THE ULTIMATE XSS PROTECTION CHEATSHEET FOR DEVELOPERS V1.0

Ajin Abraham
Author of OWASP Xenotix XSS Exploit Framework | opensecurity.in

The quick guide for developers to protect their web applications from XSS.

DISCLAIMER

The **Ultimate XSS Protection Cheat Sheet for Developers** is a compilation of information available on XSS Protection from various organization, researchers, websites, and my own experience. This document follows a simple language and justifying explanations that helps a developer to implement the correct XSS defense and to build a secure web application that prevents XSS vulnerability and Post XSS attacks. It will also discuss about the existing methods or functions provided by various programming languages to mitigate XSS vulnerability. This document will be updated regularly in order to include updated and correct in information in the domain of XSS Protection.

Quick Introduction: What is XSS?

XSS or Cross Site Scripting is a web application vulnerability that occurs when untrusted data from the user is processed by the web application without validation and is reflected back to the browser without encoding or escaping, resulting in code execution at the browser engine.

Types of XSS

- Reflected or Non-Persistent XSS
- Stored or Persistent XSS
- DOM based XSS
- mXSS or Mutation XSS

Reflected or Non- Persistent XSS

Reflected or Non-Persistent XSS is a kind of XSS vulnerability where the untrusted user input is immediately processed by the server without any validation and is reflected back in the response without encoding or escaping resulting in code execution at the browser.

Stored or Persistent XSS

Stored or Persistent XSS is a kind of XSS vulnerability where the untrusted user input is processed and stored by the server in a file or database without any validation and this untrusted data is fetched from the storage and is reflected back in response without encoding or escaping resulting in permanent code execution at the browser whenever the stored data is reflected in the response.

DOM based XSS

DOM Based XSS is a form of client side XSS which occurs in an environment where the source of the data is in the DOM, the sink is also in the DOM, and the data flow never leaves the browser. It occurs when an untrusted data is given at the source is executed as a result of modifying the DOM "environment" in the browser. DOM XSS occurs when the untrusted data is not in escaped or encoded form with respect to the context.

mXSS or mutation XSS

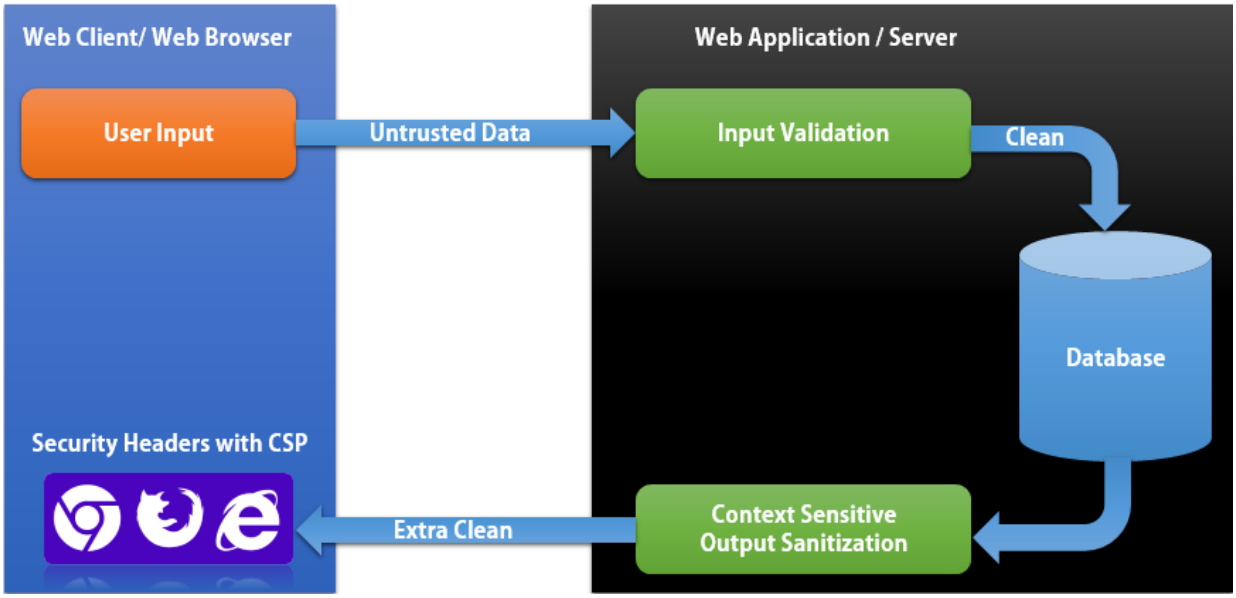
mXSS or Mutation XSS is a kind of XSS vulnerability that occurs when the untrusted data is processed in the context of DOM's innerHTML property and get mutated by the browser,

resulting as a valid XSS vector. In mXSS an user specified data that appears harmless may pass through the client side or server side XSS Filters if present or not and get mutated by the browser's execution engine and reflect back as a valid XSS vector. XSS Filters alone won't protect from mXSS. To prevent mXSS an effective CSP should be implemented, Framing should not be allowed, HTML documents should specify the document type definition that enforce the browser to follow a standard in rendering content as well as for the execution of scripts.

XSS Protection

XSS can be mitigated if you can implement a web application that satisfies the following rules.

1. Validate the Input and Escape untrusted data based on context and in correct order



Input Validation

All the untrusted data should be validated against the web application's logic before processing or moving it into storage. Input validation can prevent XSS in the initial attempt itself.

Parsing Order in Browser

HTML Parser >> CSS Parser >> JavaScript Parser

Decoding Order in Browser

HTML Decoding >> URL Decoding >> JavaScript Decoding

Decoding and Parsing order means a lot. If the encoding or decoding of the untrusted data is done in the wrong order or wrong context, again there is a chance of occurrence of XSS vulnerabilities. The encoding or escaping required for different context is different and the order in which these encoding should be done depends on the logic of the application.

A typical untrusted data can be reflected in html context, html attribute context, script variable context, script block context, REST parameter context, URL context, style context etc. Different kind of escaping methodologies has to be implemented with different context for ensuring XSS Protection.

Order and Context Sensitive Escaping

1. String in the context of HTML

For untrusted string in the context of HTML, do HTML escape.

Example:

```
1. <h1> Welcome html_escape(untrusted string) </html>
```

Symbols	Encoding
&	&amp;
<	&lt;
>	&gt;
"	&quot;
`	&#x60;
'	&#x27;
/	&#x2F;

2. String in the context of HTML Attribute

For untrusted string in the context of HTML attribute, do HTML escape and always quote your attributes, either (' or ") never use backticks (`).

Example:

```
1. <img src="x" alt="html_escape(untrusted string)">
```

Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the **&#xHH;** format (or a named entity if available) to prevent switching out of the attribute. Properly quoted attributes can only be escaped with the corresponding quote. Unquoted attributes can be broken out of with many characters, including **[space] % * + , - / ;** < = > ^ and |.

3. String in the context of Event Handler Attribute and JavaScript

For untrusted string in the context of Event Handler Attribute, do JavaScript Escape first and then perform HTML escape since the Browser performs HTML attribute decode before JavaScript string decode. For untrusted string in the context of JavaScript, do JavaScript String Escape. And always quote your attributes, either (' or ") never use backticks (`).

Example:

```
1. //In the context of Event Handler
2. <img src="x" onload="jsFunction('html_escape(
   javascript_string_escape(untrusted string))')">
3. //In the context of JavaScript
4. <script type="text/javascript">
5. var abc = 'javascript_string_escape(untrusted string)';
6. </script>
```

Except for alphanumeric characters, escape all characters less than 256 with the **\xHH** format to prevent switching out of the data value into the script context or into another attribute. Do not use any escaping shortcuts like **\"** because the quote character may be matched by the HTML attribute parser which runs first. These escaping shortcuts are also susceptible to "escape-the-escape" attacks where the attacker sends **\"** and the vulnerable code turns that into **\\"** which enables the quote. If an event handler attribute is properly quoted, breaking out requires the corresponding quote. Unquoted attributes can be broken out of with many characters including **[space] % * + , - / ; < = > ^** and **|.** Also, a **</script>** closing tag will close a script block even though it is inside a quoted string. Note that the HTML parser runs before the JavaScript parser.

4. URL Path in the context of HTML Attribute

For untrusted URL path string in the context of HTML Attribute, do URL Escape the path and not the full URL. Always quote your attributes, either (' or ") never use backticks (`). Never allow **href** or **src** to include schemes like **javascript:** or **data:** or their tricky combinations like (**javas	cript**)

Example:

```
1. <a href="http://xy.com/index?test=url_escape(untrusted string)">l</a>
2. <img src="/url_escape(untrusted url path string)">
3. <form action="/?i=url_escape(untrusted string)" method="GET"></form>
```

Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the **%HH** escaping format. If **href** or **src** attribute is properly quoted, breaking out requires the corresponding quote. Unquoted attributes can be broken out of with many characters including **[space] % * + , - / ; < = > ^** and **|**. Note that entity encoding is useless in this context.

5. String in the context of HTML style attribute and CSS

For untrusted string in the context of HTML style attribute, do CSS String Escape first and then HTML escape the string since order of parsing is HTML Parser first and then CSS Parser. Always quote your attributes and in this case quote style attribute with (") and CSS

string with (') and never use backticks (` ). For untrusted string in the context of CSS, do CSS String Escape. Also make sure that the untrusted string is within the quotes ( ' or " ) and never use backticks ( `). Do not allow expression and its tricky combinations like (expre/**/ssion).

Example:

```
1. //In the context of HTML style Attribute
2. <p style="font-
   family:'html_escape(css_string_escape(unntrusted string))'">
3. Hello World!
4. </p>
5. //In the context of CSS
6. <style>
7. #css_string_escape(untrusted string)
8. {
9.     text-align: center;
10.     color: red;
11. }
12. </style>
```

Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the **\HH** escaping format. Do not use any escaping shortcuts like **\"** because the quote character may be matched by the HTML attribute parser which runs first. These escaping shortcuts are also susceptible to "escape-the-escape" attacks where the attacker sends **\"** and the vulnerable code turns that into **\\"** which enables the quote. If attribute is quoted, breaking out requires the corresponding quote. Unquoted attributes can be broken out of with many characters including **[space] % * + , - / ; < = > ^** and **|**. Also, the **</style>** tag will close the style block even though it is inside a quoted string and note that the HTML parser runs before the CSS parser.

6. HTML in the context of JavaScript

For untrusted HTML in the context of JavaScript string, do HTML Escape first and then JavaScript String Escape, preserving the order.

Example:

```
1. <script>
2. function xyz()
3. {
4. var elm=document.getElementById("disp");
5. elm.innerHTML="<strong>javascript_string_escape(html_escape(
   untrusted string))</strong>";
6. }
7. </script>
8. <body onload=xyz()>
```

```
9. <div id="disp"></div></body>
```

2. Always follow a Whitelist approach than a Blacklist approach.

Make a whitelist of allowed tags and attributes that the web application should accept from the user. Blacklists can be easily bypassed.

3. Use UTF-8 as the default character encoding and content as text/html.

In HTML documents you can specify it in meta tag like **<meta http-equiv="content-type" content="text/html; charset=UTF-8">**

4. Don't place the text user can control before <meta> tag. Injections can result in XSS by using a different charset.

Injections before meta tag can overwrite the default charset and allows wide range of characters to create a valid XSS vector.

Demo

http://opensecurity.in/labz/opensecurity_meta.html

5. Use <!DOCTYPE html>

DOCTYPE (DTD or Document Type Declaration) tells your browser to follow the standard in rendering the HTML, CSS as well as how to execute scripts. Always use**<!doctype html>** before **<html>**.

6. Use recommended HTTP Response Headers for XSS Protection

HTTP Response Headers	Description
X-XSS-Protection: 1; mode=block	This header will enable the browser's inbuilt Anti-XSS filter.
X-Frame-Options: deny	This header will deny the page from being loaded into a frame.
X-Content-Type-Options: nosniff	This header will prevents the browser from doing MIME-type sniffing.
Content-Security-Policy: default-src 'self'	This header is one of the most effective solution for

	preventing XSS. It allows us to enforce policies on loading objects and executing it from URLs or contexts.
Set-Cookie: key=value; HttpOnly	The Set-Cookie header with the HttpOnly flag will restrict JavaScript from accessing your cookies
Content-Type: type/subtype; charset=utf-8	Always set the appropriate Content Type and Charset. Example: HTTP Response in the case of json, use application/json, for plaintext use text/plain for html, use text/html etc along with charset as utf-8.

7. Prevent CRLF Injection/ HTTP Response Splitting.

Sanitize and encode all user supplied data properly before passing out through HTTP headers. CRLF Injection can destroy and Bypass all your security headers like CSP, X-XSS Protection etc.

8. Disable TRACE and other unnecessary methods.

TRACE is an HTTP method used for debugging which will reflect the request headers from the client, back to the client in HTTP Response. Injections in request header can result in XSS when TRACE method is used.

Quick and Easy Guide for Implementing Content Security Policy (CSP) in an Effective way.

CSP or Content-Security Policy will enforce policies on browser which specifies what resource a browser should load and from where the browser should load the resource along with specifying the loading behavior of a resource defined by a directive. This documentation will give you a better idea on how you can define a CSP based on your requirements.

The CSP directives of our interest are:

Directive	Description
default-src	This directive specifies the loading policy for all resources type in case of a resource type specific directive is not defined.
script-src	This directive specifies the domain(s) or URI from which the web application can load scripts.
object-src	This directive specifies the domain(s) or URI from which the web application can load plugins like Flash.
style-src	This directive specifies the domain(s) or URI from which the web application can load CSS stylesheets.
img-src.	This directive specifies the domain(s) or URI from which the web application can load images.
media-src	This directive specifies the domain(s) or URI from which the web application can load video or audio.
frame-src	This directive specifies the domain(s) or URI that the web application can load inside a frame.
font-src	This directive specifies the domain(s) or URI from which the web application can load fonts.
connect-src.	This directive specifies the domain(s) or URI to which you can connect using script interfaces like XHR, WebSockets, and EventSource.
plugin-types	This directive specifies the MIME types of content for which plugins could be loaded. (Not yet properly supported by the latest browsers)
form-action	This directive specifies the URIs to which HTML form submissions can be done.(Not yet properly supported by the latest browsers)
reflected-xss	This directive tells browser to activate or deactivate any heuristics used to filter or block reflected cross-site scripting attacks and is equivalent to the effects of the X-XSS-Protection header. (Not yet properly supported by the latest browsers). reflected-xss block is equivalent to X-XSS-Protection: 1; mode=block

There are four source expressions.

Source expressions	Description
none	Matches with nothing.
self	Matches only from the current domain excluding sub domains.
unsafe-inline	Allows inline JavaScript and CSS. You shouldn't use this unless you are sure that a non-sanitized user input is never reflected back inline.
unsafe-eval	Allows eval() in JavaScript. You shouldn't use this unless you are sure that a non sanitized or dangerous user input is never inserted into the eval() function.

A typical modern web application requires unsafe-inline and unsafe-eval sources with script-src directive for proper functioning.

A CSP header like (Content-Security-Policy: default-src 'self') cannot be applicable for most of the modern web applications.
This (default-src 'self') policy means that fonts, frames, images, media, objects, scripts, and styles will only load from the same domain or same origin, and connections will only be made to the same origin. However this is not feasible for most of the modern web applications because for example web applications may use Google fonts, shows a Slideshare document in a frame, or include scripts for embedding Twitter or Facebook widgets or for loading jQuery library. So developers tends to avoid CSP thinking that it is complex to implement or implement CSP in a wrong way.

We can always override the default-src directive and use it effectively for implementing a CSP that is suitable for our needs as well as prone to XSS attacks.

Consider a typical CSP

Content-Security-Policy: default-src 'self'; style-src 'unsafe-inline' 'self' http://fonts.googleapis.com http://themes.googleusercontent.com; frame-src http://www.slideshare.net www.youtube.com twitter.com; object-src 'none'; font-src 'self' data: http://themes.googleusercontent.com http://fonts.googleapis.com; script-src 'unsafe-eval' 'unsafe-inline' 'self' http://www.google.com twitter.com http://themes.googleusercontent.com; img-src 'self' http://www.google.com data: https://pbs.twimg.com http://img.youtube.com twitter.com

Explanation of each directives and source expressions

Policy	Description
default-src 'self';	Allows fonts, frames, images, media, objects, scripts, and styles to be load only from the same domain
style-src 'unsafe-inline' 'self' http://fonts.googleapis.com http://themes.googleusercontent.com;	Allows using inline style or stylesheet from same domain, http://fonts.googleapis.com and http://themes.googleusercontent.com
frame-src youtube.com twitter.com;	Allows the web application to load frames only from youtube.com and twitter.com.
object-src 'none';	Allows no objects.
font-src 'self' data: http://themes.googleusercontent.com	Allows the web application to load font from same domain and http://themes.googleusercontent.com
script-src 'unsafe-eval' 'unsafe-inline' 'self' http://www.google.com twitter.com http://themes.googleusercontent.com;	Allows the web application to load script from same domain, http://www.google.com, twitter.com and http://themes.googleusercontent.com. The 'unsafe-inline' source expression allows execution of inline JavaScript and 'unsafe-eval' source expression allows the eval() function in JavaScript. (dangerous)
img-src 'self' data: http://www.google.com https://pbs.twimg.com http://img.youtube.com twitter.com	Allows the web application to load images from same domain, data URI, http://www.google.com, https://pbs.twimg.com, http://img.youtube.com, and twitter.com

XSS Protection Mechanisms available in Web Application Development Environment

XSS Protection in JavaScript

A library for encoding in JavaScript is Encoder.js. It provides various methods for escaping.

Method	Description
HTML2Numerical	This method converts HTML entities to their numerical equivalents.
numEncode	This method numerically encodes unicode characters.
htmlEncode	This method encodes HTML to either numerical or HTML entities. This is determined by the EncodeType property.
XSSEncode	This method encodes the basic characters used in XSS attacks to malform HTML.
correctEncoding	This method corrects any double encoded ampersands.
stripUnicode	This method removes all unicode characters.

Documentation

http://www.strictly-software.com/htmlencode

Download

http://www.strictly-software.com/scripts/downloads/encoder.js
or
https://gist.github.com/ajinabraham/1af8216dfb6f959503e0

DOMPurify is a DOM-only XSS sanitizer for HTML, MathML and SVG. It prevents DOM clobbering and supports whitelisting. It can be used with other JavaScript frameworks.

Download

https://github.com/cure53/DOMPurify

Usage:

```
1. <script type="text/javascript" src="purify.js"></script>
2. var clean = DOMPurify.sanitize("<p>text<iframe/\/src=jAva script:alert(
   3)>"); //becomes <p>text</p>
3. alert(clean);
```

node.js

js-xss is a library for escaping. It also includes a whitelist.

Download
https://github.com/leizongmin/js-xss

XSS is a node.js module for escaping and sanitizing.

Install
$ npm install xss

Usage:

```
1. var xss = require('xss');
2. var html = xss('<script>alert("xss");</script>');
3. console.log(html);
```

jQuery

For escaping, use **.text()** method in jQuery instead of **.html()** method.

yui

Use **html()** method to encode (**& < > " ' /** `). It also escapes backticks ( `) character since IE interprets it as an attribute delimiter.

mootools

mootools provide **escapeRegExp()** method that will escapes all regular expression characters from the string.
stripScripts() method strips the string of its tags and any string in between them. Never use **stripScripts(true)** as this method will evaluate the string before striping.

Vulnerable Example

```
1. "<script>alert(1)</script>".stripScripts(true); // This method will
   trigger the alert() function and then strip the string.
```

Safe Example

```
1. "<script>alert(1)</script>".stripScripts(); //This method will strip
   the "<script></script>" tags including "alert(1)".
```

The **JSON.decode()** converts a JSON string into a JavaScript object and checks for any hazardous syntax and returns null if any found.

Backbone.js

For escaping HTML, use **model.escape()** model
If you are using Underscore template, for escaping HTML, use **<%-** instead of **<%=**. For escaping HTML use **<%-string%>**. Never use **<%=string%>** as it will evaluate the string.

Spinejs

The syntax for escaping in spine should be used with care.
For escaping use **<%= @string %>** it will escape the string and print it.
In contrary to Backbone.js **<%- string %>** will evaluate the string and print its return value without escaping it. So don't get confused if you are using both JS Frameworks.

AngularJS

For XSS protection, AngularJS uses Strict Contextual Escaping (SCE). SCE also allows whitelisting. Technically the **$sce** service provides XSS Protection. We need to include **$sce** service to the code. SCE defines a trust in different context. Consider the following context in SCE.

SCE Service	Description
$sce.HTML	For HTML that's safe to source into the application. The ngBindHtml directive uses this context for bindings. If an unsafe value is encountered and the $sanitize module is present this will sanitize the value instead of throwing an error.
$sce.CSS	For CSS that's safe to source into the application. Currently unused. You can use it in your own directives.
$sce.URL	For URLs that are safe to follow as links. Currently unused (<a href= and <img src= sanitize their urls and don't constitute an SCE context.

$sce.RESOURCE_URL	For URLs that are not only safe to follow as links, but whose contents are also safe to include in your application. Examples include ng-include, src / ngSrc bindings for tags other than IMG (e.g. IFRAME, OBJECT, etc.)

Note that $sce.RESOURCE_URL makes a stronger statement about the URL than $sce.URL does and therefore contexts requiring values trusted for $sce.RESOURCE_URL can be used anywhere that values trusted for $sce.URL are required. $sce.JS For JavaScript that is safe to execute in your application's context and is currently unused.

Documentation

https://docs.angularjs.org/api/ng/service/$sce

XSS Protection in PHP

htmlspecialchars(string, flag, charset) - This function encodes only (**< > " &**). It also encodes (**'**) into (**'**) if the ENT_QUOTES flag is given. It is always safe to stay with 'UTF-8' as the character set. UTF-8 is the default character set starting from PHP 5.4. This version also supports some new flags other than ENT_QUOTES like

Flags	Description
ENT_HTML401	Handle code as HTML 4.01.(default)
ENT_XML1	Handle code as XML 1.
ENT_XHTML	Handle code as XHTML.
ENT_HTML5	Handle code as HTML 5.

Example:

```
1. echo htmlspecialchars($string, ENT_QUOTES | ENT_XHTML, 'UTF-8');
```

It is important to note that **htmlspecialchars()** cannot prevent XSS in the context of JavaScript, style, and URL context.
urlenconde() can be used for encoding URLs.
Use **utf8_encode()** function in PHP for encoding user specified data into UTF-8.

If you are echoing something with the php function **json_encode(),** a safer way to use that function will be like
echo json_encode($string, JSON_HEX_QUOT|JSON_HEX_TAG|JSON_HEX_AMP|JSON_HEX_APOS);
along with the Content-Type header set to **application/json; charset=utf-8**

HTML Purifier is one satisfactory PHP library for preventing XSS and also works based on a whitelist.
It is simple and easy to configure and use.

```
require_once '/path/to/HTMLPurifier.auto.php';
$config = HTMLPurifier_Config::createDefault();
$purifier = new HTMLPurifier($config);
$clean = $purifier->purify($output_to_be_reflected_at_browser);
echo $clean;
```

Documentation

http://htmlpurifier.org/live/INSTALL

PHPIDS (PHP-Intrusion Detection System) is a security layer for your PHP based web application. The IDS neither strips, sanitizes nor filters any malicious input, it simply recognizes when an attacker tries to break your site and reacts in exactly the way you want it to.

More Information

https://phpids.org/

Templating Framework in PHP: Smarty

Smarty provides the variable modifier escape. It is used to encode or escape a variables in the context of html, url, single quotes, hex, hexentity, javascript and mail. It is html by default.

Escape Modifier	Description
{$string|escape,'UTF-8'}	Default:html, Basic HTML Encoding which escapes (**&"'<>**)
{$string|escape:'html','UTF-8'}	Basic HTML Encoding which escapes (**&"'<>**)
{$string|escape:'htmlall','UTF-8'}	HTML Encoding (escapes all html entities)
{$string|escape:'url'}	URL Encoding

{$string|escape:'quotes'}	Escaping Quotes
{$string|escape:"hex"}	Hex Encoding
{$string|escape:"hexentity"}	Hex Entity Encoding
{$string|escape:'mail'}	Converts Email to Text
{$string|escape:'javascript'}	Escapes JavaScript, and keep in mind this implementation supports only strings.

XSS Protection in JAVA

Use OWASP Java Encoder that supports encoding user given data in Basic HTML Context, HTML Content Context, HTML Attribute context, JavaScript Block context, JavaScript Variable context, URL parameter values, REST URL parameters, and Full Untrusted URL.

Documentation

https://www.owasp.org/index.php/OWASP_Java_Encoder_Project#tab=Use_the_Java_Encoder_Project

Coverity Security Library (CSL) is a set of escaping routines for fixing cross-site scripting (XSS) in Java web applications. It supports Java Expression Language (EL) notation and plain Java functions.

Escape Methods	Description
cov:htmlEscape(string)	Performs HTML escape
cov:jsStringEscape(string)	Performs JavaScript string escape
cov:asURL(string)	Performs URL escape and sanitize dangerous scheme like **javascript:**
cov:cssStringEscape(string)	Performs CSS String escape
cov:asNumber(string)	Checks if the input string is a number else the default value will be zero
cov:asCssColor(string)	Allows a string with color specified as text or hex and prevents injection.
cov:uriEncode(name)	Performs URL Encoding

Documentation and Download

https://github.com/coverity/coverity-security-library

OWASP ESAPI (The OWASP Enterprise Security API) is a free, open source, web application security control library that can sanitize untrusted data.

Method	Description
ESAPI.encoder().encodeForHTML()	Escape HTML
ESAPI.encoder().encodeForHTMLAttribute()	Escape HTML Attribute
ESAPI.encoder().encodeForJavaScript()	Escape JavaScript String
ESAPI.encoder().encodeForCSS()	Escape CSS String
ESAPI.encoder().encodeForURL()	Escape URL

Documentation

https://code.google.com/p/owasp-esapi-java/wiki/Welcome?tm=6

XSS Protection in .NET

The **HttpUtility** Class (**System.Web.HttpUtility**) in .NET provides various methods for escaping untrusted data.

Methods	Description
HtmlEncode()	HTML Encoding.
HtmlAttributeEncode()	Basic HTML Encoding. It Encodes only (**"** **&** **<** ****).
UrlEncode()	URL Encoding.
JavaScriptStringEncode()	JavaScript string encoding.

For normal web application **HttpUtlity** Class is enough. But for web application that deals with reflecting data to XML and other contexts, there is **AntiXssEncoder** Class (**System.Web.Security.AntiXss.AntiXssEncoder**) with added advantages like whitelist and is inbuilt with .NET 4.5. It includes the following methods for XSS Protection.

Methods	Description
HtmlEncode()	Html Encoding and optionally specifies whether to use HTML 4.0 named entities.
HtmlAttributeEncode()	Encodes the data to be reflected in an HTML attribute.
HeaderNameValueEncode()	Encodes a header name and value into a string that can be used as an HTTP header.

HtmlFormUrlEncode()	Encodes the data for use in form submissions whose MIME type is "application/x-www-form-urlencoded" and optionally specifies the character encoding.
JavaScriptStringEncode()	JavaScript string encoding.
UrlEncode()	URL Encoding and optionally specifies the character encoding.
UrlPathEncode()	Encodes the path for using in a URL.
XmlAttributeEncode() & XmlEncode()	Encodes the data for use in XML attributes.

For older .NET version, Install Microsoft Web Protection Library from http://wpl.codeplex.com/

XSS Protection in Python django

Function	Description
{{ string }}	django is a having auto escaping feature. This will escape the string.
escape()	This function will escape (**&"'<>**)
conditional_escape()	This function is similar to **escape()** function, except that it doesn't operate on pre-escaped strings, so it will not double escape.
urlencode()	This function can be used for URL encoding.

XSS Protection in Ruby on Rails

Method	Description
sanitize()	This method can be used to sanitize/html encode the user specified data and is supported by a whitelist. It also strips **href/src** tags with invalid protocols, like **javascript:** especially. It does its best to counter any tricks used, like throwing in unicode/ascii/hex values to get past the **javascript:** filters.
sanitize_css()	This method will sanitize the string so that you can safely use it in the context of CSS.
strip_links()	This method will strip all link tags from a string.
h() or html_escape()	This method will encode (**& < > " '**) into (**& < > " ')** respectively.
html_escape_once()	This method is similar to **html_escape()** with a difference that it will encode everything that was not previously encoded.
json_escape()	This method will encode **(& > < \u2028 \u2029**) into (**\u0026 \u003e \u003c \u2028 \u2029)** respectively. Also keep in mind this method only works with valid JSON. Using this on non-JSON values can result in XSS.

Using Sanitize

You can specify a whitelist with **sanitize()** method like

<%= sanitize @article.body, tags: %w(table tr td), attributes: %w(id class style) %>

This will strip out anything other than the mentioned tags and attributes.

Using Strip Links

strip_links('OpenSecurity')
=> OpenSecurity

strip_links('Please e-mail me at email@opensecurity.in.')
=> Please e-mail me at email@opensecurity.in.

```
strip_links('Blog: <a href="http://www.opsec.opensecurity.in/" class="nav" target=\"_blank\">Visit</a>.')
# => Blog: Visit.
```

Open Source Software Firewalls for XSS Protection

ModSecurity - https://www.modsecurity.org/
IronBee - https://www.ironbee.com/

Free or Open Source tools for Detecting XSS

OWASP Xenotix XSS Exploit Framework
IronWASP
Acunetix Free
arachni
ImmuniWeb Self-Fuzzer Addon for Firefox

Acknowledgements

Mario Heiderich, Ahamed Nafeez

References

https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet
https://www.owasp.org/index.php/DOM_based_XSS_Prevention_Cheat_Sheet
https://www.owasp.org/index.php/List_of_useful_HTTP_headers
https://www.owasp.org/index.php/HttpOnly
https://www.owasp.org/index.php/OWASP_Xenotix_XSS_Exploit_Framework
https://www.owasp.org/index.php/OWASP_Java_Encoder_Project
https://code.google.com/p/owasp-esapi-java/
http://www.w3.org/TR/CSP11/
https://w3c.github.io/webappsec/specs/content-security-policy/
http://www.html5rocks.com/en/tutorials/security/content-security-policy/
https://tools.ietf.org/rfc/rfc7034.txt
http://msdn.microsoft.com/en-us/library/system.web.security.antixss.antixssencoder(v=vs.110).aspx
http://msdn.microsoft.com/en-us/library/system.web.httputility(v=vs.110).aspx
http://openmya.hacker.jp/hasegawa/security/utf7cs.html
http://www.thespanner.co.uk/2013/05/16/dom-clobbering/
http://www.slideshare.net/x00mario/the-innerhtml-apocalypse/46
http://wpl.codeplex.com/
http://opensecurity.in/

http://cure53.de/fp170.pdf
https://www.modsecurity.org/
https://www.ironbee.com/
http://taligarsiel.com/Projects/howbrowserswork1.htm
https://frederik-braun.com/xfo-clickjacking.pdf
http://mootools.net/docs/core/Types/String
http://www.strictly-software.com/htmlencode
http://backbonejs.org/#Model
https://www.ng-book.com/p/Security/
https://docs.angularjs.org/api/ng/service/$sce
http://spinejs.com/docs/views
https://github.com/cure53/DOMPurify
https://github.com/leizongmin/js-xss
http://api.rubyonrails.org/classes/ERB/Util.html
http://api.rubyonrails.org/classes/ActionView/Helpers/SanitizeHelper.html
http://yuilibrary.com/yui/docs/api/classes/Escape.html#method_html
http://prototypejs.org/doc/latest/language/String/prototype/escapeHTML/
http://docs.php.net/manual/en/function.htmlspecialchars.php
http://www.smarty.net/docsv2/en/language.modifier.escape
https://www.superevr.com/blog/2012/exploiting-xss-in-ajax-web-applications/
http://blog.opensecurityresearch.com/2011/12/evading-content-security-policy-with.html
http://www.janoszen.com/2012/04/16/proper-xss-protection-in-javascript-php-and-smarty/
http://wpcme.coverity.com/wp-content/uploads/What_Every_Developer_Should_Know_0213.pdf