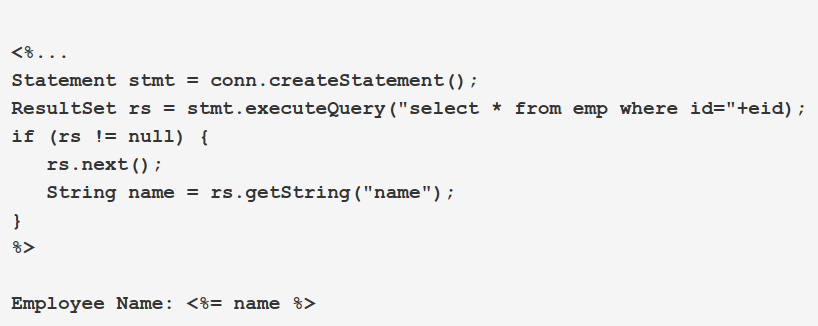
**Cross-Site Scripting (XSS): Persistent Development Mitigation SOP**

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 forms:

* **Persistent Server-side**– occurs when user input is stored on the target server, either in memory or in a database. A victim retrieves the malicious content when requesting a resource from the server. This attack targets multiple users through common resources like forums, user comments and anywhere users can submit content for multiple users to consume.
* **Persistent Client-Side** – could potentially occur with the advent of HTML 5 and the Local Storage API. Local storage is per origin (per domain and protocol). All pages, from one origin, can store and access the same data. Two objects are available: window.localStorage (stores data with no expiration date; is available even if the browser tab is closed) and window.sessionStorage (stores data for one session; not available after the browser tab is closed). The vulnerable client-side JavaScript code would retrieve the attack data from the browser local storage and update the DOM to execute the attack.

**Example**

The following JSP code segment queries a database for an employee with a given ID and prints the corresponding employee's name.



This code functions correctly when the values of name are well-behaved, but it does nothing to prevent exploits if they are not. This code can appear less dangerous because the value of name is read from a database, whose contents are apparently managed by the application. However, if the value of name originates from user-supplied data, then the database can be a conduit for malicious content. Without proper input validation on all data stored in the database, an attacker may execute malicious commands in the user's web browser. This type of exploit, known as Persistent (or Stored) XSS, is particularly insidious because the indirection caused by the data store makes it more difficult to identify the threat and increases the possibility that the attack will affect multiple users. XSS got its start in this form with web sites that offered a "guestbook" to visitors. Attackers would include JavaScript in their guestbook entries, and all subsequent visitors to the guestbook page would execute the malicious code.

**Explanation**

Cross-site scripting (XSS) vulnerabilities occur when:

1. Data enters a web application through an untrusted source. In the case of persistent (also known as stored) XSS, the untrusted source is typically a database or other back-end data store, while in the case of reflected XSS it is typically a web request.
2. The data is included in dynamic content that is sent to a web user without being validated.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash or any other type of code that the browser executes. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

**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: Persistent**

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.

**References**

1. <https://vulncat.fortify.com/en/detail?id=desc.dataflow.java.cross_site_scripting_persistent>
2. <https://owasp.org/www-community/attacks/xss/>