# REPORT ON CROSS – SITE SCRIPTING (XSS)


**Anuja Rajesh Panchariya**

# Understanding Cross-Site Scripting (XSS)

## 1. Introduction to XSS

Cross-Site Scripting (XSS) is a type of security vulnerability commonly found in web applications. It allows attackers to inject malicious scripts into trusted websites. These scripts are executed in the context of the victim's browser, enabling attackers to steal sensitive information, hijack sessions, or deface websites.

## 2. Types of XSS

There are three main types of Cross-Site Scripting attacks:

### 2.1 Reflected XSS

In Reflected XSS, the malicious script is embedded in a URL or form input, and it is immediately returned to the user's browser without proper sanitization. This type of attack typically requires social engineering to lure victims into clicking a crafted link.

### 2.2 Stored XSS

Stored XSS occurs when malicious scripts are permanently stored on a target server, such as in a database. These scripts are later served to users when they access the affected page, making it a more potent and persistent form of attack.

### 2.3 DOM-Based XSS

In DOM-Based XSS, the vulnerability exists in the client-side scripts of the application. It occurs when the web page's Document Object Model (DOM) is manipulated to execute the malicious script, without any interaction with the server.

## 3. Examples and Demonstrations

### 3.1 Stored XSS Example

The following example demonstrates a Stored XSS attack. The attacker injects a script into a blog entry field. As seen in the screenshot, the malicious script displays a pop-up alert when the blog entry is loaded.

### 3.2 Reflected XSS Example

This example showcases a Reflected XSS attack. The attacker crafts a malicious URL containing a script that generates an alert box. When the user accesses the URL, the script is executed.

### 3.3 DOM-Based XSS Example

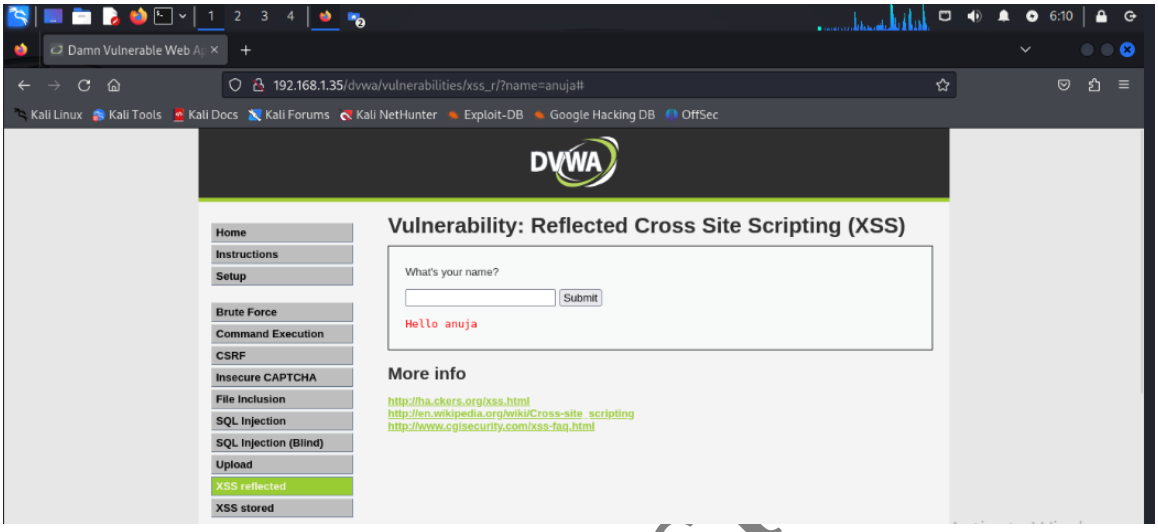
This screenshot demonstrates a DOM-Based XSS attack. The script is directly executed within the client-side DOM, causing the alert box to appear without any interaction with the server.

## 4. Mitigation Strategies

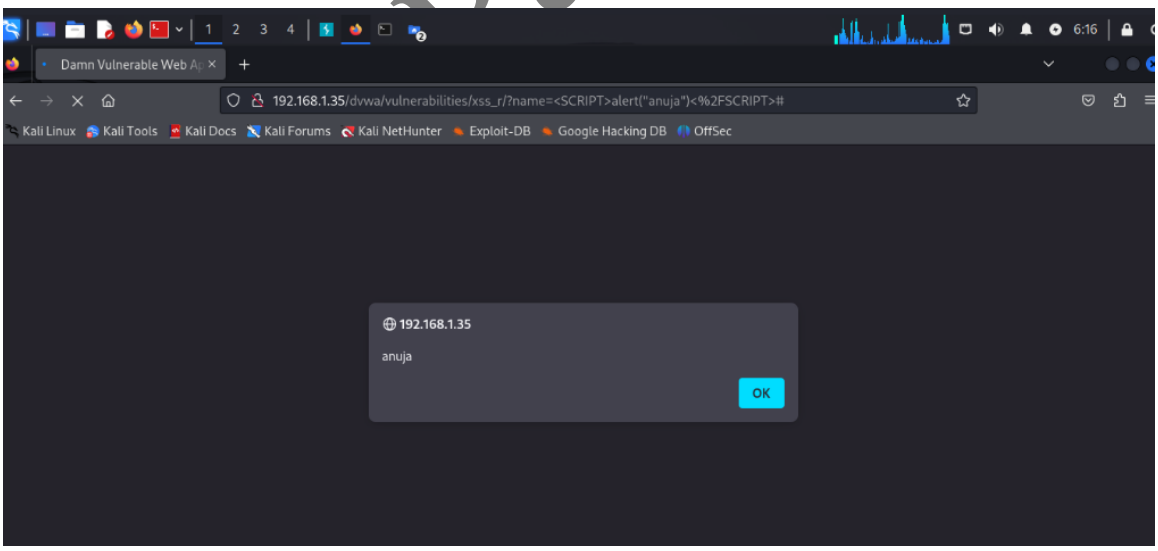
To prevent XSS attacks, developers should implement the following best practices:
1. Validate and sanitize user inputs.
2. Encode data before displaying it in the browser.
3. Use secure frameworks that provide built-in protection against XSS.
4. Implement Content Security Policies (CSP) to restrict the execution of scripts.
5. Regularly audit and update code to address vulnerabilities.

## 5. Output

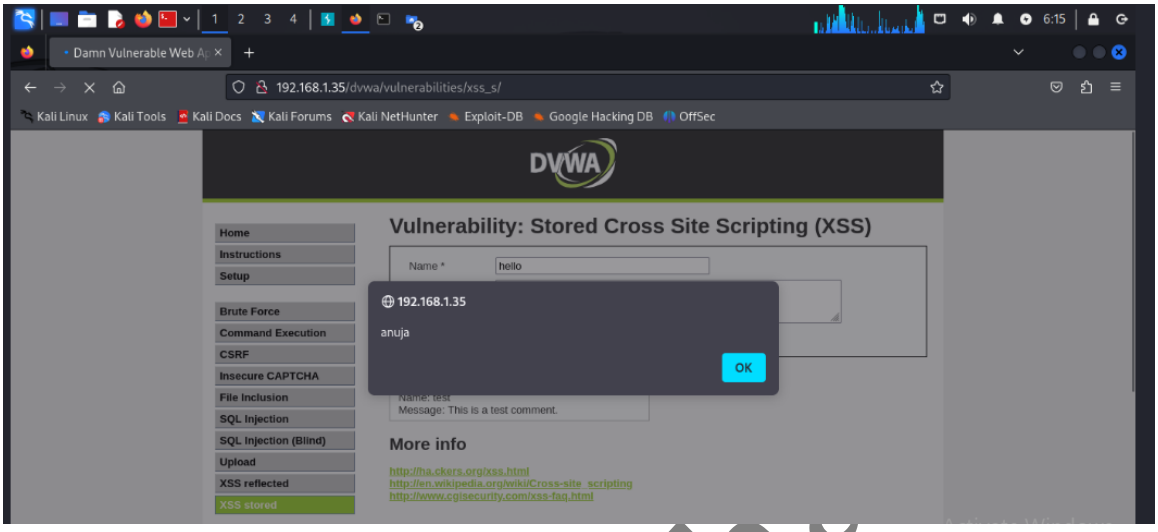
- Go to DVWA > XSS Reflected > Type your Name >It reflected your Name like ”Hello anuja”



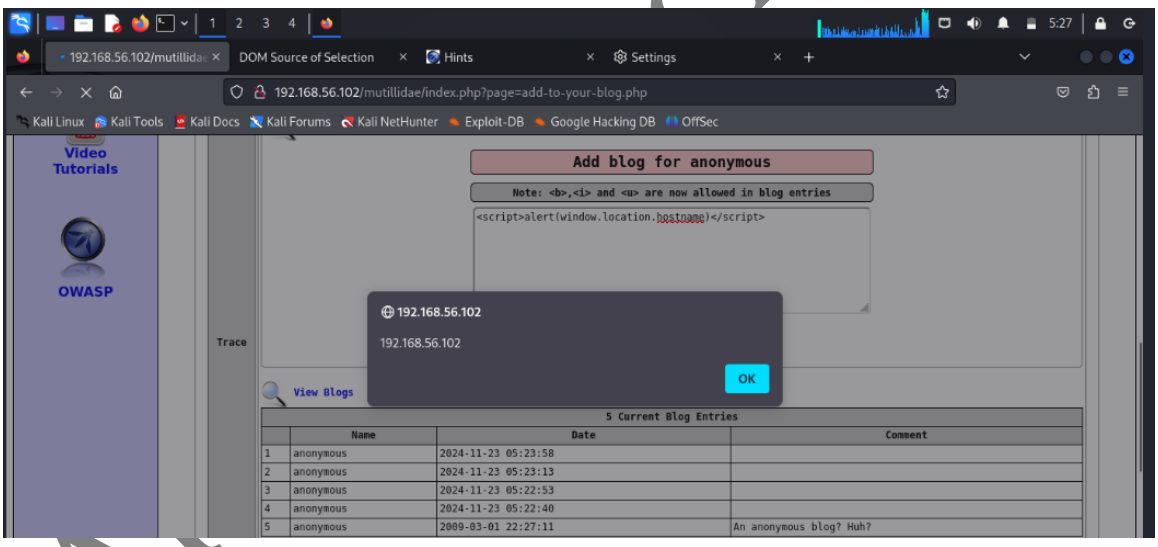

- If we write a script i.e. <script>alert (“anuja”)</script> It display alert like this



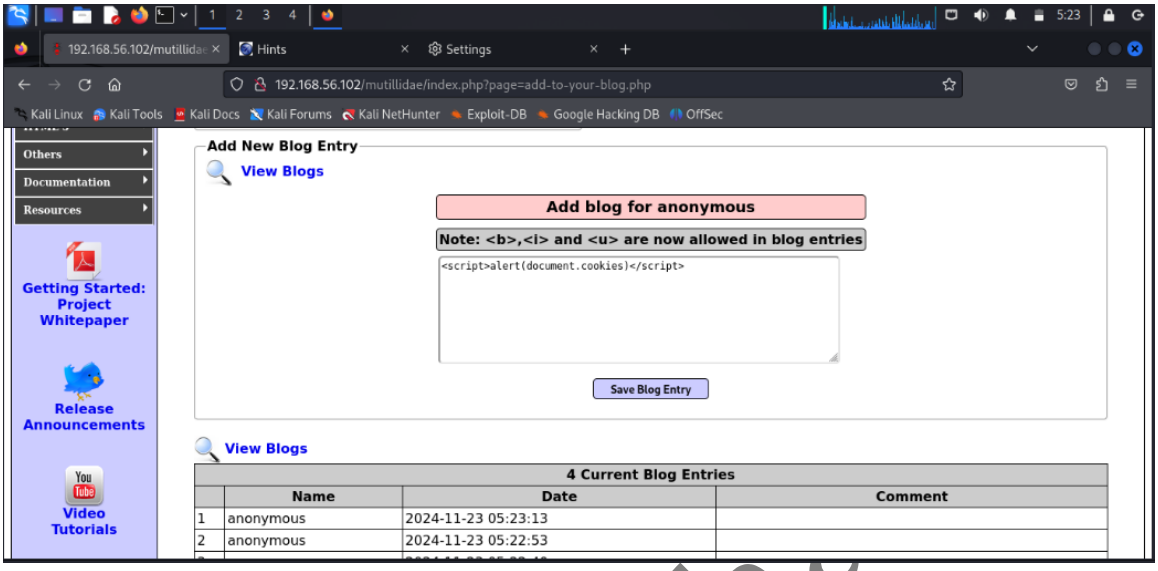
- Go to DVWA> Stored XSS > If we write a script i.e. <script>alert (“anuja”)</script> It display alert like this and it stores the alert



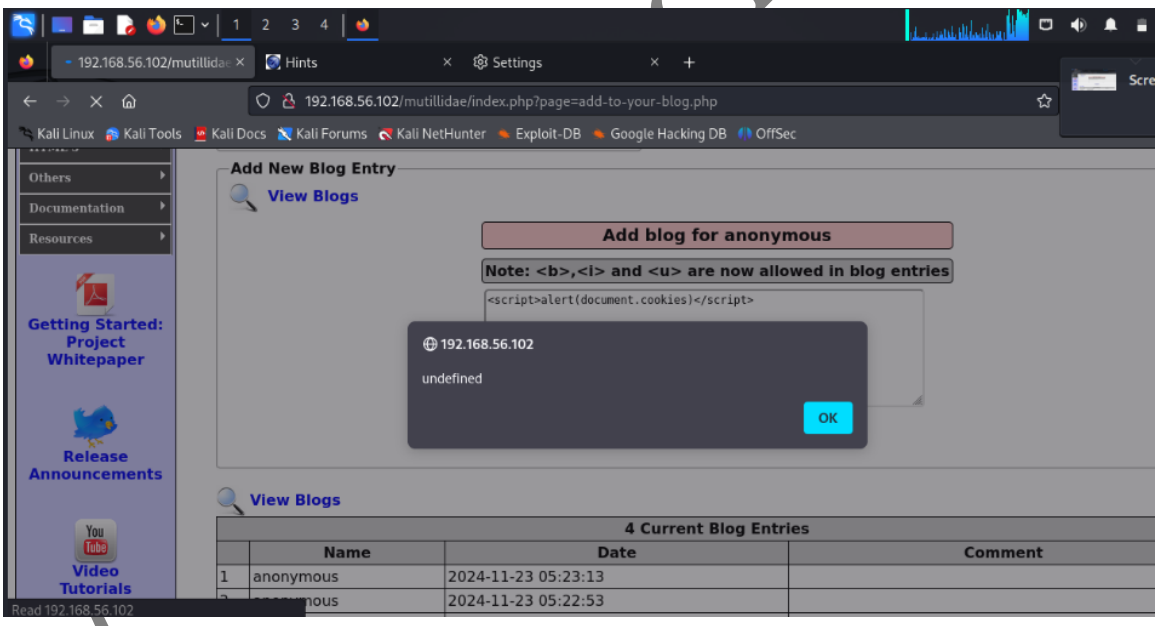

- To IP display : <script>alert (window.location.hostname)</script>



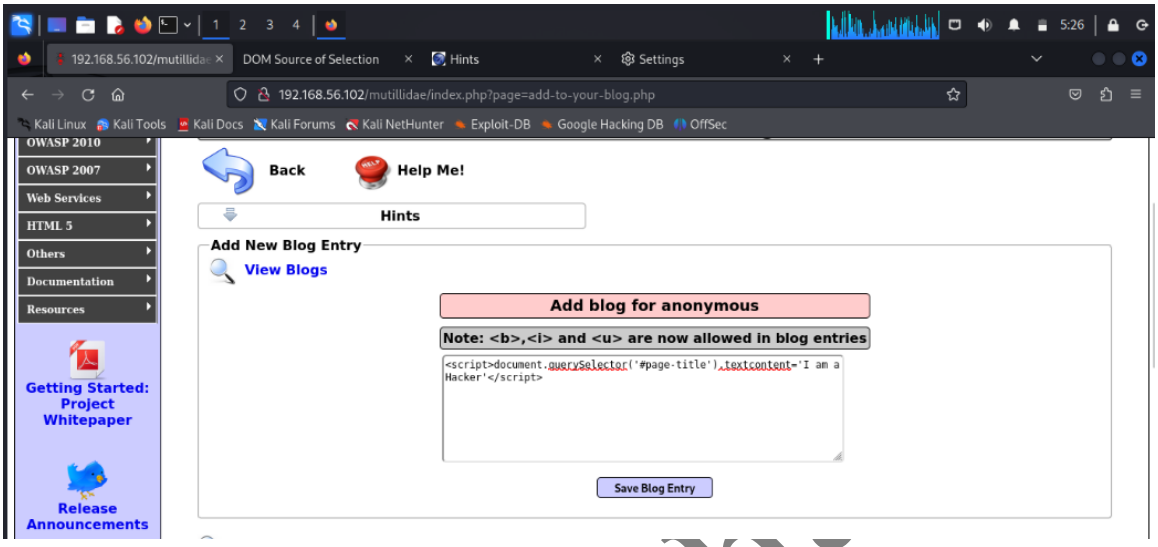
- To display Cookies : <script>alert(document.cookies)</script>



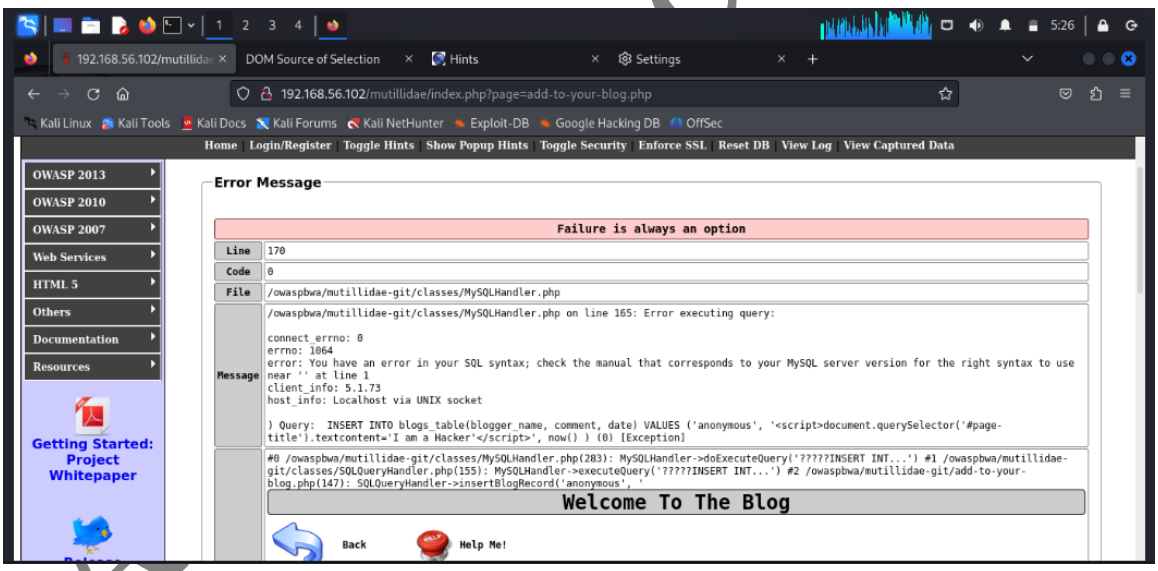

- Output : Undefined cookies



- To Change the Tittle : <script> document.querySelector('#thm-tittle').textcontent='I am a Hacker'</script>




- Output : Shows error because I did on OWASP website




## 6. Conclusion

Cross-Site Scripting (XSS) is a significant security threat that can compromise the integrity and confidentiality of web applications. Understanding the types of XSS and implementing robust mitigation strategies are essential for safeguarding users and systems

# Report on DOM-Based XSS Vulnerability

## 1. Executive Summary

This report documents the identification and exploitation of a DOM-Based Cross-Site Scripting (XSS) vulnerability in a web application. The vulnerability was found in the password generator feature, which improperly handles user inputs passed via the URL parameters. This allowed arbitrary JavaScript code execution in the client browser.

Key findings:

- Impact: The vulnerability can enable attackers to steal session cookies, manipulate the DOM, or launch phishing attacks
- Root Cause: Improper handling and reflection of user input in the DOM
- Mitigation: Implement input validation, sanitize output, and enforce secure coding practices.

## 2. Vulnerability Description

### What is DOM-Based XSS?

DOM-Based XSS is a type of Cross-Site Scripting vulnerability that occurs entirely on the client side. Instead of being processed on the server, malicious scripts are introduced into the Document Object Model (DOM) via vulnerable client-side scripts, resulting in arbitrary code execution.

### Observed Vulnerability

The application's password generator feature was vulnerable to DOM-Based XSS via the username parameter in the URL. Malicious JavaScript payloads injected into this parameter were executed in the victim's browser without validation.

### Steps to Reproduce

Open the vulnerable page:

1. http://192.168.56.102/mutillidae/index.php?page=password-generator.php&username=
2. 2. Inject a JavaScript payload into the username parameter:

```
<script>alert('Hello')</script>
```

3. The payload is executed in the browser, triggering the alert box with the text 'Hello'.

**Example Request (Captured via Burp Suite)**

GET /mutillidae/index.php?page=password-generator.php&username=<script>alert('Hello')</script>

## 3. Technical Details

**Root Cause Analysis**

Vulnerable Code:

```
var username = getParameterByName('username');
document.getElementById('welcomeMessage').innerHTML = username;
```

Here, the application directly inserts the username parameter into the DOM using innerHTML, enabling malicious scripts to execute.

Impact:

- Code Execution: Malicious JavaScript can run in the victim's browser.
- Data Theft: Attackers can steal cookies, session data, or other sensitive information.
- Defacement: The attacker can modify the page content to mislead users.

**Observed Behavior**

1. The payload <script>alert('Hello');</script> was successfully executed, displaying a pop-up with the text 'Hello'.
2. The server does not validate or sanitize the input before it is rendered in the DOM.

## 4. Exploit Scenario

**Potential Real-World Attack**

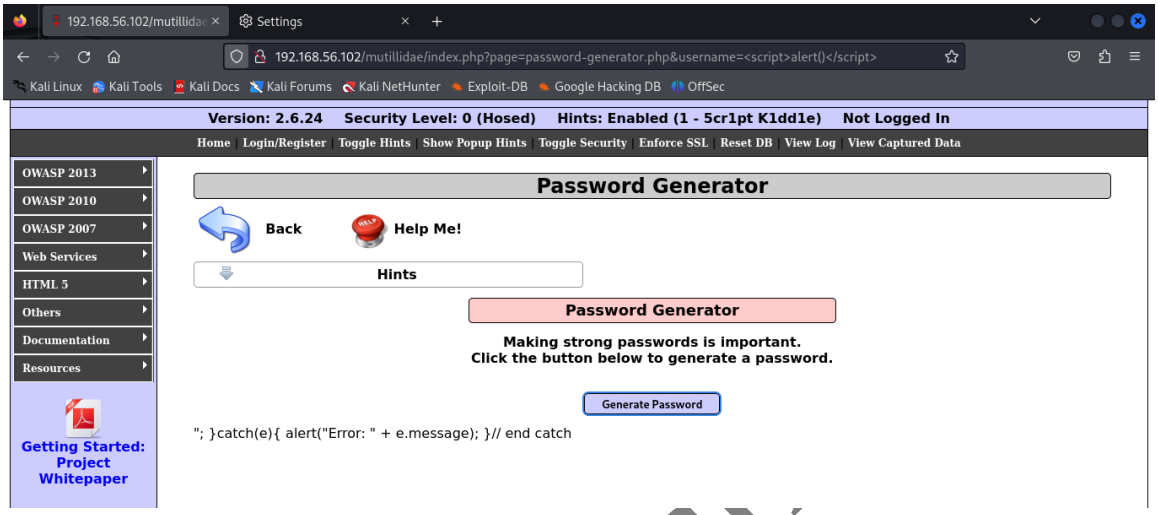
An attacker could craft a malicious link: http://192.168.56.102/mutillidae/index.php?page=password-generator.php&username=<script>document.cookie</script>

When clicked by a victim:

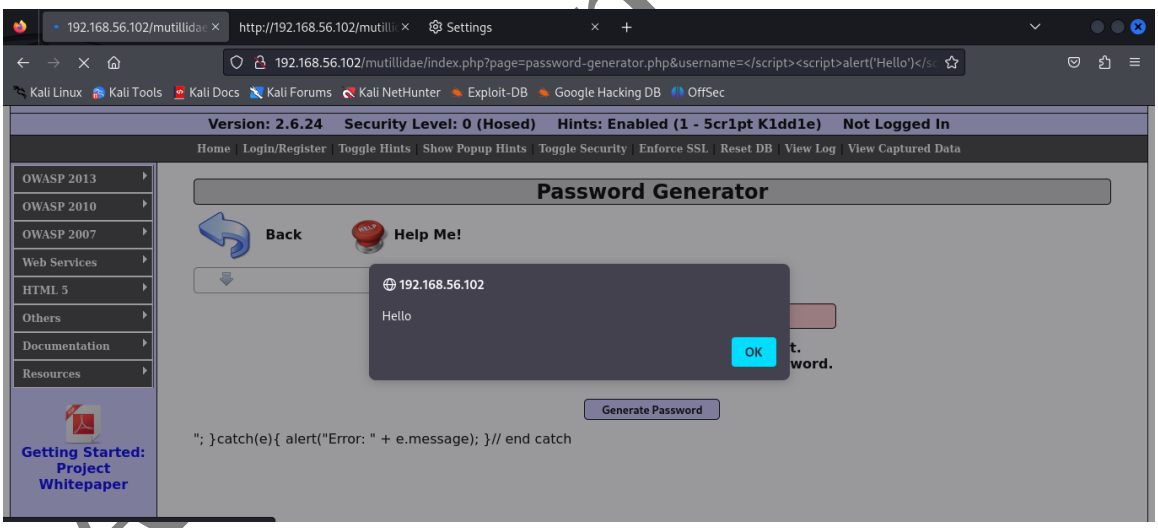
1. The malicious script extracts the victim's session cookies: document.cookie
2. These cookies could then be sent to the attacker's server, enabling session hijacking.

## Screenshot Evidence

1. Injection Point: The username parameter in the URL.




2. Execution Proof: Alert box displaying the injected message.




## 5. Mitigation and Recommendations

### Secure Coding Practices

1. Input Validation:

- Validate all user inputs, even those processed client-side. Reject unexpected characters, e.g., '<' and '>'.

2. Output Encoding:

- Sanitize all outputs before inserting into the DOM.
- Use textContent instead of innerHTML to prevent script execution:
  document.getElementById('welcomeMessage').textContent = username;

## HTTP Security Headers

- Content Security Policy (CSP): Prevent inline JavaScript execution using CSP.
- HTTPOnly and Secure Cookies: Set cookies with the HttpOnly and Secure flags to prevent client-side access.

#### Regular Security Audits

Use automated vulnerability scanners (e.g., OWASP ZAP, Burp Suite) to identify XSS vulnerabilities. Perform regular code reviews and penetration tests.

## 6. Conclusion

The identified DOM-Based XSS vulnerability in the password generator page poses a significant security risk. Exploitation can result in session hijacking, data theft, and reputational damage. Immediate steps, including input validation, output encoding, and security headers, should be implemented to secure the application.