**Cross-Site Scripting (XSS) Attack: Explained with Steps to Perform It**

### **What is a Cross-Site Scripting (XSS) Attack?**

Cross-site scripting, XSS, is a vulnerability whereby a malicious script is injected in the course of searching for harmful scripts, usually JavaScript. It occurs in cases where a web application includes user input in the output without adequate validation or encoding.

Once such scripts have been executed in the victim's browser, they might:

* steal the cookies or session tokens
* redirect users to malicious sites
* log keystrokes
* hijack accounts
* deface content

XSS is ranked in the OWASP Top 10 most critical web vulnerabilities.

## **Types of XSS Attacks**

1. **Stored XSS** – The malicious script is permanently stored on the server (e.g., in a blog comment).
2. **Reflected XSS** – The script is reflected off a web server, e.g., in search results or error messages.
3. **DOM-based XSS** – The vulnerability exists in the client-side JavaScript code and occurs when DOM elements are manipulated insecurely.

**Step-by-Step: How an Attacker Performs an XSS Attack:**

**Step 1:** Find an Input Field that Reflects Output

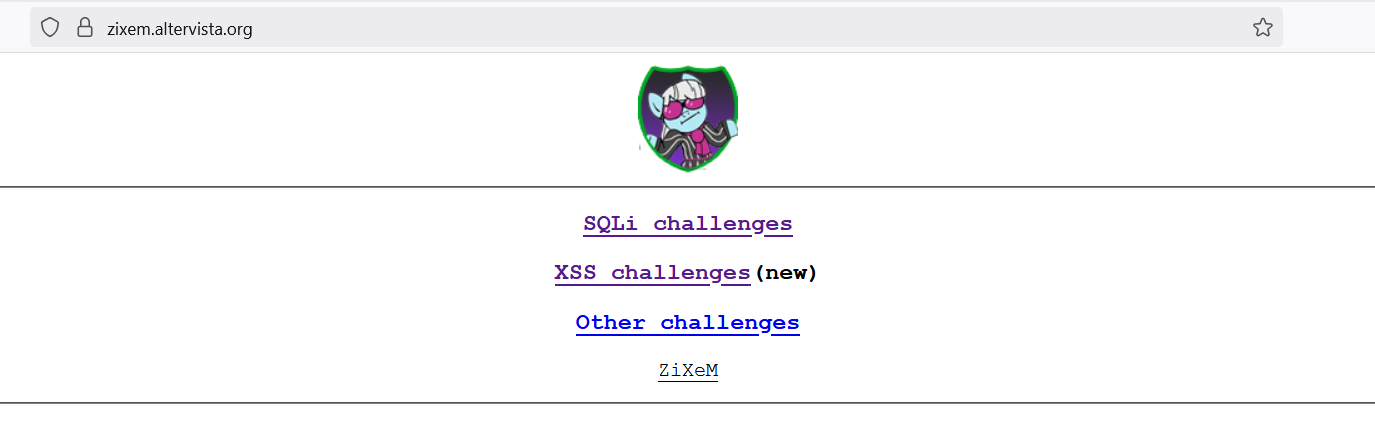
An XSS vulnerability can exist in any user-input field within a web application including URL parameters, search boxes, username or password fields, comment sections, feedback forms, or any other field that accepts and reflects user-supplied data without proper sanitization.

**Step 2:** Try Basic Script Injection

Enter this in the input or URL

<script>alert('XSS')</script>

Next, we’ll walk through an XSS challenge on the Zixem website, where we’ll demonstrate how this attack can be performed in a controlled environment for learning and testing purposes.



Welcome to another set of challenges from Zix!

These beginner-friendly XSS challenges are designed to help you understand the fundamentals of Cross-Site Scripting, including how filters work and how to navigate common attack scenarios. Your objective in each level is simple: successfully trigger a javascript alert — for example, alert(1337) — or any custom message. Once the alert box appears, you’ve successfully completed that level and can move on to the next. Good luck, and happy hacking!

Now, let's start with Level 1.

**Step 1:** Find the input field. Where you can write something.



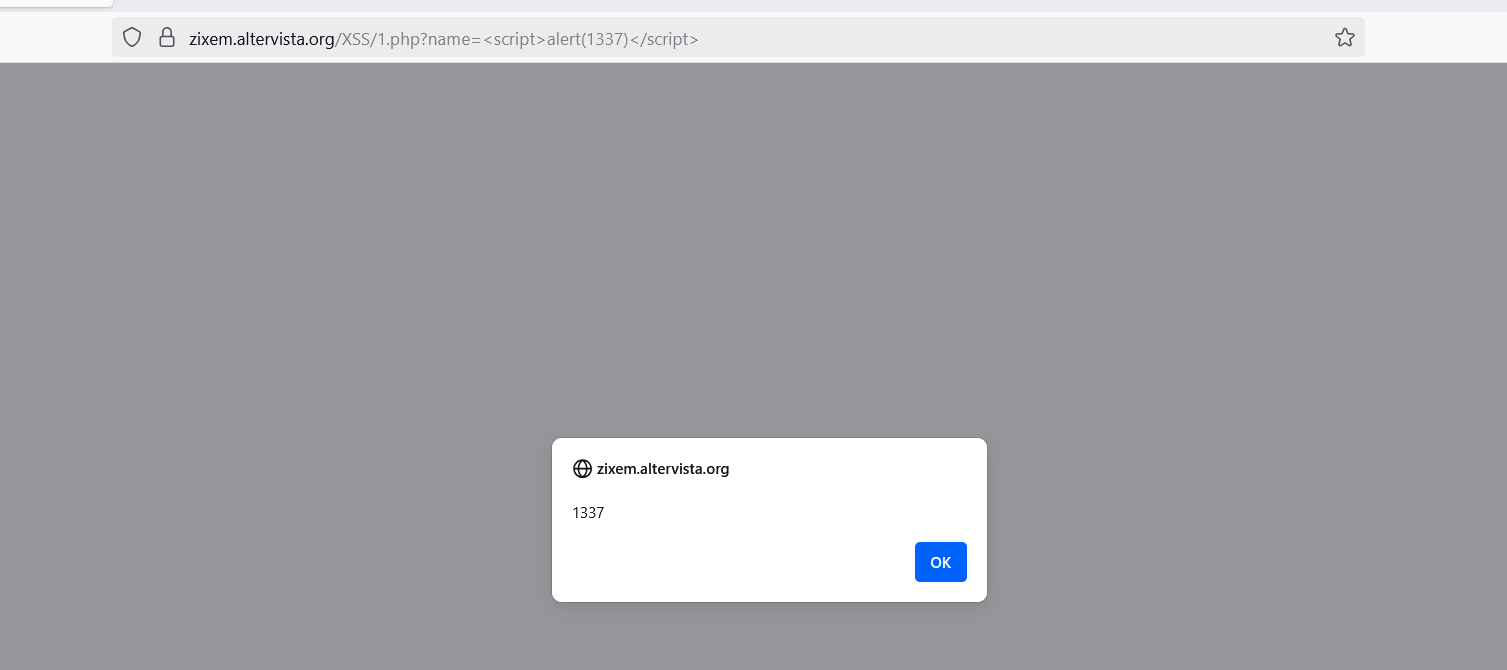
**Step 2:** Whatever input you give, it should get reflected.

Write hello, and it should get reflected.



**Step 3**: add the script here

$ <script>alert(1337)</script>



Let’s try **Level 2** now,

**Step 1:** find the field.



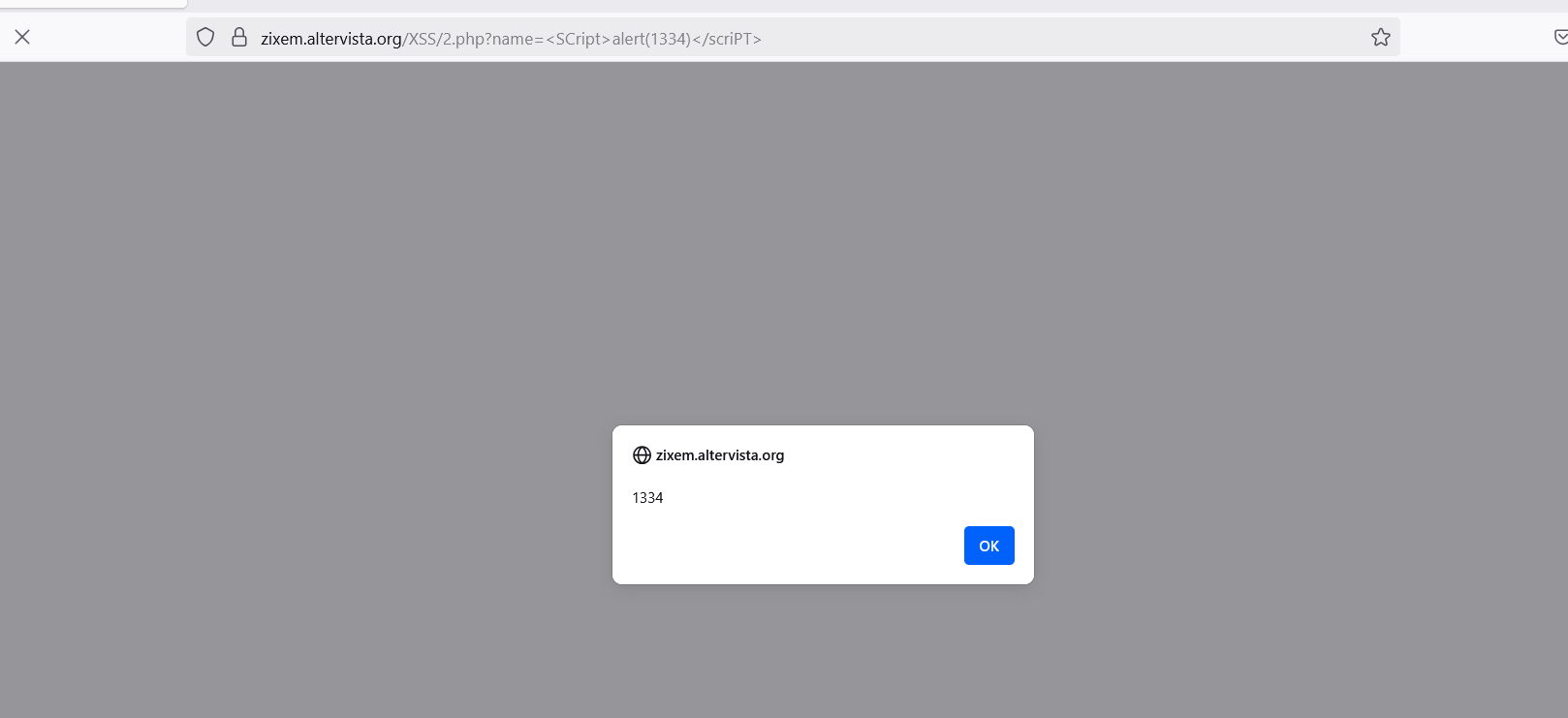
**Step 2:** write something in that field.

$ <i>cybersecurity<i>



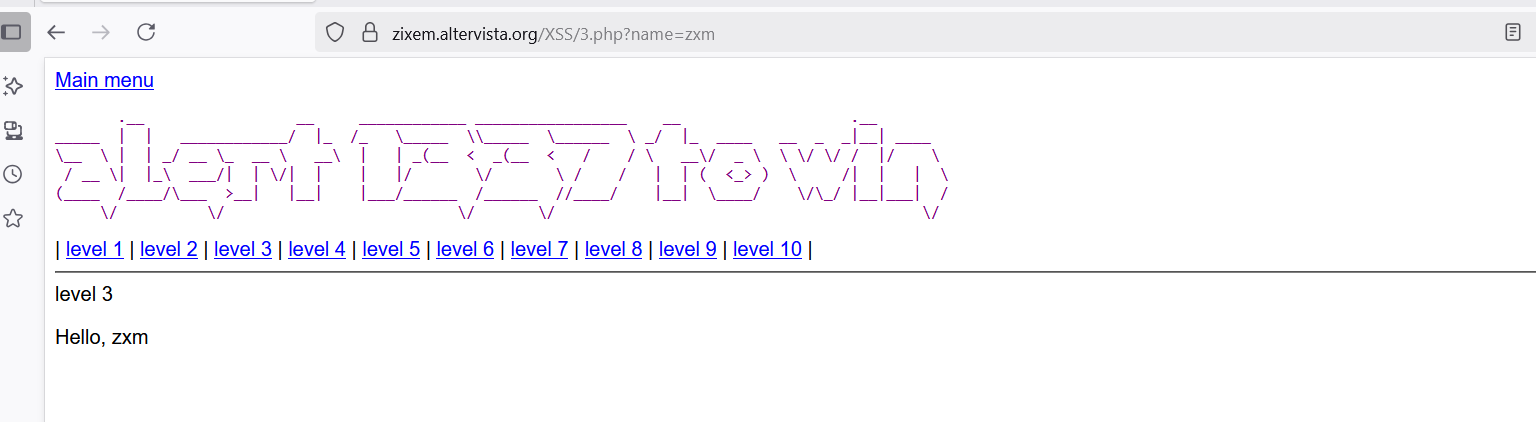
**Step 3:** add the script

$ <SCript>alert(1334)</scriPT>



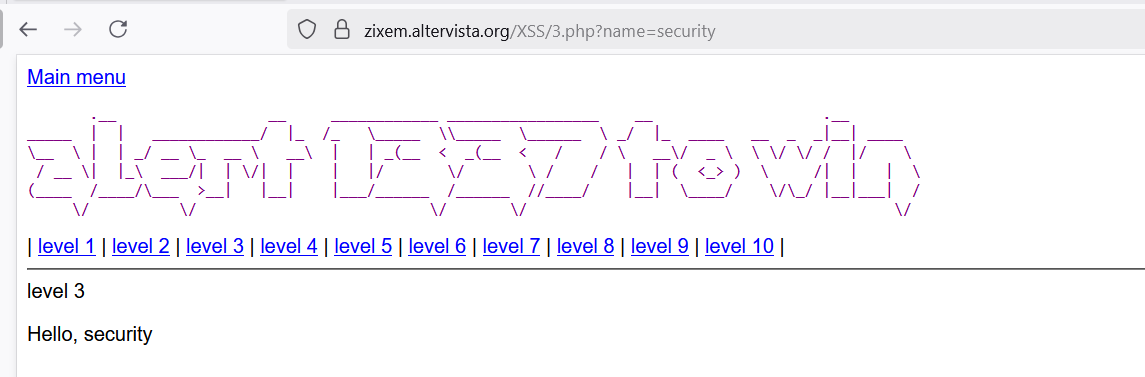
**Level 3:**

**Step 1:** Find the field



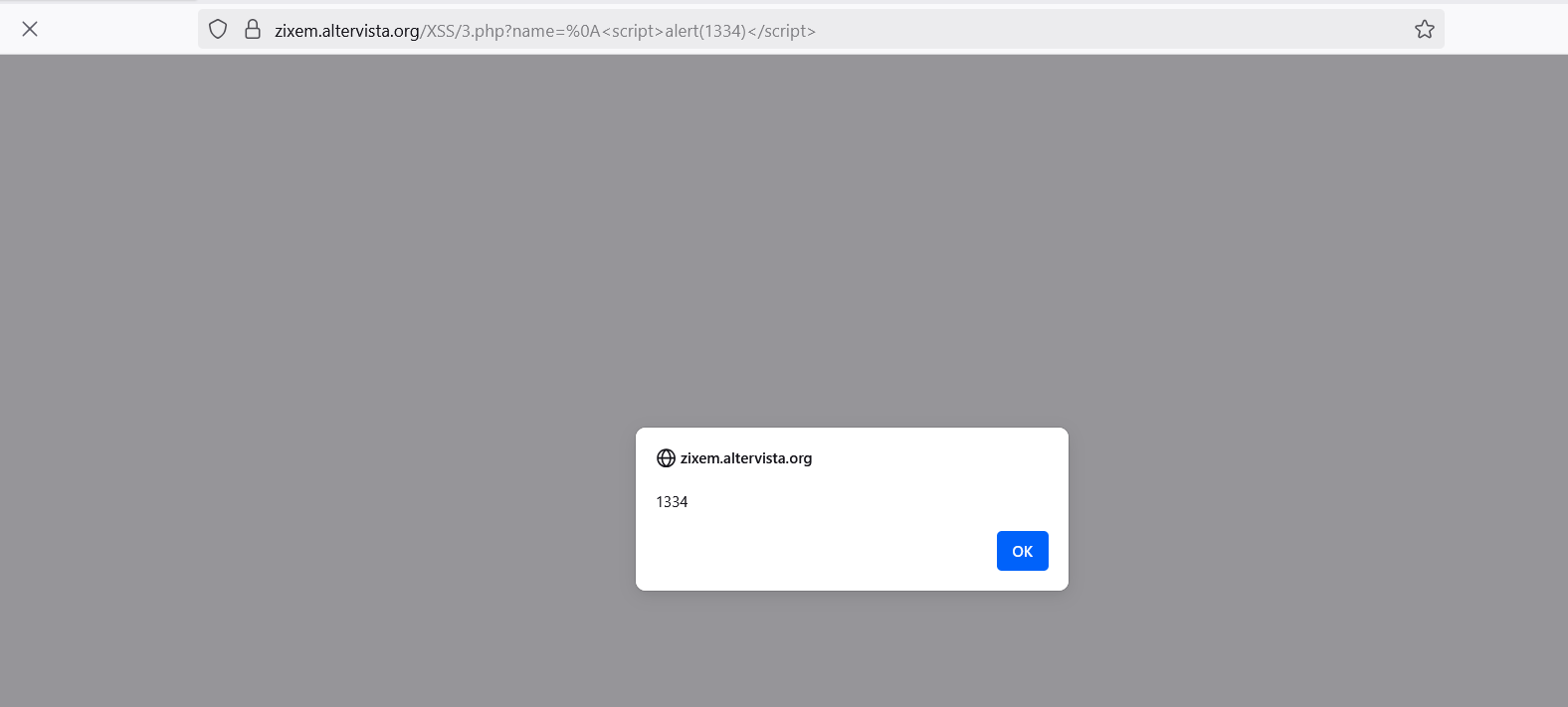
**Step 2:** write something in that field

$ name=security



**Step 3:** We’ll use %0A, the URL-encoded representation of a newline character, to bypass the filter that blocks the word script. By injecting a newline, we shift the script tag to a new line, effectively evading the developer’s restriction.

$ %0A<script>alert(1334)</script>



Let’s do **Level 6** now

**Step 1**: finding the field

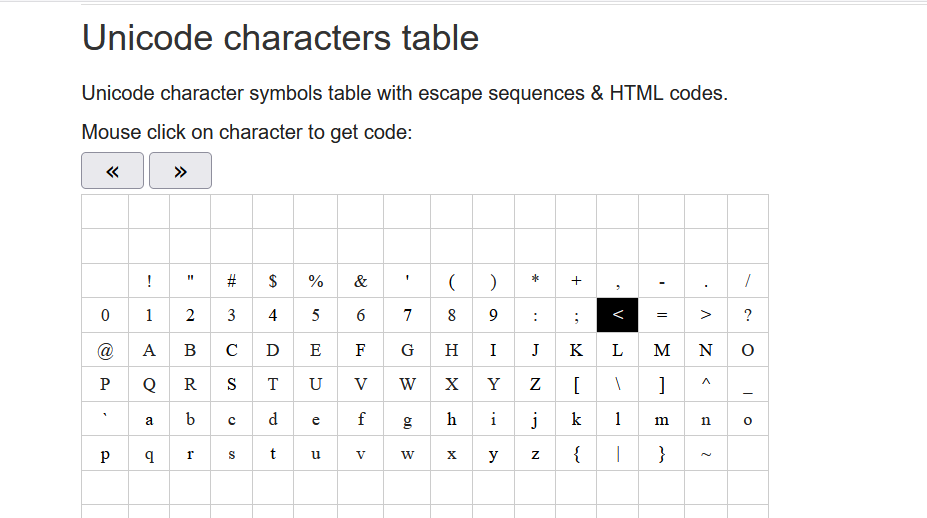


**Step 2:** write hacking in that field

$ name=hacking

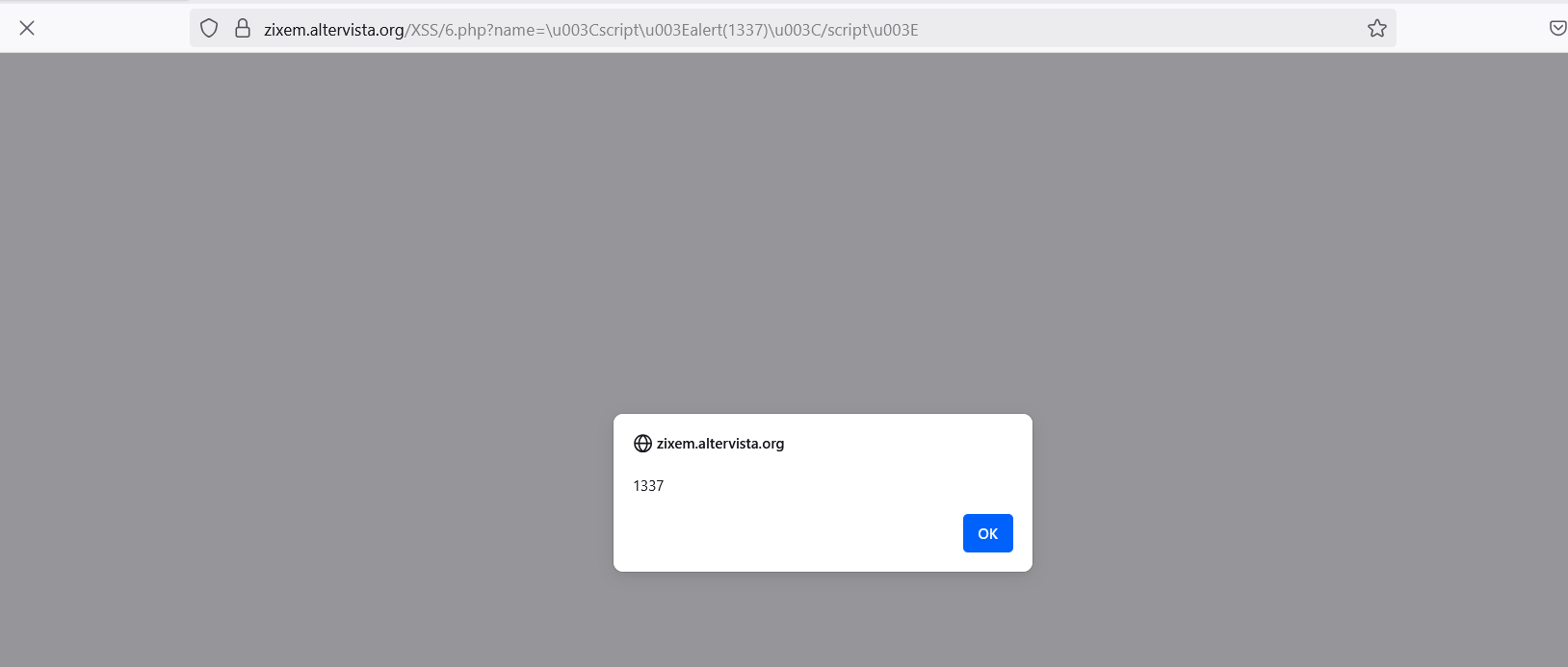


**Step 3:** here in place of < and > we will be using unicode from Rapid table.



In place of < = \u003C and > = \u003E

$ \u003Cscript\u003Ealert(1337)\u003C/script\u003E



**How to Mitigate XSS**

**Input Validation**

Accept only anticipated input via strict whitelisting. Sanitize or block all untrusted or unexpected characters.

**Output Encoding**

Encode all special characters (<, >, ", ', /) beforehand whenever displaying user-provided content inside the browser.

**Content Security Policy (CSP)**

Use a Content Security Policy (CSP) header to prevent inline scripts and restrict script sources to trusted domains.

**Use HTTPOnly Cookies**

Set the HttpOnly flag on cookies to prevent access via JavaScript, reducing the risk of session theft.

**Use Secure Frameworks with Auto-Escaping**

Rely on frameworks like React, Angular, or Django templates that automatically escape output and reduce XSS risk.

**Avoid Dangerous JavaScript Functions**

Do not use innerHTML, eval(), or document.write() with untrusted input. Prefer safe alternatives like textContent or createTextNode.

**Conclusion**

Cross-site scripting (XSS) remains one of the most prevalent and dangerous vulnerabilities in modern web applications. Through this article, we explored the fundamentals of XSS, its various forms, real-world impacts, and step-by-step methods attackers use to exploit it. We also examined how XSS challenges—such as those on the Zix platform—can offer valuable hands-on experience in understanding how these attacks operate and how to defend against them.

Learn how attackers think and act to prepare your applications against such threats. Never forget always to validate inputs, encode outputs, enforce Content Security Policies, and never trust any input coming from users. It secures the prevention of XSS and contributes to a safer web experience for everyone.

Cybersecurity is a journey that must be continually pursued—learn, test, and continuously improve your defenses.