**Cross-Site Scripting (XSS): Reflected Development Mitigation SOP**

XSS vulnerabilities occur when a Web Application includes untrusted data in its output. Because JavaScript is an interpreted language, exploitation occurs in the victim’s Web browser, using its JavaScript interpreter. Vulnerabilities can take the following four forms:

* **Reflected Server-side** – occurs when the injected script is reflected off the web server, either through an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. This attack targets a single user through either phishing or other malicious URL vector.
* **Reflected Client-Side**– occurs when a malicious script is injected into the DOM with an unsafe JavaScript call. The attack could occur through an error message or other content displayed to the user after updating the DOM. DOM-based XSS is a subset of this category in which the attack is located in the DOM.

**Defense Against Server-Side XSS**

The best way to defend against server-side XSS is to use context-sensitive server side output encoding. There are many libraries, which provide this functionality. For instance, JavaServer Pages Standard Tag Library (JSTL) provides the core out tag <c:out>. This tag has an escapeXml attribute that is a Boolean set to true by default. It “determines whether characters <,>,&,'," in the resulting string should be converted to their corresponding character entity codes.”

Input validation and data sanitization are additional techniques for mitigating server-side attacks; specifically at the point of entry. Input validation accepts or rejects data sent to the server based on pattern matching using regular expressions. The filters either use white listing (narrow acceptance, reject all others) or black listing (narrow rejection, accept all others) to define what data is allowed. Data sanitization attempts to remove or encode disallowed characters based on black listing or white listing filters in order to salvage the data. These approaches are much more difficult to get correct. Output encoding neutralizes the payload at the point of attack and therefore is the preferred defense for XSS.

Our system uses OWASP Stinger for input validation. It is implemented as a servlet filter that can intercept requests before they reach the application. Stinger has a configuration file that defines multiple regular expression patterns that are matched against expected parameter values such as a JSESSIONID. It also defines a generic safetext pattern that “Allows lower and upper case letters and all digits, as well as whitespaces, periods, hyphens, underscores, pipes, forward slashes, and ampersands.” Safetext is used by default when no other path pattern is matched and will not allow HTML tag values such as < or >.

**Defense Against Client-Side XSS**

The best defense for client-side XSS is to only use safe JavaScript APIs. If these are not known, context-sensitive output encoding can be used before sending the data to an unsafe JavaScript method.

**Example**

<input type="hidden" id="tzOffset" value='<%=session.getAttribute("tzOffset") %>'/>

**Explanation**

This hidden input field is located in a jsp file. The value attribute is being set by the return value of the expression <%=session.getAttribute("tzOffset")%> which makes it vulnerable to a cross-site scripting attack.

**Recommendation**

This vulnerability was neutralized by changing the expression to:

<input type="hidden" id="tzOffset" value="<c:out value='${sessionScope.tzOffset}'/>"

This causes the value to be output via the c:out function which strips XML.