|  |  |
| --- | --- |
| **Lab 15** | |
| **Name** | Dang Hoang Nguyen |
| **Student ID** | SE171946 |

What defines Stored Cross-Site Scripting (XSS), and how does it differ from Reflected XSS and DOM-Based XSS?

A online security flaw known as stored cross-site scripting (XSS) occurs when malicious code is persistently saved on a web server and then mirrored back to users who view the compromised material. This is different in numerous important aspects from both DOM-Based XSS and Reflected XSS:

Contained XSS:

* Persistence: Every user that visits the compromised page is impacted by the malicious code that is present on the server.
* Injection Points: Forums, comment sections, product evaluations, user profiles, and any other user-editable content kept on the server are examples of vulnerable places.
* Attack Vectors: By taking advantage of inadequate input validation and sanitization, attackers usually insert code using weak forms.
* Impact: Comparable to Reflected XSS, but potentially more persistent and affecting a larger audience. On all impacted pages, hackers have the ability to manipulate users, distribute malware, steal data, and take over sessions.

Distinctions from the Reflected XSS:

* Persistence: While Reflected XSS targets certain users based on predetermined activities, Stored XSS is permanent and affects all users.
* Attack Scope: Reflected XSS targets particular users who click on malicious links or enter malicious information, whereas Stored XSS affects all users who browse the compromised website.

Distinctions from DOM-Oriented XSS:

* Storage: While DOM-Based XSS modifies already-existing content on the client-side without requiring server storage, Stored XSS depends on the server-side storage of malicious code.
* Code Origin: While DOM-Based XSS attacks take use of flaws in the page's code, stored XSS attacks insert code from outside sources.

Explain the process of how malicious scripts are injected, stored, and subsequently executed in Stored XSS attacks, and discuss the potential consequences these attacks can have on both users and web applications.

Stored XSS: From Injection to Execution and Beyond

Stored Cross-Site Scripting (XSS) poses a serious threat as it allows attackers to plant malicious code on vulnerable websites, impacting all users who interact with the infected content. Let's delve into the process and consequences:

1. Injection:

The attacker crafts malicious JavaScript code disguised as user input. This could be:

* A forum comment containing a hidden <script> tag.
* A product review embedding a malicious URL.
* A profile bio injecting code through seemingly harmless text.

The vulnerable application lacks proper validation and sanitization, allowing the attacker to submit and store the malicious code on the server. This can happen in user-editable sections like forums, comments, profiles, or any content allowing user input.

2. Persistence

Unlike Reflected XSS, the injected code doesn't vanish after a single visit. It gets permanently stored on the server, becoming part of the infected page.

This persistence differentiates Stored XSS, making it a wider threat affecting all users who access the contaminated content.

3. Execution:

Whenever a user visits the infected page, the malicious code automatically executes within their browser. This happens because browsers trust content originating from the server.

The script now has unauthorized access to the user's browser environment, enabling the attacker to:

* Steal data: Cookies, session IDs, login credentials, or other sensitive information.
* Hijack sessions: Take control of the user's session for further exploits.
* Spread malware: Redirect users to malicious sites for malware installation.
* Deceive and manipulate: Display fake content or messages to trick users into revealing information or taking specific actions.

**Lab: Stored DOM XSS**

We can infer a few things from the lab description and the extremely brief title. We plan to resume work on our blog page. This time, we're going to leave our payload in a comment rather than the search bar.

Enter the lab. You will be directed to a blog page featuring multiple haphazard blogs. Select your preferred option and click to display it. To reach the comment form, scroll down.

We want to ascertain how our comment will be handled first. Complete all the fields. I include an alpha-numeric string in the comment section that should be simple to look for later. Select "Post Comment."

A computer screen shot of a computer

Description automatically generated

Click ‘Back to Blog’ to view our comment.

To find your remark, scroll down from this point.

From the drop-down menu, choose "Inspect" after performing a right-click on your comment. Your comment will be highlighted when this opens in your DOM-browser (you might need to expand the code to see it completely).

A screenshot of a computer

Description automatically generated

The two <script> elements above our comment are visible. They are cleaning up our comments before re-posting them on the page.

Let's investigate their operation.

Select the "Network" tab in your DOM, then reload the website. As we watch, you'll notice that the page is connecting to multiple destinations, like "src="/resources/js/loadCommentsWithVulnerableEscapeHtml.js," to create a completely functional webpage for us to view.

A computer screen shot of a computer

Description automatically generated

Clicking on ‘loadCommentsWithVulnerableEscapeHtml.js’ will bring up another window with several tabs. We’re interested in seeing the response. Click on the ‘Response’ tab in the new window.

A screen shot of a computer program

Description automatically generated

This function's initial portion is well known. I went over it in the walkthrough for the Reflected DOM XSS. It appears that the developer is utilizing JSON.parse() in place of eval(), which we learned about previously due to its drawbacks. Please visit this link to view the complete breakdown of what is going on.

In summary, the JavaScript function loadComments(postCommentPath) uses XMLHttpRequest to send an asynchronous HTTP GET request to the server. It parses the JSON-formatted comments and invokes a function ('displayComments') to manage or show the comments back to the page after obtaining a successful response (HTTP status 200).

But take a look at the following feature. This is the point at which we are vulnerable.

In JavaScript, a string pattern will only be changed once when utilizing the.replace() method. Instead, the developer in this case need to have used the.replaceAll() method. You can get more details regarding the.replace() technique at the article's conclusion.

We can now begin building our next comment by looking at how our last one was handled on the back end.

As the replace() function will only replace the initial occurrence in the comment string, we can include our payload in a follow-up tag if we include a throw away '<>'. Our payload will resemble this:

<><img src=x onerror=alert(1)>

Return to a comment form and fill out all the information again. This time, though, add our payload to the comment section and click ‘Post Comment’

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated