# CWE Detail – CWE-79

## Description

The product does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users.

## Extended Description

There are many variants of cross-site scripting, characterized by a variety of terms or involving different attack topologies. However, they all indicate the same fundamental weakness: improper neutralization of dangerous input between the adversary and a victim.

## Threat-Mapped Scoring

Score: 0.0

Priority: Unclassified

## Observed Examples (CVEs)

**•** CVE-2021-25926: Python Library Manager did not sufficiently neutralize a user-supplied search term, allowing reflected XSS.

**•** CVE-2021-25963: Python-based e-commerce platform did not escape returned content on error pages, allowing for reflected Cross-Site Scripting attacks.

**•** CVE-2021-1879: Universal XSS in mobile operating system, as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2020-3580: Chain: improper input validation (CWE-20) in firewall product leads to XSS (CWE-79), as exploited in the wild per CISA KEV. (KEV)

**•** CVE-2014-8958: Admin GUI allows XSS through cookie.

**•** CVE-2017-9764: Web stats program allows XSS through crafted HTTP header.

**•** CVE-2014-5198: Web log analysis product allows XSS through crafted HTTP Referer header.

**•** CVE-2008-5080: Chain: protection mechanism failure allows XSS

**•** CVE-2006-4308: Chain: incomplete denylist (CWE-184) only checks "javascript:" tag, allowing XSS (CWE-79) using other tags

**•** CVE-2007-5727: Chain: incomplete denylist (CWE-184) only removes SCRIPT tags, enabling XSS (CWE-79)

**•** CVE-2008-5770: Reflected XSS using the PATH\_INFO in a URL

**•** CVE-2008-4730: Reflected XSS not properly handled when generating an error message

**•** CVE-2008-5734: Reflected XSS sent through email message.

**•** CVE-2008-0971: Stored XSS in a security product.

**•** CVE-2008-5249: Stored XSS using a wiki page.

**•** CVE-2006-3568: Stored XSS in a guestbook application.

**•** CVE-2006-3211: Stored XSS in a guestbook application using a javascript: URI in a bbcode img tag.

**•** CVE-2006-3295: Chain: library file is not protected against a direct request (CWE-425), leading to reflected XSS (CWE-79).

## Related Attack Patterns (CAPEC)

* CAPEC-209
* CAPEC-588
* CAPEC-591
* CAPEC-592
* CAPEC-63
* CAPEC-85

## Modes of Introduction

**•** Implementation: REALIZATION: This weakness is caused during implementation of an architectural security tactic.

## Common Consequences

**•** Impact: Bypass Protection Mechanism, Read Application Data — Notes: The most common attack performed with cross-site scripting involves the disclosure of private information stored in user cookies, such as session information. Typically, a malicious user will craft a client-side script, which -- when parsed by a web browser -- performs some activity on behalf of the victim to an attacker-controlled system (such as sending all site cookies to a given E-mail address). This could be especially dangerous to the site if the victim has administrator privileges to manage that site. This script will be loaded and run by each user visiting the web site. Since the site requesting to run the script has access to the cookies in question, the malicious script does also.

**•** Impact: Execute Unauthorized Code or Commands — Notes: In some circumstances it may be possible to run arbitrary code on a victim's computer when cross-site scripting is combined with other flaws, for example, "drive-by hacking."

**•** Impact: Execute Unauthorized Code or Commands, Bypass Protection Mechanism, Read Application Data — Notes: The consequence of an XSS attack is the same regardless of whether it is stored or reflected. The difference is in how the payload arrives at the server. XSS can cause a variety of problems for the end user that range in severity from an annoyance to complete account compromise. Some cross-site scripting vulnerabilities can be exploited to manipulate or steal cookies, create requests that can be mistaken for those of a valid user, compromise confidential information, or execute malicious code on the end user systems for a variety of nefarious purposes. Other damaging attacks include the disclosure of end user files, installation of Trojan horse programs, redirecting the user to some other page or site, running "Active X" controls (under Microsoft Internet Explorer) from sites that a user perceives as trustworthy, and modifying presentation of content.

## Potential Mitigations

**•** Architecture and Design: Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket. (Effectiveness: N/A)

**•** Implementation: Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies. For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters. Parts of the same output document may require different encodings, which will vary depending on whether the output is in the: HTML body Element attributes (such as src="XYZ") URIs JavaScript sections Cascading Style Sheets and style property etc. Note that HTML Entity Encoding is only appropriate for the HTML body. Consult the XSS Prevention Cheat Sheet [REF-724] for more details on the types of encoding and escaping that are needed. (Effectiveness: N/A)

**•** Architecture and Design: Understand all the potential areas where untrusted inputs can enter your software: parameters or arguments, cookies, anything read from the network, environment variables, reverse DNS lookups, query results, request headers, URL components, e-mail, files, filenames, databases, and any external systems that provide data to the application. Remember that such inputs may be obtained indirectly through API calls. (Effectiveness: Limited)

**•** Architecture and Design: For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server. (Effectiveness: N/A)

**•** Architecture and Design: If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated. (Effectiveness: N/A)

**•** Implementation: Use and specify an output encoding that can be handled by the downstream component that is reading the output. Common encodings include ISO-8859-1, UTF-7, and UTF-8. When an encoding is not specified, a downstream component may choose a different encoding, either by assuming a default encoding or automatically inferring which encoding is being used, which can be erroneous. When the encodings are inconsistent, the downstream component might treat some character or byte sequences as special, even if they are not special in the original encoding. Attackers might then be able to exploit this discrepancy and conduct injection attacks; they even might be able to bypass protection mechanisms that assume the original encoding is also being used by the downstream component. The problem of inconsistent output encodings often arises in web pages. If an encoding is not specified in an HTTP header, web browsers often guess about which encoding is being used. This can open up the browser to subtle XSS attacks. (Effectiveness: N/A)

**•** Implementation: With Struts, write all data from form beans with the bean's filter attribute set to true. (Effectiveness: N/A)

**•** Implementation: To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set. (Effectiveness: Defense in Depth)

**•** Implementation: Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a list of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if the input is only expected to contain colors such as "red" or "blue." Do not rely exclusively on looking for malicious or malformed inputs. This is likely to miss at least one undesirable input, especially if the code's environment changes. This can give attackers enough room to bypass the intended validation. However, denylists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. When dynamically constructing web pages, use stringent allowlists that limit the character set based on the expected value of the parameter in the request. All input should be validated and cleansed, not just parameters that the user is supposed to specify, but all data in the request, including hidden fields, cookies, headers, the URL itself, and so forth. A common mistake that leads to continuing XSS vulnerabilities is to validate only fields that are expected to be redisplayed by the site. It is common to see data from the request that is reflected by the application server or the application that the development team did not anticipate. Also, a field that is not currently reflected may be used by a future developer. Therefore, validating ALL parts of the HTTP request is recommended. Note that proper output encoding, escaping, and quoting is the most effective solution for preventing XSS, although input validation may provide some defense-in-depth. This is because it effectively limits what will appear in output. Input validation will not always prevent XSS, especially if you are required to support free-form text fields that could contain arbitrary characters. For example, in a chat application, the heart emoticon ("<3") would likely pass the validation step, since it is commonly used. However, it cannot be directly inserted into the web page because it contains the "<" character, which would need to be escaped or otherwise handled. In this case, stripping the "<" might reduce the risk of XSS, but it would produce incorrect behavior because the emoticon would not be recorded. This might seem to be a minor inconvenience, but it would be more important in a mathematical forum that wants to represent inequalities. Even if you make a mistake in your validation (such as forgetting one out of 100 input fields), appropriate encoding is still likely to protect you from injection-based attacks. As long as it is not done in isolation, input validation is still a useful technique, since it may significantly reduce your attack surface, allow you to detect some attacks, and provide other security benefits that proper encoding does not address. Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere. (Effectiveness: N/A)

**•** Architecture and Design: When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs. (Effectiveness: N/A)

**•** Operation: Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth. (Effectiveness: Moderate)

**•** Operation: When using PHP, configure the application so that it does not use register\_globals. During implementation, develop the application so that it does not rely on this feature, but be wary of implementing a register\_globals emulation that is subject to weaknesses such as CWE-95, CWE-621, and similar issues. (Effectiveness: N/A)

## Applicable Platforms

**•** None (Class: Not Language-Specific, Prevalence: Undetermined)

## Demonstrative Examples

**•** Because the parameter can be arbitrary, the url of the page could be modified so $username contains scripting syntax, such as

**•** The following JSP code segment reads an employee ID, eid, from an HTTP request and displays it to the user.

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

**•** CreateUser.php

**•** An attacker may be able to perform an HTML injection (Type 2 XSS) attack by setting a cookie to a value like:

## Notes

**•** Other: The attack methods for XSS can vary depending on the type of XSS and the attacker’s goal. Reflected XSS exploits (Type 1) occur when an attacker causes a victim to supply dangerous content to a vulnerable web application, which is then reflected back to the victim and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to the victim. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces a victim to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the victim, the content is executed by the victim's browser. In a Stored XSS exploit (Type 2) , the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user. For example, the attacker might inject XSS into a log message, which might not be handled properly when an administrator views the logs. DOM-based XSS (Type 0) generally involves server-controlled, trusted script that is sent to the client, such as JavaScript that performs sanity checks on a form before the user submits it. If the server-supplied script processes user-supplied data and then injects it back into the web page (such as with dynamic HTML), then DOM-based XSS is possible.

**•** Other: Attackers frequently use a variety of methods to encode the malicious portion of the attack, such as URL encoding or Unicode, so the request looks less suspicious. Phishing attacks could be used to emulate trusted web sites and trick the victim into entering a password, allowing the attacker to compromise the victim's account on that web site.

**•** Other: Cross-site scripting (XSS) vulnerabilities occur when: Untrusted data enters a web application, typically from a web request. The web application dynamically generates a web page that contains this untrusted data. During page generation, the application does not prevent the data from containing content that is executable by a web browser, such as JavaScript, HTML tags, HTML attributes, mouse events, Flash, ActiveX, etc. A victim visits the generated web page through a web browser, which contains malicious script that was injected using the untrusted data. Since the script comes from a web page that was sent by the web server, the victim's web browser executes the malicious script in the context of the web server's domain. This effectively violates the intention of the web browser's same-origin policy, which states that scripts in one domain should not be able to access resources or run code in a different domain.

**•** Relationship: There can be a close relationship between XSS and CSRF (CWE-352). An attacker might use CSRF in order to trick the victim into submitting requests to the server in which the requests contain an XSS payload. A well-known example of this was the Samy worm on MySpace [REF-956]. The worm used XSS to insert malicious HTML sequences into a user's profile and add the attacker as a MySpace friend. MySpace friends of that victim would then execute the payload to modify their own profiles, causing the worm to propagate exponentially. Since the victims did not intentionally insert the malicious script themselves, CSRF was a root cause.

**•** Applicable Platform: XSS flaws are very common in web applications, since they require a great deal of developer discipline to avoid them.