Ghosts of XSS Past, Present and Future

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

January 20, 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, Aspect Security and others
• **OWASP Connections Committee Chair**
  - OWASP Podcast Series Producer/Host
  - OWASP Cheat-Sheet Series Manager
XSS: Why so Serious?

- Session hijacking
- Site defacement
- Network scanning
- Undermining CSRF defenses
- Site redirection/phishing
- Load of remotely hosted scripts
- Data theft
- Keystroke logging
- Getting “Scrooged”
Past XSS Defensive Strategies

• 1990’s style XSS prevention
• Eliminate <, >, &, "", ' characters?
• Eliminate all special characters?
• Disallow user input?
• Global filter?
Past XSS Defensive Strategies

- 1990’s style XSS prevention
- Eliminate <, >, &, " , ' characters?
- Eliminate all special characters?
- Disallow user input?
- Global filter?

WHY WON’T THIS WORK?
**XSS Defense, 1990’s**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any Data</td>
<td>Input Validation</td>
</tr>
</tbody>
</table>

#absolute-total-fail
Past XSS Defensive Strategies

- Y2K style XSS prevention
- HTML Entity Encoding
- Replace characters with their 'HTML Entity' equivalent
- Example: replace the "<" character with '&lt;"
Past XSS Defensive Strategies

• Y2K style XSS prevention
• HTML Entity Encoding
• Replace characters with their 'HTML Entity’ equivalent
• Example: replace the "<" character with "&lt;"

WHY WON’T THIS WORK?
## XSS Defense, 2000

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any Data</td>
<td>HTML Entity Encoding</td>
</tr>
</tbody>
</table>
# XSS Defense, 2000

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any Data</td>
<td>HTML Entity Encoding</td>
</tr>
</tbody>
</table>
Danger: Multiple Contexts

Browsers have multiple contexts that must be considered!

- HTML Body
- HTML Attributes
- `<STYLE>` Context
- `<SCRIPT>` Context
- URL Context
Past XSS Defensive Strategies

1. All untrusted data must first be canonicalized
   - Reduced to simplest form

2. All untrusted data must be validated
   - Positive Regular Expressions
   - Blacklist Validation

3. All untrusted data must be contextually encoded
   - HTML Body
   - Quoted HTML Attribute
   - Unquoted HTML Attribute
   - Untrusted URL
   - Untrusted GET parameter
   - CSS style value
   - JavaScript variable assignment
# XSS Defense, 2007

<table>
<thead>
<tr>
<th>Context</th>
<th>Defense</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML Body</td>
<td>HTML Entity Encoding</td>
</tr>
<tr>
<td>HTML Attribute</td>
<td>HTML Attribute Encoding</td>
</tr>
<tr>
<td>JavaScript variable assignment</td>
<td>JavaScript Hex Encoding</td>
</tr>
<tr>
<td>JavaScript function parameter</td>
<td>JavaScript Hex Encoding</td>
</tr>
<tr>
<td>CSS Value</td>
<td>CSS Hex Encoding</td>
</tr>
<tr>
<td>GET Parameter</td>
<td>URL Encoding</td>
</tr>
<tr>
<td>Untrusted URL</td>
<td>HTML Attribute Encoding</td>
</tr>
<tr>
<td>Untrusted HTML</td>
<td>HTML Attribute Encoding</td>
</tr>
</tbody>
</table>

Context

- **HTML Body**
- **HTML Attribute**
- **JavaScript variable assignment**
- **JavaScript function parameter**
- **CSS Value**
- **GET Parameter**
- **Untrusted URL**
- **Untrusted HTML**

Defense

- **HTML Entity Encoding**
- **HTML Attribute Encoding**
- **JavaScript Hex Encoding**
- **CSS Hex Encoding**
- **URL Encoding**
- **HTML Attribute Encoding**
- **AntiSamy**
- **Jsoup**
I Got Some BAD NEWS
CSS Pwnage Test Case

- `<div style="width: <%=temp3%>;""> Mouse over </div>`
- `temp3 = ESAPI.encoder().encodeForCSS("expression(alert(String.fromCharCode(88,88,88)))");`  
- `<div style="width: expression(alert(String.fromCharCode(88,88,88)))"> Mouse over </div>`
- Pops in at least IE6 and IE7.
- `lists.owasp.org/pipermail/owasp-esapi/2009-February/000405.html`
Simplified DOM Based XSS Defense

• 1. Initial loaded page should only be static content.
• 2. Load JSON data via AJAX.
• 3. Only use the following methods to populate the DOM
  - `Node.textContent`
  - `document.createTextNode`
  - `Element.setAttribute`

References: http://www.educatedguesswork.org/2011/08/guest_post_adam_barth_on_three.html and Abe Kang
Dom XSS Oversimplification Danger

• Element.setAttribute is one of the most dangerous JS methods

• If the first element to setAttribute is any of the JavaScript event handlers or a URL context based attribute ("src", "href", "backgroundImage", "background", etc.) then pop.

References: http://www.educatedguesswork.org/2011/08/guest_post_adam_barth_on_three.html and Abe Kang
Best Practice: DOM Based XSS Defense I

• 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
• Avoid use of HTML rendering methods
• Understand the dataflow of untrusted data through your JavaScript code. If you do have to use the methods above remember to HTML and then JavaScript encode the untrusted data
• Make sure that any untrusted data passed to `eval()` methods is delimited with string delimiters and enclosed within a closure or JavaScript encoded to N-levels based on usage and wrapped in a custom function
Best Practice: DOM Based XSS Defense II

- Limit the usage of dynamic untrusted data to right side operations. And be aware of data which may be passed to the application which look like code (eg. location, eval()).

- When URL encoding in DOM be aware of character set issues as the character set in JavaScript DOM is not clearly defined

- Limit access to properties objects when using object[x] access functions

- Don’t eval() JSON to convert it to native JavaScript objects. Instead use JSON.toJSONString() and JSON.parse()

- Run untrusted script in a sandbox (ECMAScript canopy, HTML 5 frame sandbox, etc)
JavaScript Sandboxing

• Capabilities JavaScript (CAJA) from Google
  - Applies an advanced security concept, capabilities, to define a version of JavaScript that can be safer than the sandbox

• JSReg by Gareth Heyes
  - JavaScript sandbox which converts code using regular expressions
  - The goal is to produce safe Javascript from a untrusted source

• 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
JSReg: Protecting JS with JS

• JavaScript re-writing
  - Parses untrusted HTML and returns trusted HTML
  - Utilizes the browser JS engine and regular expressions
  - No third-party code

• First layer is an iframe used as a safe throw away box

• The entire JavaScript objects/properties list was whitelisted by forcing all methods to use suffix/prefix of "$"

• Each variable assignment was then localized using var to force local variables

• Each object was also checked to ensure it didn’t contain a window reference
XSS Defense, Today
## XSS Defense, Today

<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>Cast to Numeric</td>
</tr>
<tr>
<td>String</td>
<td>HTML Body</td>
<td>HTML Entity Encode</td>
</tr>
<tr>
<td>String</td>
<td>HTML Attribute, quoted</td>
<td>Minimal Attribute Encoding</td>
</tr>
<tr>
<td>String</td>
<td>HTML Attribute, unquoted</td>
<td>Maximum 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>
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
  - 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)
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
## 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, 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…