test new security rules while still applying the existing policy. After the new policy is activated, the `report-uri` directive can continue to be used to receive detailed reports on detected violations. This ensures effective monitoring and analysis of policy enforcement without fully restricting actions, which is especially useful during the testing phase. Each JSON report starts with the `csp-report` attribute and looks like this:

```
{
  "csp-report": {
    "document-uri": "http://netsparker.com/index.html",
    "referrer": "http://nasty.example.com/",
    "blocked-uri": "http://nasty.example.com/nasty.js",
   "violated-directive": "script-src 'self' https://apis.google.
com",
      "original-policy": "script-src 'self' https://apis.google.
com; report-uri http://netsparker.com/your_csp_report_
parser"
  }
}
```

As you can see, the reports contain detailed information about each case of policy violation, including the blocked URI and the corresponding directive that was violated. This greatly simplifies the process of diagnosing and resolving issues, especially when the security policy includes numerous directives and parameters. This approach allows for the quick identification and correction of shortcomings, improving the overall efficiency of working with security policies [9].

CONCLUSION

In conclusion, it should be emphasized that effective protection against XSS attacks on the client side requires the application of a range of methods that complement each other, creating a robust security system for web applications. Data validation and filtering, proper encoding of user input, configuring Content Security Policy (CSP), and using modern tools for detecting and preventing the execution of malicious code are all critically important for safeguarding users' confidential information. However, despite all efforts to secure the client side, server-side security measures must not be overlooked, as only a comprehensive approach can provide maximum protection against XSS attacks.

REFERENCES

1. Pronina D. A., Loginova I. M., Eshelioglu R. I. Cross-site scripting or XSS attack //Scientific research of young scientists. - 2022. – pp. 246-250.
2. Weamie S. J. Y. Cross-site scripting attacks and defensive techniques: A comprehensive survey //International Journal of Communications, Network and System Sciences. – 2022. – Vol. 15. – No. 8. – pp. 126-148.
3. Krylov I. D. Effective ways to detect and prevent XSS vulnerabilities of sites //StudNet. - 2021. – Vol. 4. – No. 2.
4. Filatova D. K., Treshchev I. A., Karpova N. G. Approaches to the analysis of XSS vulnerabilities on web resources based on Google's XSS Game //Science, innovation and technology: from ideas to implementation. – 2022. – pp. 156-159.
5. Konkin A. A., Treshchev I. A. Effective ways to prevent XSS vulnerabilities in web applications //science, innovations and technologies: from ideas to implementation. – 2022. – pp. 115-117.
6. Petrovskaya A. S. Methods of recognizing and preventing phishing attacks //Electronic collection of works by young specialists of Polotsk State University named after Euphrosyne of Polotsk. Legal sciences. – 2021. – No. 37. – pp. 227-230.
7. Voloshko M. Yu., Kulikova N. N. Cross-site scripting (XSS). methods and methods of protecting Web systems // problems of effective use of the scientific potential of society. - 2021. – pp. 99-101.
8. Applying the Content Security Policy for ASP.NET Core Blazor. [Electronic resource] Access mode: https://learn.microsoft.com/ru-ru/aspnet/core/blazor/security/content-security-policy ?view=aspnetcore-8.0 (accessed 08/31/2024).
9. Modern feedback from web systems: we configure the report-uri. [Electronic resource] Access mode: https://www.atraining.ru/report-uri-settings / (accessed 12.09.2024).

Citation: Okhonko Pylyp, "Methods of Protection Against XSS Attacks at the Client Side Level", American Research Journal of Computer Science and Information Technology, Vol 7, no. 1, 2024, pp. 37-41.