Future of XSS Defense

Jim Manico
VP of Security Architecture
Jim.Manico@whitehatsec.com

May 1, 2012
Jim Manico

- VP Security Architecture, WhiteHat Security
- 15 years of web-based, database-driven software development and analysis experience
- Over 7 years as a provider of secure developer training courses for SANS and others
- **OWASP Connections Committee Chair**
  - *OWASP Podcast Series Producer/Host*
  - *OWASP Cheat-Sheet Series Manager*
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Context</th>
<th>Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>HTML Body</td>
<td>HTML Entity Encode</td>
</tr>
<tr>
<td>String</td>
<td>HTML Attribute</td>
<td>Minimal Attribute 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: URL’s, Attribute encoding, safe URL verification</td>
</tr>
<tr>
<td>String</td>
<td>CSS</td>
<td>Strict structural validation, CSS Hex encoding, good design</td>
</tr>
<tr>
<td>HTML</td>
<td>HTML Body</td>
<td>HTML Validation (JSoup, AntiSamy, HTML Sanitizer)</td>
</tr>
<tr>
<td>Any</td>
<td>DOM</td>
<td>DOM XSS Cheat sheet</td>
</tr>
<tr>
<td>Untrusted JavaScript</td>
<td>Any</td>
<td>Sandboxing</td>
</tr>
<tr>
<td>JSON</td>
<td>Client parse time</td>
<td>JSON.parse() or json2.js</td>
</tr>
</tbody>
</table>

**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, nosize, noshade, nowrap, ref, rel, rev, rows, rowspan, scrolling, shape, span, summary, tabindex, title, usemap, valign, value, vlink, vspace, width
<span>UNTRUSTED DATA</span>
<input type="text" name="fname" value="UNTRUSTED DATA">
<a href="/site/search?value=UNTRUSTED DATA">clickme</a>
<a href="UNTRUSTED URL">clickme</a>
<iframe src="UNTRUSTED URL"/>
CSS Value Context

<div style="width: UNTRUSTED DATA;">Selection</div>
JavaScript Variable Context

```html
<script>
var currentValue='UNTRUSTED DATA';
</script>

<script>
someFunction('UNTRUSTED DATA');
</script>
```
JSON Parsing Context

```
JSON.parse(UNTRUSTED JSON DATA)
```
### jQuery Methods That Directly Update the DOM

<table>
<thead>
<tr>
<th>Method</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.after()</code></td>
<td><code>.prependTo()</code></td>
</tr>
<tr>
<td><code>.append()</code></td>
<td><code>.replaceAll()</code></td>
</tr>
<tr>
<td><code>.appendTo()</code></td>
<td><code>.replaceWith()</code></td>
</tr>
<tr>
<td><code>.before()</code></td>
<td><code>.unwrap()</code></td>
</tr>
<tr>
<td><code>.html()</code></td>
<td><code>.wrap()</code></td>
</tr>
<tr>
<td><code>.insertAfter()</code></td>
<td><code>.wrapAll()</code></td>
</tr>
<tr>
<td><code>.insertBefore()</code></td>
<td><code>.wrapInner()</code></td>
</tr>
<tr>
<td><code>.prepend()</code></td>
<td></td>
</tr>
</tbody>
</table>

**Properly escape before sending untrusted data to these methods!**
Jquery Encoding with JQencoder

- Contextual encoding is a crucial technique needed to stop all types of XSS.
- **jqencoder** is a jQuery plugin that allows developers to do contextual encoding in JavaScript to stop DOM-based XSS

  - `$('#element').encode('html', cdata);`
Best Practice: DOM Based XSS Defense

- Untrusted data should only be treated as displayable text
- JavaScript encode and delimit untrusted data as quoted strings
- Use `document.createElement("...")`, `element.setAttribute("...","value")`, `element.appendChild(…)` etc. to build dynamic interfaces (safe attributes only)
- Avoid use of HTML rendering methods
- Make sure that any untrusted data passed to `eval()` methods is delimited with string delimiters and enclosed within a closure
  - `eval(someFunction('UNTRUSTED DATA'))`;
FUTURE of XSS Defense
XSS Defense Future Technologies

• Capabilities based JavaScript (CAJA) from Google
  - Subset’s the JavaScript language using server and client controls

• Content Security Policy
  - Mozilla/W3C standard to block inline JavaScript execution

• Auto-Escaping Templates
  - Web UI frameworks that automatically sanitize output

• ECMAScript 5 Object Sealing, Freezing & Extension Prevention
  - Sealing JavaScript objects to prevents other code from deleting/changing

• HTML 5 Frame Sandboxing
  - Lock down contents of a frame using different policies
Google CAJA: Subset of JavaScript

- Caja sanitizes JavaScript into Cajoled JavaScript
- Caja uses multiple sanitization techniques
  - Caja uses STATIC ANALYSIS when it can
  - Caja modifies JavaScript to include additional run-time checks for additional defense
CAJA workflow

• The web app loads the Caja runtime library which is written in JavaScript

• All un-trusted scripts must be provided as Caja source code to be statically verified and cajoled by the Caja sanitizer

• The sanitizer's output is either included directly in the containing web page or loaded by the Caja runtime engine
Caja Compliant JavaScript

• A Caja-compliant JavaScript program is one which
  - is statically accepted by the Caja sanitizer
  - does not provoke Caja-induced failures when run cajoled

• Such a program should have the same semantics whether run cajoled or not
Most of Caja’s complexity is needed to defend against JavaScript's rules regarding the binding of "this".

JavaScript's rules for binding "this" depends on whether a function is invoked
- by construction
- by method call
- by function call
- or by reflection

If a function written to be called in one way is instead called in another way, its "this" might be rebound to a different object or even to the global environment.
Context Aware Auto-Escaping

• Context-Sensitive Auto-Sanitization (CSAS) from Google
  - Runs during the compilation stage of the Google Closure Templates to add proper sanitization and runtime checks to ensure the correct sanitization.

• Java XML Templates (JXT) from OWASP by Jeff Ichnowski
  - Fast and secure XHTML-compliant context-aware auto-encoding template language that runs on a model similar to JSP.

• Apache Velocity Auto-Escaping by Ivan Ristic
  - Fast and secure XHTML-compliant context-aware auto-encoding template language that runs on a model similar to JSP.
Auto Escaping Tradeoffs

• Developers need to write highly compliant templates
  - No "free and loose" coding like JSP or raw PHP
  - Requires extra time but increases quality

• These technologies often do not support complex contexts
  - Some are not context aware (really really bad)
  - Some choose to let developers disable auto-escaping on a case-by-case basis (really bad)
  - Some choose to encode wrong (bad)
  - Some choose to reject the template (better)
### Experimental Minimal Encoding Rules

The following examples demonstrate experimental minimal encoding rules for XSS prevention.

<table>
<thead>
<tr>
<th>Context</th>
<th>Code Sample</th>
<th>Rules</th>
</tr>
</thead>
</table>
| JavaScript, quoted string in a script block | `<script>alert("Hello "+%= UNTRUSTED DATA %");
</script>` | - Use these escapes: `\v \v \b \t \f \n`  
- For any other character in range `0..0x1f`, use hex escapes  
- If using non-Unicode charset, any character above `0x7e`, use `\u` encoding |
| JavaScript, quoted string in an event handler attribute | `onclick="alert(\%= UNTRUSTED DATA %)";` | - Use these escapes: `\v \v \b \t \f \n`  
- Use hex escapes for these characters: `" &  
- For any other character in range `0..0x1f`, use hex escapes  
- If using non-Unicode charset, any character above `0x7e`, use `\u` encoding |
| HTML Body (up to HTML 4.01): | `<div>%= UNTRUSTED DATA %</div>` | - HTML Entity encode `&`  
- specify charset in metatag to avoid UTF7 XSS |
| XHTML Body: | `<div>%= UNTRUSTED DATA %</div>` | - HTML Entity encode `&`  
- limit input to charset [http://www.w3.org/TR/2008/REC-xml-20081126/#charsets](http://www.w3.org/TR/2008/REC-xml-20081126/#charsets) |

**Related Articles**

- OWASP Cheat Sheets Project Homepage
- Cheat Sheets
- OWASP Cheat Sheet Series
  - OWASP Top Ten Cheat Sheet
  - Authentication Cheat Sheet
  - Cross-Site Request Forcing (CSRF) Prevention Cheat Sheet
Content Security Policy

• Externalize all JavaScript within Web pages
  - No inline script tag
  - No inline JavaScript for onclick or other handling events
  - Push all JavaScript to formal .js files using event binding

• Define the policy for your site and whitelist the allowed domains where the externalized JavaScript is located

• Add the X-Content-Security-Policy response header to instruct the browser that CSP is in use

• Will take 3-5 years for wide adoption and support
JavaScript Object Sealing, Freezing and Extension Prevention

• **Object.freeze(obj) and Object.isFrozen(obj)**
  - object properties can no longer be modified in any way

• **Object.seal(obj) and Object.isSealed(obj)**
  - new properties cannot be added
  - existing properties cannot be changed to accessors (and vice versa)
  - existing property **VALUES** can be changed

• **Object.preventExtensions(obj) and Object.isExtensible(obj)**
  - new properties cannot be added
  - existing properties can be deleted
  - existing properties **can** be changed
HTML5 Frame Sandboxing

<iframe src="demo_iframe_sandbox_origin.htm" sandbox="allow-same-origin allow-scripts"></iframe>

allow-same-origin: Allows the content to be treated as being from the same origin as the containing document. If this keyword is not used, the embedded content is treated as being from a unique origin.

allow-top-navigation: Allows the embedded browsing context to navigate (load) content from the top-level browsing context. If this keyword is not used, this operation is not allowed.

allow-forms: Allows the embedded browsing context to submit forms. If this keyword is not used, this operation is not allowed.

allow-scripts: Allows the embedded browsing context to run scripts (but not create pop-up windows). If this keyword is not used, this operation is not allowed.

<iframe src="demo_iframe_sandbox_origin.htm" sandbox=""></iframe>
## XSS Defense, Future?

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Context</th>
<th>Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric, Type safe language</td>
<td>Doesn’t Matter</td>
<td>Auto Escaping Templates, Content Security Policy, HTML5 Sandboxing</td>
</tr>
<tr>
<td>String</td>
<td>HTML Body</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>HTML Attribute, quoted</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>HTML Attribute, unquoted</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>GET Parameter</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>Untrusted URL</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>CSS</td>
<td></td>
</tr>
<tr>
<td>Untrusted JavaScript</td>
<td>Any</td>
<td></td>
</tr>
<tr>
<td>HTML</td>
<td>HTML Body</td>
<td></td>
</tr>
<tr>
<td>Any</td>
<td>DOM</td>
<td></td>
</tr>
<tr>
<td>Untrusted JavaScript</td>
<td>Any</td>
<td></td>
</tr>
<tr>
<td>JSON</td>
<td>Client parse time</td>
<td>JSON.parse()</td>
</tr>
</tbody>
</table>
Thank You

Jim Manico
VP of Security Architecture
Jim.Manico@whitehatsec.com

A BIG THANK YOU TO:
Gaz Heyes
Abe Kang
Mike Samuel
Jeff Ichnowski
Adam Barth
Jeff Williams
many many others…