

Cross-site Scripting

Cross-site Scripting (XSS/CSS) attacks are a subset of injection attacks, in which malicious scripts are injected into otherwise clean and trusted websites. XSS may be used by attackers to bypass access controls. In contrast to other exploits, Cross-site Scripting may not require social engineering, thus making it much more useful for attackers and threatening for users. XSS is a security vulnerability that results from website managers not properly handling and validating user-input. XSS attacks are categorized into 3 types.

1. Stored XSS or Persistent XSS (Type-1 XSS)
2. Reflected XSS or Non-persistent (Type-2 XSS)
3. DOM based XSS (Type-0 XSS)

Stored XSS attacks: These arise when the injected script is permanently stored on the target server, such as in a database, as a comment.

Reflected XSS attacks: These arise when the injected script is reflected off the server such as in an error message, search result, or any other response that includes the input sent to the server as part of the request. In other words, these occur when the attacker submits a form input which the target server uses to parse and display a page of results. A reflected attack is typically delivered via email or a neutral web site. The bait is an innocent-looking URL, pointing to a trusted site but containing the XSS vector. If the trusted site is vulnerable to the vector, clicking the link can cause the victim's browser to execute the injected script.

DOM based XSS attacks: This is a relatively less-known vulnerability, first identified in 2005 by Amit Klein. DOM Based

XSS is a form of XSS where the entire affected data flow from source to sink takes place in the browser itself, i.e., the source of the data is in the DOM, the sink is also in the DOM, and the data flow doesn't leave the browser. For example, the source could be the URL of the page or it could be an element of the HTML, and the sink is a sensitive method call that causes the execution of the malicious data.

What do we demonstrate?

We demonstrate stored XSS attacks on a demo target called DVWA. Note that similar techniques apply for Reflected and DOM too. DVWA allows us to set its security level to low, medium, or high. Once we are able to inject a simple script, then we could inject almost anything. These levels are set according to standards of a typical random website that one would find on the Internet. We also describe Black-box testing, Gray-box testing and some basic preventive measures needed to avoid XSS,

Stored XSS

Stored Cross-site Scripting (XSS) is the most dangerous type of Cross Site Scripting. Web applications that allow users to store data are potentially exposed to this type of attack. This vulnerability can be used to conduct a number of browser-based attacks including:

- Hijacking another user's browser
- Capturing sensitive information viewed by application users
- Pseudo defacement of the application
- Port scanning of internal hosts ("internal" in relation to the users of the web application)

- Directed delivery of browser-based exploits
- Other malicious activities

Stored XSS does not need a malicious link to be exploited. A successful exploitation occurs when a user visits a page with a stored XSS. The following phases relate to a typical stored XSS attack scenario:

- Attacker stores malicious code into the vulnerable page
- User authenticates in the application
- User visits vulnerable page
- Malicious code is executed by the user's browser

Low security level:

With low security level, there are no restrictions and the input is parsed straight without any validation. Therefore, any script is allowed.



Home
Instructions
Setup / Reset DB

Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA

SQL Injection
SQL Injection (Blind)
Weak Session IDs
XSS (DOM)
XSS (Reflected)
XSS (Stored)

DVWA Security
PHP Info
About

Logout

Username: admin
Security Level: low
PHPIDS: disabled

Vulnerability: Stored Cross Site Scripting (XSS)

Name *

Message *

More Information

- [https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- <http://www.cgisecurity.com/xss-faq.html>
- <http://www.scriptalert1.com/>

This is an example forum.



Home
Instructions
Setup / Reset DB

Brute Force
Command Injection
CSRF
File Inclusion
File Upload
Insecure CAPTCHA
SQL Injection
SQL Injection (Blind)
Weak Session IDs
XSS (DOM)
XSS (Reflected)
XSS (Stored)

DVWA Security
PHP Info
About

Logout

Vulnerability: Stored Cross Site Scripting (XSS)

Name *

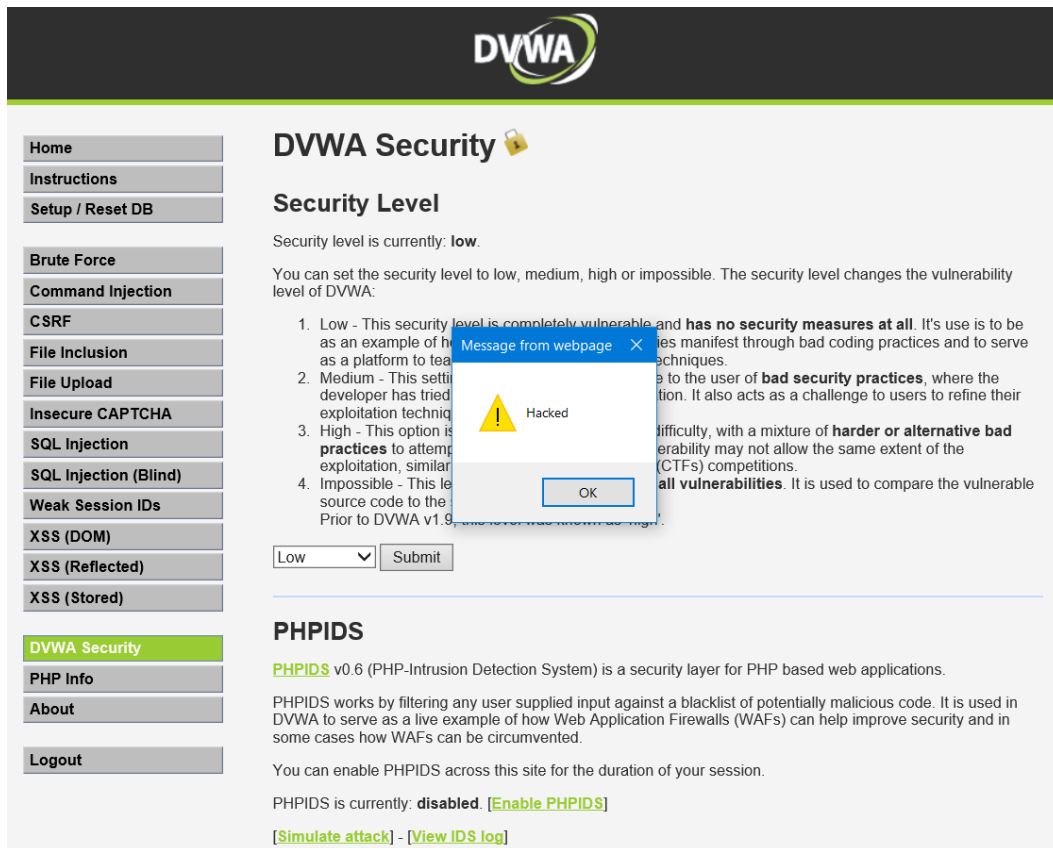
Message *

More Information

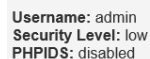
- [https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- <http://www.cgisecurity.com/xss-faq.html>
- <http://www.scriptalert1.com/>

Username: admin
Security Level: low
PHPIDS: disabled

The attacker enters the illustrated test script instead of a plaintext.



This message pops up when anyone tries to access the website.



The actual website doesn't show the script to the user as it's parsed and executed in the browser.

Medium Security:

When there's medium security, improper sanitation and validation of user input is enabled. For e.g. the following PHP script shows the sanitation performed on the server side.

```
<?php
// Sanitize message input
$message = strip_tags( addslashes( $message ) );
$message = ((isset($GLOBALS["__mysqli_ston"]) && is_object($GLOBALS["__mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["__mysqli_ston"], $message) : ((trigger_error("[MySQLConverterToo] Fix the mysql_escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));
$message = htmlspecialchars( $message );

// Sanitize name input
$name = str_replace( '<script>', '', $name );
$name = ((isset($GLOBALS["__mysqli_ston"]) && is_object($GLOBALS["__mysqli_ston"])) ? mysqli_real_escape_string($GLOBALS["__mysqli_ston"], $name) : ((trigger_error("[MySQLConverterToo] Fix the mysql_escape_string() call! This code does not work.", E_USER_ERROR)) ? "" : ""));

//mysql_close();
}

?>
```

As you can see, the message is validated using `strip_tags` and `htmlspecialchars()` functions. Therefore, HTML elements won't be parsed as such and is considered equivalent to a string. Also, the name field is validated by replacing any '`<script>`' tags are erased.

However, we could get through this incomplete validation using something like `<ScRiPt>` instead of `<script>`. Also, one could use `<scr<script>ipt>` instead so that it's converted to `<script>`.



- Home
- Instructions
- Setup / Reset DB
- Brute Force
- Command Injection
- CSRF
- File Inclusion
- File Upload
- Insecure CAPTCHA
- SQL Injection
- SQL Injection (Blind)
- Weak Session IDs
- XSS (DOM)
- XSS (Reflected)
- XSS (Stored)**
- DVWA Security
- PHP Info
- About
- Logout

Vulnerability: Stored Cross Site Scripting (XSS)

Name *

Message *

More Information

- [https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- <http://www.cgisecurity.com/xss-faq.html>
- <http://www.scriptalert1.com/>

Username: admin
Security Level: medium
PHPIDS: disabled



- Home
- Instructions
- Setup / Reset DB
- Brute Force
- Command Injection
- CSRF
- File Inclusion
- File Upload
- Insecure CAPTCHA
- SQL Injection
- SQL Injection (Blind)
- Weak Session IDs
- XSS (DOM)
- XSS (Reflected)
- XSS (Stored)**
- DVWA Security
- PHP Info
- About
- Logout

Vulnerability: Stored Cross Site Scripting (XSS)

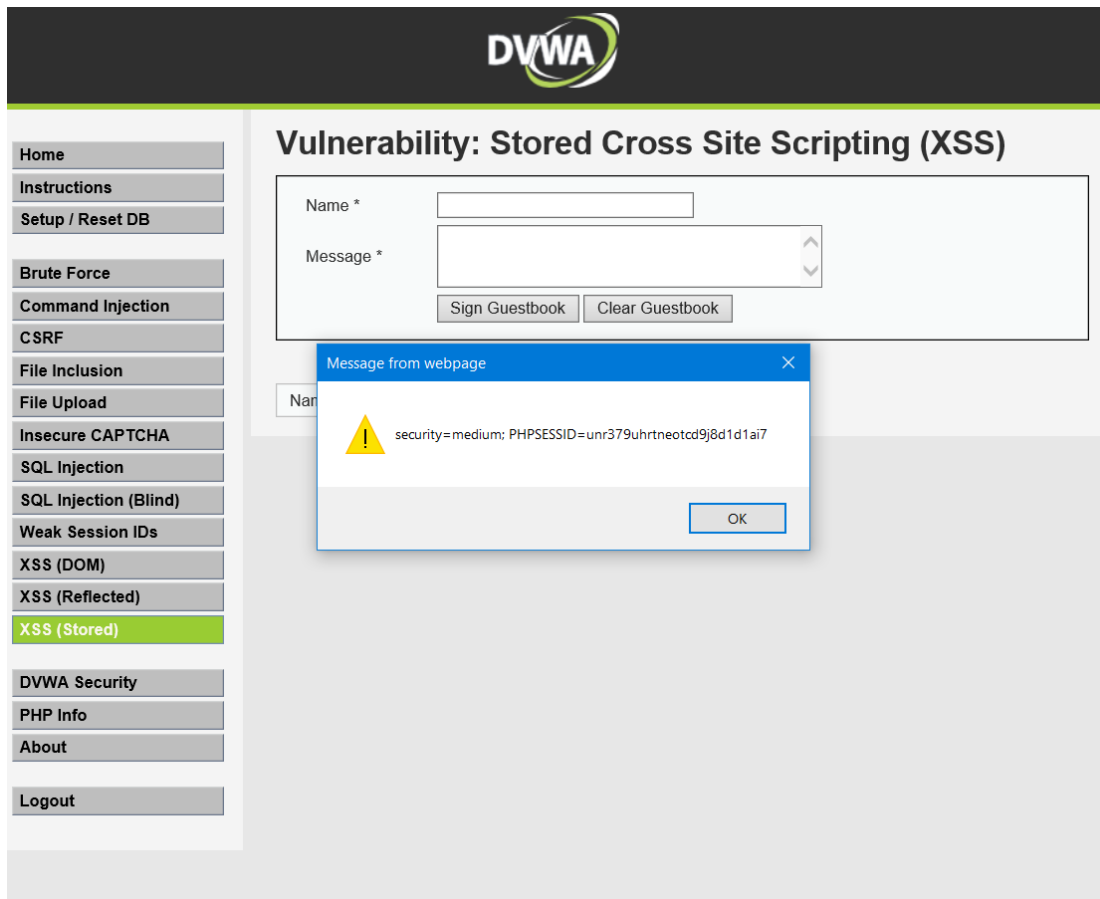
Name *

Message *

Name:

Message from webpage

 hacked



One could also get away with session and cookie info using very simple JavaScript like `alert(document.cookie)`


High Security:

With High level of Security, the following additional condition was imposed in the source code.

```
$name = preg_replace( '/<(.*?)s(.*?)c(.*?)r(.*?)i(.*?)p(.*?)t/i', '', $name );
```

Therefore, our above methods fail to work here. However, we're not out of options. Who told JavaScript could only be injected in a script element? E.g. we could use an error message, like, `<img`

src=x onError=alert('Hacked!')> instead.



Home

Instructions

Setup / Reset DB

Brute Force

Command Injection

CSRF

File Inclusion

File Upload

Insecure CAPTCHA

SQL Injection

SQL Injection (Blind)

Weak Session IDs

XSS (DOM)

XSS (Reflected)

XSS (Stored)

DVWA Security

PHP Info

About

Logout

Vulnerability: Stored Cross Site Scripting (XSS)

Name *

mg src=x onError=alert('Hacked!')>

Message *

Example Message.

Sign Guestbook

Clear Guestbook

More Information

- [https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- <http://www.cgisecurity.com/xss-faq.html>
- <http://www.scriptalert1.com/>

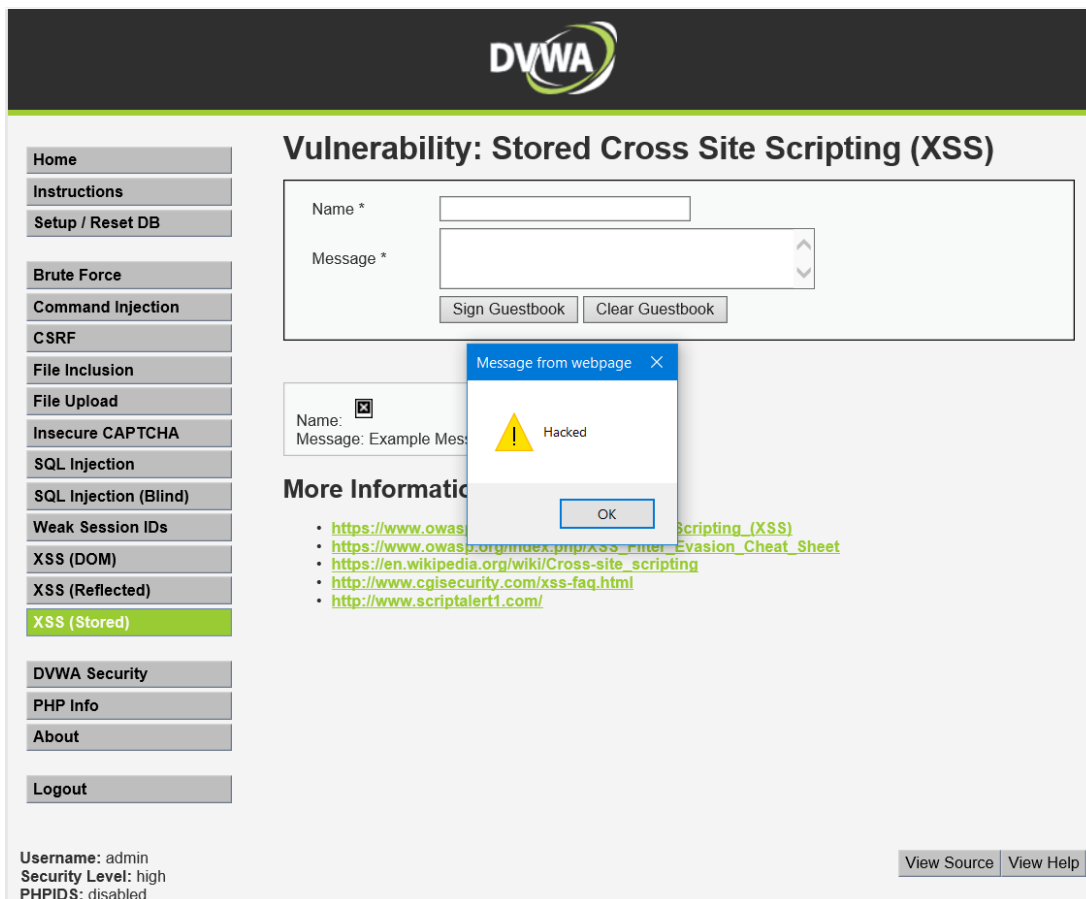
Username: admin

Security Level: high

PHPIDS: disabled

View Source

View Help



Black Box testing (this section can be skipped.)¹

A black-box test will include at least three phases:

1. Detect input vectors. For each web page, the tester must determine all the web application's user-defined variables and how to input them. This includes hidden or non-obvious inputs such as HTTP parameters, POST data, hidden form field values, and predefined radio or selection values. Typically, in-browser HTML editors or web proxies are used to view these hidden variables. See the example below.¹

2. Analyze each input vector to detect potential vulnerabilities. To detect an XSS vulnerability, the tester will typically use specially crafted input data with each input vector. Such input data is typically harmless, but trigger responses from the web browser that manifests the vulnerability. Testing data can be generated by using a web application fuzzer, an automated predefined list of known attack strings, or manually.

Some example of such input data are the following:

```
<script>alert(123)</script>
```

¹ This section has been adapted from the OWASP Testing Guide for Reflected XSS. Please find the complete page [here](#)

```
"><script>alert(document.cookie)</script>
```

For a comprehensive list of potential test strings, see the [XSS Filter Evasion Cheat Sheet](#).

3. For each test input attempted in the previous phase, the tester will analyze the result and determine if it represents a vulnerability that has a realistic impact on the web application's security. This requires examining the resulting web page HTML and searching for the test input. Once found, the tester identifies any special characters that were not properly encoded, replaced, or filtered out. The set of vulnerable unfiltered special characters will depend on the context of that section of HTML.

Ideally all HTML special characters will be replaced with HTML entities. The key HTML entities to identify are:

```
> (greater than)
< (less than)
& (ampersand)
' (apostrophe or single quote)
" (double quote)
```

Within the context of an HTML action or JavaScript code, a different set of special characters will need to be escaped, encoded, replaced, or filtered out. These characters include:

```
\n (new line)
\r (carriage return)
\' (apostrophe or single quote)
\" (double quote)
\\ (backslash)
\uXXXX (unicode values)
```

For a more complete reference, [see](#) the Mozilla JavaScript guide.

Gray Box testing

Gray Box testing is similar to Black box testing. In gray box testing, the pen-tester has partial knowledge of the application. In this case, information regarding user input, input validation controls, and how the user input is rendered back to the user might be known by the pen-tester.

If source code is available (White Box), all variables received from users should be analyzed. Moreover the tester should analyze any sanitization procedures implemented to decide if these can be circumvented.

Some automated tools, references and resources.

- **OWASP CAL9000**

CAL9000 is a collection of web application security testing tools that complement the feature set of current web proxies and automated scanners

- **XSS-Proxy**

XSS-Proxy is an advanced Cross-Site-Scripting (XSS) attack tool.

- **ratproxy**

A semi-automated, largely passive web application security audit tool, optimized for an accurate and sensitive detection, and automatic annotation, of potential problems and security-relevant design patterns based on the observation of existing, user-initiated traffic in complex web 2.0 environments.

- **Burp Proxy**

Burp Proxy is an interactive HTTP/S proxy server for attacking and testing web applications

- **OWASP Zed Attack Proxy (ZAP)**

ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. It is designed to be used by people with a wide range of security experience and as such is ideal for developers and functional testers who are new to penetration testing. ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually.

- **OWASP Xenotix XSS Exploit Framework**

OWASP Xenotix XSS Exploit Framework is an advanced Cross Site Scripting (XSS) vulnerability detection and exploitation framework. It provides Zero False Positive scan results with its unique Triple Browser Engine (Trident, WebKit, and Gecko) embedded scanner. It is claimed to have the world's 2nd largest XSS Payloads of about 1600+ distinctive XSS Payloads for effective XSS vulnerability detection and WAF Bypass. Xenotix Scripting Engine allows you to create custom test cases and addons over the Xenotix API. It is incorporated with a feature rich Information Gathering module for target Reconnaissance. The Exploit Framework includes offensive XSS exploitation modules for Penetration Testing and Proof of Concept creation.

Bibilography

- Joel Scambray, Mike Shema, Caleb Sima - "Hacking Exposed Web Applications", Second Edition, McGraw-Hill, 2006 - [ISBN 0-07-226229-0](#)
- Dafydd Stuttard, Marcus Pinto - "The Web Application's Handbook - Discovering and Exploiting Security Flaws", 2008, Wiley, [ISBN 978-0-470-17077-9](#)
- Jeremiah Grossman, Robert "RSnake" Hansen, Petko "pdp" D. Petkov, Anton Rager, Seth Fogie - "Cross Site Scripting Attacks: XSS Exploits and Defense", 2007, Syngress, ISBN-10: 1-59749-154-3

WhitePapers

- **CERT** - Malicious HTML Tags Embedded in Client Web Requests: [Read](#)
- **Rsnake** - XSS Cheat Sheet: [Read](#)
- **cgisecurity.com** - The Cross Site Scripting FAQ: [Read](#)
- **G.Ollmann** - HTML Code Injection and Cross-site scripting: [Read](#)
- **A. Calvo, D.Tiscornia** - alert('A javascrip agent'): [Read](#)
- **S. Frei, T. Dübendorfer, G. Ollmann, M. May** - Understanding the Web browser threat: [Read](#)

How to avoid XSS? (The more important question)²

The following rules are intended to prevent all XSS in your application. While these rules do not allow absolute freedom in putting untrusted data into an HTML document, they should cover the vast majority of common use cases. You do not have to allow all the rules in your organization. Many organizations may find that allowing only Rule #1 and Rule #2 are sufficient for their needs.

Do NOT simply escape the list of example characters provided in the various rules. It is NOT sufficient to escape only that list. Blacklist approaches are quite fragile. The whitelist rules here have been carefully designed to provide protection even against future vulnerabilities introduced by browser changes.

RULE #0 - Never Insert Untrusted Data Except in Allowed Locations

The first rule is to **deny all** - don't put untrusted data into your HTML document unless it is within one of the slots defined in Rule #1 through Rule #5. The reason for Rule #0 is that there are so many strange contexts within HTML that the list of escaping rules gets very complicated. We can't think of any good reason to put untrusted data in these contexts. This includes "nested contexts" like a URL inside a javascript -- the encoding rules for those locations are tricky and dangerous. If you insist on putting untrusted data into nested contexts, please do a lot of cross-browser testing and let us know what you find out.

<code><script>...NEVER PUT UNTRUSTED DATA HERE...</script></code>	directly in a script
<code><!--...NEVER PUT UNTRUSTED DATA HERE...--></code>	inside an HTML comment
<code><div ...NEVER PUT UNTRUSTED DATA HERE...=test /></code>	in an attribute name
<code><NEVER PUT UNTRUSTED DATA HERE... href="/test" /></code>	in a tag name
<code><style>...NEVER PUT UNTRUSTED DATA HERE...</style></code>	directly in CSS

Most importantly, never accept actual JavaScript code from an untrusted source and then run it. For example, a parameter named "callback" that contains a JavaScript code snippet. No amount of escaping can fix that.

² This section has been adapted from OWASP XSS prevention cheat sheet, please find the complete reference [here](#).

RULE #1 - HTML Escape Before Inserting Untrusted Data into HTML Element Content

Rule #1 is for when you want to put untrusted data directly into the HTML body somewhere. This includes inside normal tags like `div`, `p`, `b`, `td`, etc. Most web frameworks have a method for HTML escaping for the characters detailed below. However, this is not sufficient for other HTML contexts. You need to implement the other rules detailed here as well.

```
<body>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</body>
```

```
<div>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</div>
```

```
any other normal HTML elements
```

Escape the following characters with HTML entity encoding to prevent switching into any execution context, such as `script`, `style`, or event handlers. Using hex entities is recommended in the spec. In addition to the 5 characters significant in XML (`&`, `<`, `>`, `"`, `'`), the forward slash is included as it helps to end an HTML entity.

```
& --> &amp;
< --> &lt;
> --> &gt;
" --> &quot;
' --> &#x27;      &apos; not recommended because it's not in the HTML
spec &apos; is in the XML and XHTML specs.
/ --> &#x2F;      forward slash is included as it helps end an HTML
entity
```

RULE #2 - Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes

Rule #2 is for putting untrusted data into typical attribute values like `width`, `name`, `value`, etc. This should not be used for complex attributes like `href`, `src`, `style`, or any of the event handlers like `onmouseover`. It is extremely important that event handler attributes should follow Rule #3 for HTML JavaScript Data Values. 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. The reason this rule is so broad is that developers frequently leave attributes unquoted. Properly quoted attributes can only be escaped with the corresponding quote. Unquoted attributes can be broken out of with many characters, including `[space]` `%` `*` `+` `,` `-` `/` `;` `<` `=` `>` `^` and `|`.

RULE #3-JavaScript escape before inserting untrusted data into JavaScript data values

Rule #3 concerns dynamically generated JavaScript code - both script blocks and event-handler attributes. The only safe place to put untrusted data into this code is inside a quoted "data value." Including untrusted data inside any other JavaScript context is quite dangerous, as it is extremely easy to switch into an execution context with characters including (but not limited to) semi-colon, equals, space, plus, and many more, so use with caution.

Please note there are some JavaScript functions that can never safely use untrusted data as input - **EVEN IF JAVASCRIPT ESCAPED!**

For example:

```
<script>
  window.setInterval('...EVEN IF YOU ESCAPE UNTRUSTED DATA YOU ARE XSSED HERE...');
</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 is properly quoted, breaking out requires the corresponding quote. However, we have intentionally made this rule quite broad because event handler attributes are often left unquoted. 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 because the HTML parser runs before the JavaScript parser.

RULE #4 - CSS Escape and strictly validate before inserting untrusted data into HTML style property values

Rule #4 is for when you want to put untrusted data into a stylesheet or a style tag. CSS is surprisingly powerful, and can be used for numerous attacks. Therefore, it's important that you only use untrusted data in a property **value** and not into other places in style data. You should stay away from putting untrusted data into complex properties like url, behavior, and custom (-moz-binding). You should also not put untrusted data into IE's expression property value which allows JavaScript.

Please note there are some CSS contexts that can never safely use untrusted data as input - **EVEN IF PROPERLY CSS ESCAPED!** You will have to ensure that URLs only start with "http" not "javascript" and that properties never start with "expression".

For example:

```
{ background-url : "javascript:alert(1)"; } // and all other URLs
{ text-size: "expression(alert('XSS'))"; } // only in IE
```

Again, the same escape rules follow as in Rule #3.

RULE #5 - URL Escape Before Inserting Untrusted Data into HTML URL Parameter Values

Rule #5 is for when you want to put untrusted data into HTTP GET parameter value.

```
<a href="http://www.somesite.com?test=...ESCAPE UNTRUSTED DATA BEFORE
PUTTING HERE...">link</a >
```

WARNING: Do not encode complete or relative URL's with URL encoding! If untrusted input is meant to be placed into href, src or other URL-based attributes, it should be validated to make sure it does not point to an unexpected protocol, especially Javascript links. URL's should then be encoded based on the context of display like any other piece of data. For example, user driven URL's in HREF links should be attribute encoded. For example:

```
String userURL = request.getParameter( "userURL" )
boolean isValidURL = Validator.IsValidURL(userURL, 255);
if (isValidURL) {
    <a href="<%=encoder.encodeForHTMLAttribute(userURL)%>">link</a>
}
```

RULE #6 - Sanitize HTML Markup with a Library Designed for the Job

If your application handles markup -- untrusted input that is supposed to contain HTML -- it can be very difficult to validate. Encoding is also difficult, since it would break all the tags that are supposed to be in the input. Therefore, you need a library that can parse and clean HTML formatted text. There are several available at OWASP that are simple to use. The HTML is cleaned with a white list approach. All allowed tags and attributes can be configured. For more info and the appropriate tools, click [here](#).

These are some of the precautions to be taken to avoid XSS. For more useful tips, find the complete list [here](#). We'll summarize some things for you.

Data Type	Context	Code Sample	Defense
String	HTML Body	<code>UNTRUSTED DATA</code>	<ul style="list-style-type: none"> HTML Entity Encoding
String	Safe HTML Attributes	<code><input type="text" name="fname" value="UNTRUSTED DATA"></code>	<ul style="list-style-type: none"> Aggressive HTML Entity Encoding Only place untrusted data into a whitelist of safe attributes (listed below). Strictly validate unsafe attributes such as background, id and name.
String	GET Parameter	<code>clickme</code>	<ul style="list-style-type: none"> URL Encoding
String	Untrusted URL in a SRC or HREF attribute	<code>clickme <iframe src="UNTRUSTED URL" /></code>	<ul style="list-style-type: none"> Canonicalize input URL Validation Safe URL verification Whitelist http and https URL's only (Avoid the JavaScript Protocol to Open a new Window) Attribute encoder
String	CSS Value	<code><div style="width: UNTRUSTED DATA;">Selection</div></code>	<ul style="list-style-type: none"> Strict structural validation CSS Hex encoding Good design of CSS Features
String	JavaScript Variable	<code><script>var currentValue='UNTRUSTED DATA';</script> <script>someFunction('UNTRUSTED DATA');</script></code>	<ul style="list-style-type: none"> Ensure JavaScript variables are quoted JavaScript Hex Encoding JavaScript Unicode Encoding Avoid backslash encoding (\" or \' or \\)
HTML	HTML Body	<code><div>UNTRUSTED HTML</div></code>	<ul style="list-style-type: none"> HTML Validation (JSoup, AntiSamy, HTML Sanitizer)
String	DOM XSS	<code><script>document.write("UNTRUSTED INPUT: " + document.location.hash);</script></code>	<ul style="list-style-type: none"> DOM based XSS Prevention Cheat Sheet

Output Encoding Rules Summary

The purpose of output encoding (as it relates to Cross Site Scripting) is to convert untrusted input into a safe form where the input is displayed as data to the user without executing as code in the browser. The following charts details a list of critical output encoding methods needed to stop Cross Site Scripting.

Encoding Type	Encoding Mechanism
HTML Entity Encoding	Convert & to & Convert < to < Convert > to > Convert " to " Convert ' to ' Convert / to /
HTML Attribute Encoding	Except for alphanumeric characters, escape all characters with the HTML Entity &#xHH; format, including spaces. (HH = Hex Value)
URL Encoding	Standard percent encoding, see: http://www.w3schools.com/tags/ref_urlencode.asp . URL encoding should only be used to encode parameter values, not the entire URL or path fragments of a URL.
JavaScript Encoding	Except for alphanumeric characters, escape all characters with the \uXXXX unicode escaping format (X = Integer).
CSS Hex Encoding	CSS escaping supports \XX and \XXXXXX. Using a two character escape can cause problems if the next character continues the escape sequence. There are two solutions (a) Add a space after the CSS escape (will be ignored by the CSS parser) (b) use the full amount of CSS escaping possible by zero padding the value.