XSS (Cross Site Scripting) Prevention Cheat Sheet

Last revision (mm/dd/yy): **05/9/2018**

Introduction

[hide]

1 Introduction
- 1.1 A Positive XSS Prevention Model
- 1.2 Why Can't I Just HTML Entity Encode Untrusted Data?
- 1.3 You Need a Security Encoding Library

2 XSS Prevention Rules
- 2.1 RULE #0 - Never Insert Untrusted Data Except in Allowed Locations
- 2.2 RULE #1 - HTML Escape Before Inserting Untrusted Data into HTML Element Content
- 2.3 RULE #2 - Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes
- 2.4 RULE #3 - JavaScript Escape Before Inserting Untrusted Data into JavaScript Data Values
  - 2.4.1 RULE #3.1 - HTML escape JSON values in an HTML context and read the data with JSON.parse
    - 2.4.1.1 JSON serialization
    - 2.4.1.2 HTML entity encoding
- 2.5 RULE #4 - CSS Escape And Strictly Validate Before Inserting Untrusted Data into HTML Style Property Values
- 2.6 RULE #5 - URL Escape Before Inserting Untrusted Data into HTML URL Parameter Values
- 2.7 RULE #6 - Sanitize HTML Markup with a Library Designed for the Job
- 2.8 RULE #7 - Prevent DOM-based XSS
- 2.9 Bonus Rule #1: Use HTTPOnly cookie flag
- 2.10 Bonus Rule #2: Implement Content Security Policy
- 2.11 Bonus Rule #3: Use an Auto-Escaping Template System
- 2.12 Bonus Rule #4: Use the X-XSS-Protection Response Header

3 XSS Prevention Rules Summary

4 Output Encoding Rules Summary

5 Related Articles

6 Other Cheatsheets

7 Authors and Primary Editors

This article provides a simple positive model for preventing XSS using output escaping/encoding properly. While there are a huge number of XSS attack vectors, following a few simple rules can completely defend against this serious attack. This article does not explore the technical or business impact of XSS. Suffice it to say that it can lead to an attacker gaining the ability to do anything a victim can do through their browser.

Both reflected and stored XSS can be addressed by performing the appropriate validation and escaping on the server-side. DOM Based XSS can be addressed with a special subset of rules described in the DOM based XSS Prevention Cheat Sheet.

For a cheatsheet on the attack vectors related to XSS, please refer to the XSS Filter Evasion Cheat

Home
About OWASP
Acknowledgements
Advertising
AppSec Events
Books
Brand Resources
Chapters
Donate to OWASP
Downloads
Funding
Governance
Initiatives
Mailing Lists
Membership
Merchandise
News
Community portal
Presentations
Press
Projects
Video
Volunteer

Reference
Activities
Attacks
Code Snippets
Controls
Glossary
How To...
Java Project
.NET Project
Principles
Technologies
Threat Agents
Vulnerabilities

Tools
What links here
Related changes
Special pages
Printable version
Permanent link
Page information

Sheet. More background on browser security and the various browsers can be found in the Browser Security Handbook.

Before reading this cheatsheet, it is important to have a fundamental understanding of Injection Theory.

A Positive XSS Prevention Model

This article treats an HTML page like a template, with slots where a developer is allowed to put untrusted data. These slots cover the vast majority of the common places where a developer might want to put untrusted data. Putting untrusted data in other places in the HTML is not allowed. This is a "whitelist" model, that denies everything that is not specifically allowed.

Given the way browsers parse HTML, each of the different types of slots has slightly different security rules. When you put untrusted data into these slots, you need to take certain steps to make sure that the data does not break out of that slot into a context that allows code execution. In a way, this approach treats an HTML document like a parameterized database query - the data is kept in specific places and is isolated from code contexts with escaping.

This document sets out the most common types of slots and the rules for putting untrusted data into them safely. Based on the various specifications, known XSS vectors, and a great deal of manual testing with all the popular browsers, we have determined that the rules proposed here are safe.

The slots are defined and a few examples of each are provided. Developers SHOULD NOT put data into any other slots without a very careful analysis to ensure that what they are doing is safe. Browser parsing is extremely tricky and many innocuous looking characters can be significant in the right context.

Why Can't I Just HTML Entity Encode Untrusted Data?

HTML entity encoding is okay for untrusted data that you put in the body of the HTML document, such as inside a tag. It even sort of works for untrusted data that goes into attributes, particularly if you're religious about using quotes around your attributes. But HTML entity encoding doesn't work if you're putting untrusted data inside a <script> tag anywhere, or an event handler attribute like onmouseover, or inside CSS, or in a URL. So even if you use an HTML entity encoding method everywhere, you are still most likely vulnerable to XSS. **You MUST use the escape syntax for the part of the HTML document you're putting untrusted data into.** That's what the rules below are all about.

You Need a Security Encoding Library

Writing these encoders is not tremendously difficult, but there are quite a few hidden pitfalls. For example, you might be tempted to use some of the escaping shortcuts like \" in JavaScript. However, these values are dangerous and may be misinterpreted by the nested parsers in the browser. You might also forget to escape the escape character, which attackers can use to neutralize your attempts to be safe. OWASP recommends using a security-focused encoding library to make sure these rules are properly implemented.

Microsoft provides an encoding library named the Microsoft Anti-Cross Site Scripting Library for the .NET platform and ASP.NET Framework has built-in ValidateRequest function that provides **limited** sanitization.

The OWASP Java Encoder Project provides a high-performance encoding library for Java.

XSS Prevention Rules

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**. Please add a note to the discussion page if there is an additional context that is often required and can be secured with escaping.

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.

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

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 **absolutely 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 its not in the HTML spec (See: section 24.4.1) &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.

```
<div attr=...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...>content</div>
inside UNquoted attribute <div attr='...ESCAPE UNTRUSTED DATA BEFORE
PUTTING HERE...'>content</div> inside single quoted attribute <div
attr="...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...">content</div> inside
double quoted attribute
```

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.

```
<script>alert('...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...')</script>
inside a quoted string <script>x='...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE...'</script> one side of a quoted expression <div
onmouseover="x='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'"</div>
inside quoted event handler
```

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 #3.1 - HTML escape JSON values in an HTML context and read the data with JSON.parse

In a Web 2.0 world, the need for having data dynamically generated by an application in a javascript context is common. One strategy is to make an AJAX call to get the values, but this isn't always performant. Often, an initial block of JSON is loaded into the page to act as a single place to store multiple values. This data is tricky, though not impossible, to escape correctly without breaking the format and content of the values.

Ensure returned *Content-Type* header is application/json and not text/html. This shall instruct the browser not misunderstand the context and execute injected script

Bad HTTP response:

```
HTTP/1.1 200 Date: Wed, 06 Feb 2013 10:28:54 GMT Server: Microsoft-
IIS/7.5.... Content-Type: text/html; charset=utf-8 <-- bad .... Content-
Length: 373 Keep-Alive: timeout=5, max=100 Connection: Keep-Alive
{"Message":"No HTTP resource was found that matches the request URI
'dev.net.ie/api/pay/.html?HouseNumber=9&AddressLine
=The+Gardens<script>alert(1)
</script>&AddressLine2=foxlodge+woods&TownName=Meath'.","MessageDetail":"No
type was found that matches the controller named 'pay'."} <-- this script
will pop!!
```

Good HTTP response

```
HTTP/1.1 200 Date: Wed, 06 Feb 2013 10:28:54 GMT Server: Microsoft-
IIS/7.5.... Content-Type: application/json; charset=utf-8 <--good .....
.....
```

A common **anti-pattern** one would see:

```
<script> var initData = <%= data.to_json %>; // Do NOT do this without
encoding the data with one of the techniques listed below. </script>
```

JSON serialization

A safe JSON serializer will allow developers to serialize JSON as string of literal JavaScript which can be embedded in an HTML in the contents of the <script> tag. HTML characters and JavaScript line terminators need be escaped. Consider the Yahoo JavaScript Serializer for this task. https://github.com/yahoo/serialize-javascript

HTML entity encoding

This technique has the advantage that html entity escaping is widely supported and helps separate data from server side code without crossing any context boundaries. Consider placing the JSON block on the page as a normal element and then parsing the innerHTML to get the contents. The javascript that reads the span can live in an external file, thus making the implementation of CSP enforcement easier.

```
<div id="init_data" style="display: none"> <%= html_escape(data.to_json) %>
</div>
```

```
// external js file var dataElement = document.getElementById('init_data');
// decode and parse the content of the div var initData =
JSON.parse(dataElement.textContent);
```

An alternative to escaping and unescaping JSON directly in JavaScript, is to normalize JSON server-side by converting '<' to '\u003c' before delivering it to the browser.

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.

```
<style>selector { property : ...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE...; } </style> property value
<style>selector { property : "...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE..."; } </style> property value
<span style="property : ...ESCAPE UNTRUSTED DATA BEFORE PUTTING
HERE...">text</span> property value
```

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
```

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. All attributes should be quoted but your encoding should be strong enough to prevent XSS when untrusted data is placed in unquoted contexts. 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 because the HTML parser runs before the JavaScript parser. Please note that we recommend aggressive CSS encoding and validation to prevent XSS attacks for both quoted and unquoted attributes.

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 >
```

Except for alphanumeric characters, escape all characters with ASCII values less than 256 with the %HH escaping format. Including untrusted data in data: URLs should not be allowed as there is no good way to disable attacks with escaping to prevent switching out of the URL. All attributes should be quoted. Unquoted attributes can be broken out of with many characters including [space] % * + , - / ; < = > ^ and |. Note that entity encoding is useless in this context.

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:

HtmlSanitizer - https://github.com/mganss/HtmlSanitizer

An open-source .Net library. The HTML is cleaned with a white list approach. All allowed tags and attributes can be configured. The library is unit tested with the OWASP XSS Filter Evasion Cheat Sheet

```
var sanitizer = new HtmlSanitizer();
sanitizer.AllowedAttributes.Add("class"); var sanitized =
sanitizer.Sanitize(html);
```

OWASP Java HTML Sanitizer - OWASP Java HTML Sanitizer Project

```
import org.owasp.html.Sanitizers; import org.owasp.html.PolicyFactory;
PolicyFactory sanitizer = Sanitizers.FORMATTING.and(Sanitizers.BLOCKS);
String cleanResults = sanitizer.sanitize("<p>Hello, <b>World!</b>");
```

For more information on OWASP Java HTML Sanitizer policy construction, see https://github.com/OWASP/java-html-sanitizer

Ruby on Rails SanitizeHelper - http://api.rubyonrails.org/classes/ActionView/Helpers/SanitizeHelper.html

The SanitizeHelper module provides a set of methods for scrubbing text of undesired HTML elements.

```
<%= sanitize @comment.body, tags: %w(strong em a), attributes: %w(href) %>
```

Other libraries that provide HTML Sanitization include:

PHP HTML Purifier - http://htmlpurifier.org/
JavaScript/Node.js Bleach - https://github.com/ecto/bleach
Python Bleach - https://pypi.python.org/pypi/bleach

RULE #7 - Prevent DOM-based XSS

For details on what DOM-based XSS is, and defenses against this type of XSS flaw, please see the OWASP article on DOM based XSS Prevention Cheat Sheet.

Bonus Rule #1: Use HTTPOnly cookie flag

Preventing all XSS flaws in an application is hard, as you can see. To help mitigate the impact of an XSS flaw on your site, OWASP also recommends you set the HTTPOnly flag on your session cookie and any custom cookies you have that are not accessed by any Javascript you wrote. This cookie flag is typically on by default in .NET apps, but in other languages you have to set it manually. For more details on the HTTPOnly cookie flag, including what it does, and how to use it, see the OWASP article on HTTPOnly.

Bonus Rule #2: Implement Content Security Policy

There is another good complex solution to mitigate the impact of an XSS flaw called Content Security Policy. It's a browser side mechanism which allows you to create source whitelists for client side resources of your web application, e.g. JavaScript, CSS, images, etc. CSP via special HTTP header instructs the browser to only execute or render resources from those sources. For example this CSP

```
Content-Security-Policy: default-src: 'self'; script-src: 'self'
static.domain.tld
```

will instruct web browser to load all resources only from the page's origin and JavaScript source code files additionaly from static.domain.tld. For more details on Content Security Policy, including what it does, and how to use it, see the OWASP article on Content_Security_Policy

Bonus Rule #3: Use an Auto-Escaping Template System

Many web application frameworks provide automatic contextual escaping functionality such as AngularJS strict contextual escaping and Go Templates. Use these technologies when you can.

Bonus Rule #4: Use the X-XSS-Protection Response Header

This HTTP response header enables the Cross-site scripting (XSS) filter built into some modern web browsers. This header is usually enabled by default anyway, so the role of this header is to re-enable the filter for this particular website if it was disabled by the user.

XSS Prevention Rules Summary

The following snippets of HTML demonstrate how to safely render untrusted data in a variety of different contexts.

Data Type	Context	Code Sample	Defense
String	HTML Body	UNTRUSTED DATA	. HTML Entity Encoding
String	Safe HTML Attributes	<input type="text" name="fname" value="UNTRUSTED DATA">	. 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	clickme	. URL Encoding
			. Canonicalize input . URL Validation

String	Untrusted URL in a SRC or HREF attribute	clickme <iframe src="UNTRUSTED URL" />	• Safe URL verification • Whitelist http and https URL's only (Avoid the JavaScript Protocol to Open a new Window) • Attribute encoder
String	CSS Value	Selection	• Strict structural validation • CSS Hex encoding • Good design of CSS Features
String	JavaScript Variable	<script>var currentValue='UNTRUSTED DATA';</script> <script>someFunction('UNTRUSTED DATA');</script>	• Ensure JavaScript variables are quoted • JavaScript Hex Encoding • JavaScript Unicode Encoding • Avoid backslash encoding (\" or \' or \\)
HTML	HTML Body	UNTRUSTED HTML	• HTML Validation (JSoup, AntiSamy, HTML Sanitizer)
String	DOM XSS	<script>document.write("UNTRUSTED INPUT: " + document.location.hash); <script/>	• DOM based XSS Prevention Cheat Sheet

Safe HTML Attributes include: align, alink, alt, bgcolor, border, cellpadding, cellspacing, class, color, cols, colspan, coords, dir, face, height, hspace, ismap, lang, marginheight, marginwidth, multiple, nohref, noresize, noshade, nowrap, ref, rel, rev, rows, rowspan, scrolling, shape, span, summary, tabindex, title, usemap, valign, value, vlink, vspace, width

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.

Related Articles

XSS Attack Cheat Sheet

The following article describes how to exploit different kinds of XSS Vulnerabilities that this article was created to help you avoid:

- OWASP: XSS Filter Evasion Cheat Sheet - Based on - RSnake's: "XSS Cheat Sheet"

A Systematic Analysis of XSS Sanitization in Web Application Frameworks

http://www.cs.berkeley.edu/~prateeks/papers/empirical-webfwks.pdf

Description of XSS Vulnerabilities

- OWASP article on XSS Vulnerabilities

Discussion on the Types of XSS Vulnerabilities

- Types of Cross-Site Scripting

How to Review Code for Cross-site scripting Vulnerabilities

- OWASP Code Review Guide article on Reviewing Code for Cross-site scripting Vulnerabilities

How to Test for Cross-site scripting Vulnerabilities

- OWASP Testing Guide article on Testing for Cross site scripting Vulnerabilities
- XSS Experimental Minimal Encoding Rules

Other Cheatsheets

V - T - E	Cheat Sheets	[Collapse]
Developer / Builder	3rd Party Javascript Management · Access Control · AJAX Security Cheat Sheet · Authentication (ES) · Bean Validation Cheat Sheet · Choosing and Using Security Questions · Clickjacking Defense · Credential Stuffing Prevention Cheat Sheet · Cross-Site Request Forgery (CSRF) Prevention · Cryptographic Storage · C-Based Toolchain Hardening · Deserialization · DOM based XSS Prevention · Forgot Password · HTML5 Security · HTTP Strict Transport Security · Injection Prevention Cheat Sheet · Injection Prevention Cheat Sheet in Java · JSON Web Token (JWT) Cheat Sheet for Java · Input Validation · Insecure Direct Object Reference Prevention · JAAS · Key Management · LDAP Injection Prevention · Logging · Mass Assignment Cheat Sheet · .NET Security · OS Command Injection Defense Cheat Sheet · OWASP Top Ten · Password Storage · Pinning · Query Parameterization · REST Security · Ruby on Rails · Session Management · SAML Security · SQL Injection Prevention · Transaction Authorization · Transport Layer Protection · Unvalidated Redirects and Forwards · User Privacy Protection · Web Service Security · **XSS (Cross Site Scripting) Prevention** · XML External Entity (XXE) Prevention Cheat Sheet
Assessment / Breaker	Attack Surface Analysis · REST Assessment · Web Application Security Testing · XML Security Cheat Sheet · XSS Filter Evasion
Mobile	Android Testing · IOS Developer · Mobile Jailbreaking
OpSec / Defender	Virtual Patching · Vulnerability Disclosure
Draft and Beta	Application Security Architecture · Business Logic Security · Content Security Policy · Denial of Service Cheat Sheet · Grails Secure Code Review · IOS Application Security Testing · PHP Security · Regular Expression Security Cheatsheet · Secure Coding · Secure SDLC · Threat Modeling
	All Pages In This Category

Authors and Primary Editors

Jeff Williams - jeff.williams[at]contrastsecurity.com
Jim Manico - jim[at]owasp.org
Neil Mattatall - neil[at]owasp.org

Categories: Cheatsheets | Popular