**XSS(DOM)**

"Cross-Site Scripting (XSS)" attacks are a type of injection problem, in which malicious scripts are injected into the otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application using input from a user in the output, without validating or encoding it.An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the JavaScript. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by your browser and used with that site. These scripts can even rewrite the content of the HTML page.

DOM Based XSS is a special case of reflected where the JavaScript is hidden in the URL and pulled out by JavaScript in the page while it is rendering rather than being embedded in the page when it is served. This can make it stealthier than other attacks and WAFs or other protections which are reading the page body do not see any malicious content.

**Objective:** Run your own JavaScript in another user's browser, use this to steal the cookie of a logged in user.

**Description:** DOM Based XSS (or as it is called in some texts, “type-0 XSS”) is an XSS attack wherein the attack payload is executed as a result of modifying the DOM “environment” in the victim's browser used by the original client side script, so that the client side code runs in an “unexpected” manner.

**Impact:** DOM XSS attacks are difficult to detect by server-side attack detection and prevention tools. The malicious payload usually does not reach the server and therefore cannot be sanitized in server-side code. However, the root of the problem still resides in the code of the page, this time in client-side code.

**Prevention:**

* 1. Avoid using data received from the client for client-side sensitive actions such as rewriting or redirection.
  2. Sanitize client-side code by inspecting references to DOM objects that pose a threat, for example, URL, location, and referrer. This is especially important if DOM may be modified.
  3. Use intrusion prevention systems that are able to inspect inbound URL parameters and prevent the inappropriate pages to be served.
  4. Avoid methods such as document.innerHTML and instead use safer functions, for example, document.innerText and document.textContent. If you can, entirely avoid using user input, especially if it affects DOM elements such as the document.url, the document.location, or the document.referrer.

**LOW**

**Steps to reproduce:**

1. Configure your browser

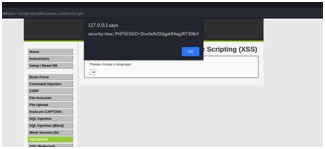
2. Go to the dvwa page and set level of XSS(DOM) to the low level

3. Select any language and click on select button.

4. In URL replace the value parameter default to xss payload that is “<script>alert(1)</script>”

5. Now Press enter, get a pop up of alert 1.

6. For user cookies use this payload “<script>alert(document.cookie)</script>”



**MEDIUM**

**Steps to reproduce:**

1. Configure your browser

2. Go to the dvwa page and set level of XSS(DOM) to the medium level

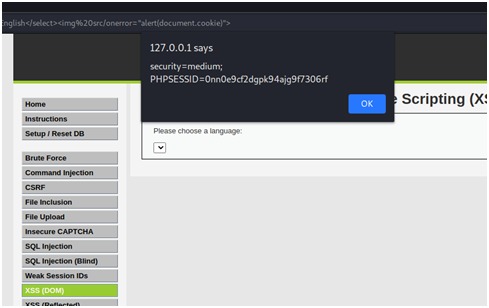
3. Select any language and click on select button.

4. In URL add “</select><imgsrc/orerror=”alert(1)”>” tag with default parameter.

5. Now press enter, get pop up of alert 1.

6. For user cookies use this payload “</select><img

src/onerror=”alert(document.cookie)”>”.



**HIGH**

**Steps to reproduce:**

1. Configure your browser

2. Go to the dvwa page and set level of XSS(DOM) to the high level

3. Select any language and click on select button.

4. In URL add “#<script>alert(1)</script>” tag with default parameter.

5. Now press enter and reload the page, get pop up of alert 1.

6. For user cookies use this payload “#<script>alert(document.cookie)</script>”

