WEB APPLICATION VULNERABILITY.

CROSS SITE SCRIPTING

What is cross-site scripting (XSS)?

How does XSS work?

What are the types of XSS attacks?

Content security policy

How to prevent XSS attacks

There are what is referred to as entry points where malicious payloads are injected to test for web application vulnerability.

How to find and test for reflected XSS vulnerabilities

The vast majority of reflected cross-site scripting vulnerabilities can be found quickly and reliably using Burp Suite's web vulnerability scanner.

Testing for reflected XSS vulnerabilities manually involves the following steps:

Test every entry point. Test separately every entry point for data within the application's HTTP requests. This includes parameters or other data within the URL query string and message body, and the URL file path. It also includes HTTP headers, although XSS-like behavior that can only be triggered via certain HTTP headers may not be exploitable in practice.

Submit random alphanumeric values. For each entry point, submit a unique random value and determine whether the value is reflected in the response. The value should be designed to survive most input validation, so needs to be fairly short and contain only alphanumeric characters. But it needs to be long enough to make accidental matches within the response highly unlikely. A random alphanumeric value of around 8 characters is normally ideal. You can use Burp Intruder's number payloads with randomly generated hex values to generate suitable random values. And you can use Burp Intruder's grep payloads settings to automatically flag responses that contain the submitted value.

Determine the reflection context. For each location within the response where the random value is reflected, determine its context. This might be in text between HTML tags, within a tag attribute which might be quoted, within a JavaScript string, etc.

**Meaning:** Determining the **reflection context** involves analyzing where the reflected value appears within the response, as this affects potential security risks (such as cross-site scripting, or XSS). Here are common contexts where a randomly injected value might appear:

1. **HTML Body (Text Content)**
   * Example: <p>RandomValue</p>
   * Context: Plain text between HTML tags.
   * Risk: Low, unless interpreted as HTML due to improper encoding.
2. **HTML Attribute (Quoted or Unquoted)**
   * Example (quoted): <input type="text" value="RandomValue">
   * Example (unquoted): <div class=RandomValue></div>
   * Context: Inside HTML attributes.
   * Risk: High if injected inside event handlers (e.g., onmouseover="RandomValue") or unquoted attributes.
3. **JavaScript String (Inline or External)**
   * Example (inline script): <script>var userInput = "RandomValue";</script>
   * Example (inside function call): <script>alert(RandomValue);</script>
   * Context: Inside a JavaScript string or function.
   * Risk: High if not properly sanitized, allowing code execution.
4. **JavaScript Object or JSON Response**
   * Example: { "user": "RandomValue" }
   * Context: Inside a JSON response or JavaScript object.
   * Risk: Medium; depends on whether the data is parsed and executed dynamically.
5. **URL or Query Parameter (Reflected in Links)**
   * Example: <a href="https://example.com?search=RandomValue">Search</a>
   * Context: Within a URL or query parameter.
   * Risk: Medium; can lead to open redirects or other attacks if manipulated.
6. **CSS Context (Style Tags or Inline)**
   * Example (inline style): <div style="color: RandomValue;">Text</div>
   * Example (external CSS): body { background: RandomValue; }
   * Context: Inside a CSS declaration.
   * Risk: Low unless used in combination with other vulnerabilities.

Test a candidate payload. Based on the context of the reflection, test an initial candidate XSS payload that will trigger JavaScript execution if it is reflected unmodified within the response. The easiest way to test payloads is to send the request to Burp Repeater, modify the request to insert the candidate payload, issue the request, and then review the response to see if the payload worked. An efficient way to work is to leave the original random value in the request and place the candidate XSS payload before or after it. Then set the random value as the search term in Burp Repeater's response view. Burp will highlight each location where the search term appears, letting you quickly locate the reflection.

Test alternative payloads. If the candidate XSS payload was modified by the application, or blocked altogether, then you will need to test alternative payloads and techniques that might deliver a working XSS attack based on the context of the reflection and the type of input validation that is being performed. For more details, see cross-site scripting contexts

Test the attack in a browser. Finally, if you succeed in finding a payload that appears to work within Burp Repeater, transfer the attack to a real browser (by pasting the URL into the address bar, or by modifying the request in Burp Proxy's intercept view, and see if the injected JavaScript is indeed executed. Often, it is best to execute some simple JavaScript like alert(document.domain) which will trigger a visible popup within the browser if the attack succeeds.

Reflected cross – site scripting labs.

1. Reflected XSS into HTML context with nothing encoded

Payload used: <script>alert('You are hacked')</script>

1. Reflected XSS into attribute with angle brackets HTML-encoded

Payload used: <script>alert('You are hacked')</script> doesn’t work here, view source code for any hints using ctrl plus u. from the source code we saw that the payload used though not executed was reflected in the html attribute so we need to use a payload that is suitable for it, such as " onmouseover="alert('OKOWAN')" x=" which worked out.

1. Reflected XSS into a JavaScript string with angle brackets HTML encoded

Payload used: <script>alert(1)</script> doesn’t work here, view source code for any hints using ctrl plus u. from the source code we saw that the payload used though not executed was reflected in the javascript string so we need to use a payload that is suitable for it, such as ';alert(1);// which worked out.

1. Reflected XSS into HTML context with most tags and attributes blocked
2. Inject a standard XSS payload
3. Observe that this gets blocked. In the next few steps, we'll use Burp Intruder to test which tags and attributes are being blocked.
4. Open Burp's browser and use the search function in the lab. Send the resulting request to Burp Intruder.
5. In Burp Intruder, replace the value of the search term with: <>
6. Place the cursor between the angle brackets and click **Add §** to create a payload position. The value of the search term should now look like: <§§>
7. Visit the [XSS cheat sheet](https://portswigger.net/web-security/cross-site-scripting/cheat-sheet) and click **Copy tags to clipboard**.
8. In the **Payloads** side panel, under **Payload configuration**, click **Paste** to paste the list of tags into the payloads list. Click **Start attack**.
9. When the attack is finished, review the results. Note that most payloads caused a 400 response, but the body payload caused a 200 response.
10. Go back to Burp Intruder and replace your search term with:

<body%20=1>

1. Place the cursor before the = character and click **Add §** to create a payload position. The value of the search term should now look like: <body%20§§=1>
2. Visit the [XSS cheat sheet](https://portswigger.net/web-security/cross-site-scripting/cheat-sheet) and click **Copy events to clipboard**.
3. In the **Payloads** side panel, under **Payload configuration**, click **Clear** to remove the previous payloads. Then click **Paste** to paste the list of attributes into the payloads list. Click **Start attack**.
4. When the attack is finished, review the results. Note that most payloads caused a 400 response, but the onresize payload caused a 200 response.
5. Now that we know the tags and attributes accepted lets use it in the entry point in our case the search field.
6. It would reflect but not execute, copy the url.
7. Go to the exploit server and paste the following code, replacing YOUR-LAB-ID with your lab ID:

<iframe src="the url from step 15 space onload=this.style.width='100px'>

In our case <iframe src="<https://0a6a006f04ad220a81fbd4bf00010060.web-security-academy.net/?search=%3Cbody+onresize%3Dprint%281%29%3E> onload=this.style.width='100px'>

1. Click **Store** and **Deliver exploit to victim**.

Let’s breakdown our exploit serve craft response:

The src attribute points to a URL

The URL-encoded payload %22%3E%3Cbody%20onresize=print()%3E decodes to "><body onresize=print()>(remember we used burp suite intruder to check which tags and attributes are ok, these we have used encoded to bypass sanitization/filtering)

Body onresize event: You're using the onresize event on the <body> tag to execute the print() function when the iframe is resized.

Onload event: You're using the onload event on the iframe to set its width to 100px, which triggers the onresize event.

An iframe (short for inline frame) is an HTML element that allows you to embed another HTML document within a web page.

The use of the **<iframe>** element with a src attribute in a Cross-Site Scripting (XSS) payload is a technique employed to **bypass security mechanisms** like a Web Application Firewall (WAF) or filters.

Some Web Application Firewalls (WAFs) and security filters are designed to **block or sanitize typical XSS vectors** such as <script>, onerror, and onload events. By embedding malicious payloads inside an iframe, attackers can evade these filters because:

* **iframes are often treated as less suspicious** by security tools. They are commonly used for embedding third-party content (like advertisements or embedded media).
* The **payload** injected inside the iframe (such as an event handler or JavaScript) may not be detected or blocked by the WAF since the iframe is an external element, and WAFs may only look for **inline JavaScript** or suspicious attributes directly inside the HTML.