XSS (Cross Site Scripting) Prevention Cheat Sheet

From OWASP

Contents

Introduction

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.

These rules apply to all the different varieties of XSS. Both reflected and stored XSS can be addressed by performing the appropriate escaping on the server-side. The use of an escaping/encoding library like the one in ESAPI is strongly recommended as there are many special cases. DOM Based XSS can be addressed by applying these rules on the client on untrusted data.

For a great cheatsheet on the attack vectors related to XSS, please refer to the excellent XSS Cheat Sheet (http://ha.ckers.org/xss.html) by RSnake. More background on browser security and the various browsers can be found in the Browser Security Handbook (http://code.google.com/p/browsersec/) .

Untrusted Data

Untrusted data is most often data that comes from the HTTP request, in the form of URL parameters, form fields, headers, or cookies. But data that comes from databases, web services, and other sources is frequently untrusted from a security perspective. That is, it might not have been perfectly validated. The OWASP Code Review Guide has a decent list of methods that return untrusted data in various languages, but you should be careful about your own methods as well.

Untrusted data should always be treated as though it contains an attack. That means you should not send it **anywhere** without taking steps to make sure that any attacks are detected and neutralized. As applications get more and more interconnected, the likelihood of a buried attack being decoded or executed by a downstream interpreter increases rapidly.

Traditionally, input validation has been the preferred approach for handling untrusted data. However, input validation is not a great solution for injection attacks. First, input validation is typically done when the data is received, before the destination is known. That means that we don't know which characters might be significant in the target interpreter. Second, and possibly even more importantly, applications must allow potentially harmful characters in. For example, should poor Mr. O'Malley be prevented from registering in the database simply because SQL considers ' a special character?

While input validation is important and should always be performed, it is not a complete solution for injection attacks. It's better to think of input validation as defense in depth and use **escaping** as described below as the primary defense.

Escaping (aka Output Encoding)

"Escaping (http://www.w3.org/TR/charmod/#sec-Escaping) " is a technique used to ensure that characters are treated as data, not as characters that are relevant to the interpreter's parser. There are lots of different types of escaping, sometimes confusingly

called output "encoding." Some of these techniques define a special "escape" character, and other techniques have a more sophisticated syntax that involves several characters.

Do not confuse output escaping with the notion of Unicode character encoding (http://www.w3.org/TR/charmod/#sec-Digital) , which involves mapping a Unicode character to a sequence of bits. This level of encoding is automatically decoded, and does **not** defuse attacks. However, if there are misunderstandings about the intended charset between the server and browser, it may cause unintended characters to be communicated, possibly enabling XSS attacks. This is why it is still important to specify (http://www.w3.org/TR/charmod/#sec-Encodings) the Unicode character encoding (charset), such as UTF-8, for all communications.

Escaping is the primary means to make sure that untrusted data can't be used to convey an injection attack. There is **no harm** in escaping data properly - it will still render in the browser properly. Escaping simply lets the interpreter know that the data is not intended to be executed, and therefore prevents attacks from working.

Injection Theory

Injection is an attack that involves breaking out of a data context and switching into a code context through the use of special characters that are significant in the interpreter being used. A data context is like data context . If the attacker's data gets placed into the data context, they might break out like this data < script>alert("attack")</script> context .

XSS is a form of injection where the interpreter is the browser and attacks are buried in an HTML document. HTML is easily the worst mashup of code and data of all time, as there are so many possible places to put code and so many different valid encodings. HTML is particularly difficult because it is not only hierarchical, but also contains many different parsers (XML, HTML, JavaScript, VBScript, CSS, URL, etc...).

To really understand what's going on with XSS, you have to consider injection into the hierarchical structure of the HTML DOM (http://www.w3schools.com/HTMLDOM /default.asp) . Given a place to insert data into an HTML document (that is, a place where a developer has allowed untrusted data to be included in the DOM), there are two ways to inject code:

Injecting UP
: The most common way is to close the current context and start a new code context. For example, this is what you do when you close an HTML attribute with a "> and start a new <script> tag. This attack closes the original context (going up in the hierarchy) and then starts a new tag that will allow script code to execute. Remember that you may be able to skip many layers up in the hierarchy when trying to break out of your current context. For example, a </script> tag may be able to terminate a script block even if it is injected inside a quoted string inside a method call inside the script. This happens because the HTML parser runs before the JavaScript parser.

Injecting DOWN

The less common way to perform XSS injection is to introduce a code subcontext without closing the current context. For example, if the attacker is able to change into < img src="javascript:alert(document.cookie)" /> they do not have to break out of the HTML attribute context. Instead, they introduce a subcontext that allows scripting within the src attribute (in this case a javascript url). Another example is the expression() functionality in CSS properties. Even though you may not be able to escape a quoted CSS property to inject up, you may be able to introduce something like xss:expression(document.write(document.cookie)) without ever leaving the current context.

There's also the possibility of injecting directly in the current context. For example, if you take untrusted input and put it directly into a JavaScript context. While insane, accepting code from an attacker is more common than you might think in modern applications. Generally it is impossible to secure untrusted code with escaping (or anything else). If you do this, your application is just a conduit for attacker code to get running in your users' browsers.

The rules in this document have been designed to prevent both UP and DOWN varieties of XSS injection. To prevent injecting up, you must escape the characters that would allow you to close the current context and start a new one. To prevent attacks that jump up several levels in the DOM hierarchy, you must also escape all the characters that are significant in all enclosing contexts. To prevent injecting down, you must escape any characters that can be used to introduce a new sub-context within the current context.

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 rule 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.

The OWASP ESAPI project has created an escaping library in a variety of languages including Java, .NET, PHP, Classic ASP, Cold Fusion, Python, and Haskell. The ESAPI library can be used for escaping as described here and also for decoding (aka canonicalization), which is critical for input validation. Microsoft provides an encoding library named AntiXSS (http://www.codeplex.com/AntiXSS) .

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.

```
<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
```

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; is not recommended
/ --> &#x2F;     forward slash is included as it helps end an HTML entity
```

See the ESAPI reference implementation (http://code.google.com/p/owasp-esapi-

java/source/browse/trunk/src/main/java/org/owasp/esapi/codecs/HTMLEntityCodec.java) of HTML entity escaping and unescaping.

```
String safe = ESAPI.encoder().encodeForHTML( request.getParameter( "input" ) );
```

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 |.

See the ESAPI reference implementation (http://code.google.com/p/owasp-esapi-java/source/browse/trunk/src/main/java/org/owasp/esapi/codecs/HTMLEntityCodec.java) of HTML entity escaping and unescaping.

```
String safe = ESAPI.encoder().encodeForHTMLAttribute( request.getParameter( "input" ) );
```

RULE #3 - JavaScript Escape Before Inserting Untrusted Data into HTML JavaScript Data Values

Rule #3 concerns the JavaScript event handlers that are specified on various HTML elements. The only safe place to put untrusted data into these event handlers as a quoted "data value." Including untrusted data inside any other code block is quite dangerous, as it is very easy to switch into an execution context, 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. If an event handler is quoted, breaking out requires the corresponding quote. The reason this rule is so broad is that developers frequently leave event handler 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 |. 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.

See the ESAPI reference implementation (http://code.google.com/p/owasp-esapi-java/source/browse/trunk/src/main/java/org/owasp/esapi/codecs/JavaScriptCodec.java) of JavaScript escaping and unescaping.

```
String safe = ESAPI.encoder().encodeForJavaScript( request.getParameter( "input" ) );
```

RULE #4 - CSS Escape 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</style>       property value
```

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. Prevent switching out of the property value and into another property or attribute. Also prevent switching into an expression or other property value that allows scripting. 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 to prevent XSS attacks for both quoted and unquoted attributes.

See the ESAPI reference implementation (http://code.google.com/p/owasp-esapi-java/source/browse/trunk/src/main/java/org/owasp/esapi/codecs/CSSCodec.java) of CSS escaping and unescaping.

```
String safe = ESAPI.encoder().encodeForCSS( request.getParameter( "input" ) );
```

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.

See the ESAPI reference implementation (http://code.google.com/p/owasp-esapi-java/source/browse/trunk/src/main/java/org/owasp/esapi/codecs/PercentCodec.java) of URL escaping and unescaping.

```
String safe = ESAPI.encoder().encodeForURL( request.getParameter( "input" ) );
```

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 = ESAPI.validator().isValidInput("URLContext", userURL, "URL", 255, false);
if (isValidURL) {
    <a href="<%=encoder.encodeForHTMLAttribute(userURL)%>">link</a>
}
```

RULE #6 - Use an HTML Policy engine to validate or clean user-driven HTML in an outbound way

```
 import org.owasp.validator.html.*;
 Policy policy = Policy.getInstance(POLICY_FILE_LOCATION);
 AntiSamy as = new AntiSamy();
 CleanResults cr = as.scan(dirtyInput, policy);
 MyUserDAO.storeUserProfile(cr.getCleanHTML()); // some custom function
```

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.

Encoding Information

OWASP Development Guide (http://code.google.com/p/owasp-development-guide/wiki/WebAppSecDesignGuide_D6)

Additional XSS Defense (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.

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:

- RSnake: "XSS Cheat Sheet" - http://ha.ckers.org/xss.html

Description of XSS Vulnerabilities

- OWASP article on XSS Vulnerabilities

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

Other Articles in the OWASP Prevention Cheat Sheet Series

- Authentication Cheat Sheet
- Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet
- Forgot Password Cheat Sheet
- Cryptographic Storage Cheat Sheet
- SQL Injection Prevention Cheat Sheet
- Transport Layer Protection Cheat Sheet
- **XSS (Cross Site Scripting) Prevention Cheat Sheet**
- DOM based XSS Prevention Cheat Sheet

Authors and Primary Editors

Jeff Williams - jeff.williams[at]aspectsecurity.com

Jim Manico - jim[at]manico.net

Retrieved from "https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet"
Categories: OWASP Application Security Verification Standard Project | OWASP Enterprise Security API | How To | Cheatsheets

- Powered by MediaWiki OWASP Foundation © 2011