|  |  |
| --- | --- |
| **Lab 14** | |
| **Name** | Dang Hoang Nguyen |
| **Student ID** | SE171946 |

What is Reflected Cross-Site Scripting (XSS), and how does it differ from other types of XSS attacks like Stored XSS and DOM-Based XSS?

A sort of computer security flaw known as "reflected cross-site scripting" (XSS) occurs when malicious code—typically JavaScript—is introduced into a website through user input. The victim's browser is subsequently forced to run the injected malware, which may compromise their session, steal their data, or send them to malicious websites.

Important traits of reflected XSS include:

* Instantaneous reflection: The user sees the attacker-injected code immediately reflected back to them in the form of their input.
* Non-persistent: The malicious code only appears in the mirrored response and is not saved on the server.
* focuses on specific users: The attack's main strategy is to fool a particular victim into clicking on a malicious link or entering dangerous data.

Effects of XSS Reflected:

* Data theft: Cookies, session IDs, and other private information are all susceptible to stolen by hackers.
* Attackers have the ability to seize control of a victim's session in order to carry out more exploits.
* Spreading malware: Taken to dangerous websites where malware is installed.
* Phishing and deception: presenting phony messages or content to trick users.

Explain how Reflected XSS works, including how malicious scripts are injected and executed in the victim's browser, and discuss the typical attack vectors for this type of vulnerability.

Web applications and user security are seriously threatened by Reflected Cross-Site Scripting (XSS). This is an explanation of how it functions:

1. Injection

* Malicious code, frequently Javascript, is created by the attacker and presented as user input. This might be included in: <script>alert('Gotcha!')</script> Search bar queries
* areas for comments
* user accounts
* URLs that have weak parameters

2. Introspection

The application adds the injected code to its response without performing the necessary validation and sanitization. This contemplation may take place in:

* The malicious script is displayed in search results.
* Rendering the injected code in the comments section
* Pages with user profiles that display the attacker's script
* Content produced with the weak URL parameter

3. Carrying out:

The malicious script is unintentionally parsed and executed by the victim's browser as part of the genuine website when they view the reflected page. Browsers trust material from the server from which it originated, which is why this occurs.

4. Effect

The script that is executed gives the attacker access to the victim's browser environment without authorization, which allows them to:

* Cookies, session IDs, and other private information can be stolen.
* Sessions of hijacking: Seize control of the victim's session to launch additional assaults.
* Distribute malicious software: send users to fraudulent websites so they can install malware.
* Deceive and manipulate: Conceive users with phony content or messages.

**Lab: Reflected DOM XSS**

Let’s see what needs to be done. Access the lab. You will be brought to our simple blog page, again, with a search bar.

A screenshot of a computer

Description automatically generated

In the search bar add your alpha-numeric search string that will yield 0 results. I’ll be using my usual nguyendh.

A screenshot of a computer

Description automatically generated

The entire search string we entered is returned to the website. We now need to investigate the location of our search string within the DOM and its processing method. This will help us understand the lab problem better.   
  
Choose Inspect from the drop-down menu by performing a right-click on the search string that was returned. Your search string will be highlighted when your DOM-browser opens as a result.

A screenshot of a computer

Description automatically generated

Can you see our search string reflected in the <h1> tag? Look at the <script> elements that are above our search query as well. Our search phrase is being processed by these two scripts, which then return the results in the <h1> tag below. However, how are they achieving that?

The JavaScript file located at'src=/resources/js/searchResults.js' is external. Imagine having your laundry done at a dry cleaner rather than at home. Our string is being processed "off-site" by the website, which then uses "search("search-results')" to return the results to the page.

However, we are still unable to watch the processing of our string. Click the 'Network' tab in your dev tools, as indicated below, to accomplish that.

Refresh the blog page in order to see all the different connections the page is making.

A screenshot of a computer

Description automatically generated

Do you recall the two <script> tags from before? The first one indicates where we should begin our search using '<script src="/resources/js/searchResults.js"></script>'.

On searchResults.js, click. A new window with multiple tabs will appear. Initially, you will see the "Headers" page; nevertheless, we must view its answer.

Select "Response."

A screenshot of a computer

Description automatically generated

Let’s take a look at the JavaScript code that processes our search string.

* function search(path): This declares a function with the name search and takes path as its only parameter. The second function calls this one. In this case, "path" is being handed in as "search-results."
* var xhr = new XMLHttpRequest(); — It makes a request to a given "path" together with the search parameters of the current URL using the "XMLHttpRequest" object.
* function() xhr.onreadystatechange {…} — Configures a callback function to be called each time the XMLHttpRequest's "readyState" property varies.
* If (this.status == 200 && this.readyState == 4){…} — Verifies that the response status is OK ('status == 200') and that the request has been finished ('readyState == 4').
* eval('var searchResultsObj ='+ this.responseText); — This code interprets and runs the response text as JavaScript code using the 'eval()' method. In essence, it establishes a variable called "searchResultsObj" and sets its value to the answer.
* The code xhr.open(“GET”, path + window.location.search); sets up the XMLHttpRequest to send a GET request to the path that is supplied, along with the search parameters (‘window.location.search’) of the current URL.
* The HTTP request is sent by using xhr.send();.

The object ‘xhr’ was created and an HTTP ‘GET’ request was sent to:

(“GET”, path + window.location.search) OR (‘search-results’ + ‘?search=nguyendh)

A screenshot of a computer

Description automatically generated

Clicking on ‘search-results?search=nguyendh and looking at the Response tab you’ll see the response in JSON. Our results are also enclosed in double quotes.

A screenshot of a computer

Description automatically generated

Here we can see both the Request and the response from ‘search-results?search=nguyendh in Burp.

From this point, you can right-click in the request box and select ‘Send to Repeater’. Next, click on the Repeater tab.

A screenshot of a computer

Description automatically generated

You can modify your search string on the Burp Repeater's left side in an attempt to "break-out" of the response. We will begin by appending a single double quotation to the end of both our search string and our alert() function because the results of our "searchTerm" are enclosed in double quotes.

A screenshot of a computer

Description automatically generated

The backslash that appears is an escape character that is intended to signal to the computer to "ignore the next character" rather than end the "searchTerm."

You can still see the double quote on the page.

Our search string has now ended with a double quote, and our '-alert(1)' has been exposed!

Still, the payload is invalid. The curly brace and last double quote that are meant to finish the JavaScript object are still present.

We must manually close the JavaScript object by adding a closing curly brace '}' to our payload in order to have a useable payload. Next, comment out everything that comes after "//."

A screenshot of a computer

Description automatically generated

Copy and paste the payload from the Burp Request into our blog page search bar and hit Enter.

A screenshot of a computer

Description automatically generated