Advanced XSS Defense
Part 1: Basic XSS Defense

Jim Manico
OWASP Board Member
Independent Security Instructor

Eoin Keary
OWASP Board Member
CTO BCC Risk Advisory
www.bccriskadvisory.com

Copyright © Jim Manico and Eoin Keary.
Where are we going?

Stop XSS!

- What is Cross Site Scripting (XSS)
- Reflective XSS
- Stored XSS
- Contextual Output Encoding
- HTML Sanitization
- Testing for XSS
## What is XSS?

<table>
<thead>
<tr>
<th>Cross-site Scripting (XSS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attacker driven JavaScript</td>
</tr>
<tr>
<td>Most common web vulnerability</td>
</tr>
<tr>
<td>Easy vulnerability to find via auditing</td>
</tr>
<tr>
<td>Easy vulnerability to exploit</td>
</tr>
<tr>
<td>Certain types of XSS are very complex to fix</td>
</tr>
<tr>
<td>Difficult to fix all XSS for a large app</td>
</tr>
<tr>
<td>Easy to re-introduce XSS in development</td>
</tr>
<tr>
<td>Significant business and technical impact potential</td>
</tr>
</tbody>
</table>
# XSS Attack Payload Types

<table>
<thead>
<tr>
<th>Payload Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session hijacking</td>
</tr>
<tr>
<td>Site defacement potential</td>
</tr>
<tr>
<td>Network scanning</td>
</tr>
<tr>
<td>Undermining CSRF defenses</td>
</tr>
<tr>
<td>Site redirection/phishing</td>
</tr>
<tr>
<td>Data theft</td>
</tr>
<tr>
<td>Keystroke logging</td>
</tr>
<tr>
<td>Loading of remotely hosted scripts</td>
</tr>
</tbody>
</table>
Input Example

Consider the following URL:

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

6  <h3> Thank you for your comments! </h3>
7  You wrote:
8  <p/>
9  Great Site!
10 <p/>

How can an attacker misuse this?
# XSS Variants

<table>
<thead>
<tr>
<th>Reflected/Transient</th>
<th>Data provided by a client is immediately used by server-side scripts to generate a page of results for that user.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Search engines</td>
</tr>
<tr>
<td>Stored/Persistent</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>DOM based XSS</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>
Reflected XSS

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 Code Sample

//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
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

http://www.com/index.jsp#name=<script>alert(document.cookie)</script>

<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 is NOT embedded in the HTML
## Test for Cross-Site Scripting

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make note of all pages that display input originating from current or other users</td>
<td>Make note of all pages that display input originating from current or other users. Keep track of these pages and their input fields.</td>
</tr>
<tr>
<td>Test by inserting malicious script or characters to see if they are ultimately displayed back to the user</td>
<td>Insert malicious script or characters into the input fields of the pages that you have noted. Watch for the script to be displayed back to the user.</td>
</tr>
<tr>
<td>Examine code to ensure that application data is HTML encoded before being rendered to users</td>
<td>Examine the code of the application to ensure that all data, especially user input, is properly HTML encoded before it is rendered to users.</td>
</tr>
<tr>
<td>Very easy to discover XSS via dynamic testing</td>
<td>Dynamic testing is effective for discovering XSS vulnerabilities.</td>
</tr>
<tr>
<td>More difficult to discover via code review</td>
<td>Code review is more complex and time-consuming but can be more effective in some cases.</td>
</tr>
</tbody>
</table>

Eoin Keary & Jim Manico

Copyright 2014 – all rights reserved.
Test for Cross-Site Scripting

Remember the three common types of attacks:

- Input parameters that are rendered directly back to the user
- Server-Side
- Client-Side
- Input that is rendered within other pages
- Hidden fields are commonly vulnerable to this exploit as there is a perception that hidden fields are read-only
- Error messages that redisplay user input
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

```
<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`

```
<SCRIPT> <% = userdata %> </SCRIPT>
```
Danger: XSS Weak Defense Used

Getting rid of XSS is a difficult task

How can we prevent XSS in our web application

Eliminate <, >, &,, ', " characters?

Eliminate all special characters?

Disallow user input? (not possible)

Global filter?

Why won't these strategies work?
XSS Defense: The Solution?

Depends on the type of user input
- HTML, Strings, Uploaded Files

Depends on **where** user input is displayed in an HTML document
- HTML Body
- HTML Attribute
- JavaScript Variable Assignment

Several defensive techniques needed depending on context
- Input Validation
- Output Encoding
- Sandboxing
# HTML Entity Encoding: The Big 6

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>&amp; &amp;</td>
</tr>
<tr>
<td>2.</td>
<td>&lt; &lt;</td>
</tr>
<tr>
<td>3.</td>
<td>&gt; &gt;</td>
</tr>
<tr>
<td>4.</td>
<td>&quot; &quot;</td>
</tr>
<tr>
<td>5.</td>
<td>` &amp;#x27;</td>
</tr>
<tr>
<td>6.</td>
<td>/ &amp;#x2F;</td>
</tr>
</tbody>
</table>
Output Encoding Code Sample

StringBuffer buff = new StringBuffer();
if ( value == null ) {
    return null;
}
for(int i=0; i<value.length(); i++) {
    char ch = value.charAt(i);
    if ( ch == '&' )
        buff.append("&");
    else if ( ch == '<' )
        buff.append("&lt;");
    else if ( ch == '>' )
        buff.append("&gt;");
    else if ( Character.isWhitespace(ch) )
        buff.append(ch);
    else if ( Character.isLetterOrDigit(ch) )
        buff.append(ch);
    else if ( Integer.valueOf(ch).intValue() >= 20 &&
              Integer.valueOf(ch).intValue() <= 126 )
        buff.append("&" + (int)ch + ";");
}
return buff.toString();
Best Practice: Validate and Encode

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: " + Encode.forHTML(email));
}
else
{
    //log & throw a specific validation exception and fail safely
}
Danger: Multiple Contexts

Different encoding and validation techniques needed for different contexts!
## 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/Attribute</td>
<td>HTML Entity Encode/HTML Attribute Encoder</td>
</tr>
<tr>
<td>String</td>
<td>Java String</td>
<td>Java String Encoding</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: URL’s, Attribute encoding, safe URL verification</td>
</tr>
<tr>
<td>String</td>
<td>CSS Value</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
# XSS Defense by Data Type and Context

<table>
<thead>
<tr>
<th>Context</th>
<th>Encoding</th>
<th>Freemarker</th>
<th>OWASP Java Encoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML Body</td>
<td>HTML Entity Encode</td>
<td>?html</td>
<td>Encoder.forHtmlContent</td>
</tr>
<tr>
<td>HTML Attribute</td>
<td>HTML Entity Encode</td>
<td></td>
<td>Encoder.forHtmlAttribute</td>
</tr>
<tr>
<td>Java String</td>
<td>Java String Encoding</td>
<td>?j_string</td>
<td>Encoder.forJava</td>
</tr>
<tr>
<td>JavaScript Variable</td>
<td>JavaScript Hex Encoding</td>
<td>?js_string</td>
<td>Encoder.forJavaScript Encoder</td>
</tr>
<tr>
<td>JavaScript Variable</td>
<td></td>
<td></td>
<td>Encoder.forJavaScriptBlock Encoder, Encoder.forJavaScriptAttribute</td>
</tr>
<tr>
<td>GET Parameter</td>
<td>URL Encoding</td>
<td>?url</td>
<td>Encoder.forUriComponent</td>
</tr>
<tr>
<td>Untrusted URL</td>
<td>URL Validation, avoid javascript: URL’s, attribute encoding, safe URL verification</td>
<td>?html</td>
<td>Encoder.forUri</td>
</tr>
<tr>
<td>CSS Value</td>
<td>Strict structural validation, CSS Hex encoding, good design</td>
<td>???</td>
<td>Encoder.forCssString Encoder, Encoder.forCssUrl</td>
</tr>
<tr>
<td>HTML Body</td>
<td>HTML Validation (JSoup, AntiSamy, HTML Sanitizer)</td>
<td>???</td>
<td>OWASP HTML Sanitizer</td>
</tr>
</tbody>
</table>
OWASP Java Encoder Project
https://www.owasp.org/index.php/OWASP_Java_Encoder_Project

• No third party libraries or configuration necessary.
• This code was designed for high-availability/high-performance encoding functionality. Redesigned for performance.
• Simple drop-in encoding functionality
• More complete API (uri and uri component encoding, etc) in some regards.
• This is a Java 1.5 project.
• Last updated February 14, 2013 (version 1.1)
The Problem

Web Page built in Java JSP is vulnerable to XSS

The Solution

```html
<%-- Basic HTML Context --%>
<body><b><%= Encode.forHtml(UNTRUSTED) %>" /></b></body>

<%-- HTML Attribute Context --%>
<input type="text" name="data" value="<%= Encode.forHtmlAttribute(UNTRUSTED) %>" />

<%-- Javascript Block context --%>
<script type="text/javascript">
var msg = "<%= Encode.forJavaScriptBlock(UNTRUSTED) %>"; alert(msg);
</script>

<%-- Javascript Variable context --%>
<button onclick="alert('<%= Encode.forJavaScriptAttribute(UNTRUSTED) %>');">click me</button>
```
HTML Contexts
Encode#forHtmlContent(String)
Encode#forHtmlAttribute(String)
Encode#forHtmlUnquotedAttribute(String)

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

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

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

URI/URL contexts
Encode#forUri(String)
Encode#forUriComponent(String)
XSS in HTML Body

Reflective XSS attack example:

example.com/error?error_msg=You cannot access that file.

Untrusted data may land in a UI snippet like the following:

\(<div><%= \text{request.getParameter(\"error\_msg\")} \%>\</div>\)

Sample test attack payload:

example.com/error?
error_msg=<script>alert(document.cookie)</script>

HTML Encoding stops XSS in this context!
HTML Body Escaping Examples

**OWASP Java Encoder**

```html
<body><b><%= Encode.forHtml(UNTRUSTED) %></b></body>

<p>Title: <%= Encode.forHtml(UNTRUSTED) %></p>

<textarea name="text">
<%= Encode.forHtmlContent(UNTRUSTED) %>
</textarea>

**Freemarker**

```html
<textarea name="text">
${textValue?html}
</textarea>

<p>Title: ${book.title?html}</p>
```
XSS in HTML Attributes

- Where else can XSS go?
  - `<input type="text" name="comments" value=""/>

- What could an attacker put in here?
  - `<input type="text" name="comments"
    value="hello" onmouseover="/*fire attack*/"/>

- Attackers can add event handlers:
  - onMouseOver
  - onLoad
  - onUnload
  - etc...
HTML Attribute Context

- Aggressive escaping is needed when placing untrusted data into typical attribute values like width, name, value, etc.
- This rule is NOT ok for complex attributes like href, src, style, or any event handlers like onblur or onclick.
- Escape all non alpha-num characters with the &\#xHH; format
- This rule is so aggressive because developers frequently leave attributes unquoted
- <div id=DATA></div>
HTML Attribute Escaping Examples

**OWASP Java Encoder**

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

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

**Freemarker**

```html
<input type=text name="data" value="$\{user?xhtml}\">
```

*Freemarker requires that your attributes are double-quoted in order to be safe*
URL Parameter Escaping

Escape **all** non alpha-num characters with the %HH format

```html
<a href="/search?data=<%=DATA %>">

Be careful not to allow untrusted data to drive entire URL’s or URL fragments

This encoding only protects you from XSS at the time of rendering the link

Treat DATA as untrusted after submitted
URL Parameter Escaping Examples

**OWASP Java Encoder**

```java
<%-- Encode URL parameter values --%>
<a href="/search?value=<
%Encode.forUriComponent(parameterValue)%>&order=1#top">

<%-- Encode REST URL parameters --%>
<a href="http://www.codemagi.com/page/<
%Encode.forUriComponent(restUrlParameter)%>">

**Freemarker**

```freemarker
<a id="activate" href="action?token=${token?url}">Click
Here</a>
Handling Untrusted URL’s

1) First validate to ensure the string is a valid URL
2) Avoid Javascript: URL’s
3) Only allow HTTP or HTTPS only
4) Check the URL for malware inbound and outbound
5) Encode URL in the right context of display

<a href="UNTRUSTED URL">UNTRUSTED URL</a>
Escaping when managing complete URL’s

Assuming the untrusted URL has been properly validated....

**OWASP Java Encoder**

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

**Freemarker**

```html
<a href="${untrustedURL?xhtml}">${untrustedURL?xhtml}</a>
```
XSS in JavaScript Context

http://example.com/viewPage?name=Jerry

627  <script>
628   //create variable for Jerry
629   var name = “Jerry”;
630  </script>

- What attacks would be possible?
- What would a %0d%0a in the name parameter do in the output?
JavaScript Escaping Examples

**OWASP Java Encoder**

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

<button onclick="alert('&lt;%= Encode.forJavaScriptAttribute(alertMsg) %&gt;');">click me</button>

<script type="text/javascript">
var msg = '&lt;%= Encode.forJavaScriptBlock(alertMsg) %&gt;';
alert(msg);
</script>

**Freemarker**

```html
<button onclick="alert('&{$message?js_string?xhtml}');">click me</button>
```
XSS in the Style Tag

Applications sometimes take user data and use it to generate presentation style

Consider this example:

http://example.com/viewDocument?background=white
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
## CSS Context: XSS Defense

- Escape **all** non alpha-num characters with the `\HH` format
- `<span style=bgcolor:DATA;>text</style>`
- Do not use any escaping shortcuts like `\”`
- Strong positive structural validation is also required
- If possible, design around this “feature”
  - Use trusted CSS files that users can choose from
  - Use client-side only CSS modification (font size)
XSS in CSS String Context Examples

**OWASP Java Encoder**

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

  ```

**Freemarker**

```html
???
```
Other Encoding Libraries

- **Ruby on Rails**
  - [http://api.rubyonrails.org/classes/ERB/Util.html](http://api.rubyonrails.org/classes/ERB/Util.html)

- **Reform Project**
  - Java, .NET v1/v2, PHP, Python, Perl, JavaScript, Classic ASP

- **ESAPI**
  - PHP.NET, Python, Classic ASP, Cold Fusion

- **.NET AntiXSS Library**
  - [http://wpl.codeplex.com/releases/view/80289](http://wpl.codeplex.com/releases/view/80289)
There are just certain places in HTML documents where you cannot place untrusted data

- Danger: `<a $DATA>`

There are just certain JavaScript functions that cannot safely handle untrusted data for input

- Danger: `<script>eval($DATA);</script>`
HTML Sanitization and XSS
What is HTML Sanitization

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

- HTML sanitization is everywhere
  - TinyMCE/CKEditor Widgets
  - Web forum posts w/markup
  - Javascript-based Windows 8 Store apps
  - Outlook.com
  - Advertisements
Welcome to the TinyMCE editor demo!

Feel free to try out the different features that are provided, please note that the MCImageManager and MCFImageManager 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, its a great resource wiki for understanding how TinyMCE works and integrates.

<table>
<thead>
<tr>
<th>Element</th>
<th>HTML</th>
</tr>
</thead>
</table>
| content   | `<h1 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 MCImageManager and MCFImageManager 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.</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>, its 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>  
<p>And here is a simple table for you to play with.</p>` |

Eoin Keary & Jim Manico
Why are HTML sanitization bugs important?

- **Worst case scenario**
  - Script running from a mail message executes within the security context of the mail application
  - ...from the preview pane that appears automatically
  - Attacker could set up auto-forwarding, impersonate you, steal all your mail, etc.

- **Yet, HTML sanitization bugs are pervasive**
  - Fuzzing? Can be helpful, but difficult
  - Threat modeling? Not so relevant...
  - Smart hackers with some free time – very relevant

And the underlying platforms continue to change. All of them.

This is a **hard problem**.
The Many Bugs – Example #1

Sanitizer Bypass in validator Node.js Module by @NealPoole (https://t.co/5omk5ec2UD)

- Nesting

**Input:** `<scrRedirecRedirect 302t 302ipt type="text/javascript">prompt(1);</scrRedirecRedirect 302t 302ipt>`

**Output:** `<script type="text/javascript">prompt(1);</script>`

Observation: Removing data from markup can create XSS where it didn’t previously exist
The Many Bugs – Example #2

- CVE-2011-1252 / MS11-074
  - SharePoint / SafeHTML

Input:
```html
<style>div{color:rgb(0,0,0)&a=expression(alert(1))}
</style>
```

& → &amp;

Output:
```html
<style>div{color:rgb(0,0,0)&amp;a=expression(alert(1))}
</style>
```

Observations:
- Sanitizer created a delimiter (the semi-colon)
- Legacy IE CSS expression syntax required to execute script
- Context mismatch
  - Sanitizer: “expression” is considered to be in a benign location
  - Browser: “expression” is considered to be the RHS of a CSS property set operation
The Many Bugs – Example #3

- CodeIgniter <= 2.1.1 xss_clean() sanitizer bug
  - (CVE-2012-1915)
  - Credit: Krzysztof Kotowicz (@kkotowicz)
    - Thx @NealPoole and @adam_baldwin!

- Fake tag closure

  Input and Output:

  `<img/src=""> onerror=alert(1)>
  
  `<button/a="">` autofocus onfocus=alert;&#40;1;&#40;></button>`

Observations:

- No content modification required to trigger the vulnerability
- Sanitizer: “>” recognized as closing the IMG or BUTTON element
  - Allows script through, believing it to be in the raw HTML context
- Browser: “>” recognized as an attribute value
  - Browser executes onerror / onfocus event handlers
The Many Bugs – Example #4

- Wordpress 3.0.3 (kses.php)
  - Credit: Mauro Gentile (@sneak_)
    - Thx @superevr!
  - Lower case attribute name check

Input and Output:

```html
<a HREF="javascript:alert(0)">click me</a>
```

Observations:

- No content modification required to trigger the vulnerability
- Sanitizer: Only lower case “href” recognized as an attribute name
- Browser: HREF attribute recognized, javascript: URL executes on click
- Sanitizer and browser don’t agree on what constitutes an attribute name
HTML Sanitization Bugs: What’s the lesson?

1. High level: Parsing / “context management” is hard

Sanitizers must *exactly* emulate client-side parsing

- An opportunity for a vulnerability any time the sanitizer and browser get out of sync
- Sanitizer output must be safe for *all* useragents
  - If this is even possible!
What’s the lesson? (cont.)

2. Parsing is the **difficult** part of sanitization

- The “business logic” is easy
  - Eg: What tags, attributes, CSS, etc. are considered acceptable
- Logically, the sanitizer was built to define this business logic
  - The sanitizer’s value is not derived from being yet another HTML parser!
- All the bugs identified previously are parsing / “context management” related
- Thank you to to @randomdross for this info!
OWASP HTML Sanitizer Project

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

- 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
- Very easy to use.
- It allows for simple programmatic POSITIVE policy configuration. 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.
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("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);
```
Other HTML Sanitizers

- Pure JavaScript, client side HTML Sanitization with CAJA!
  - [http://code.google.com/p/google-caja/wiki/JsHtmlSanitizer](http://code.google.com/p/google-caja/wiki/JsHtmlSanitizer)
  - [https://code.google.com/p/google-caja/source/browse/trunk/src/com/google/caja/plugin/html-sanitizer.js](https://code.google.com/p/google-caja/source/browse/trunk/src/com/google/caja/plugin/html-sanitizer.js)

- Python
  - [https://pypi.python.org/pypi/bleach](https://pypi.python.org/pypi/bleach)

- PHP
  - [http://htmlpurifier.org/](http://htmlpurifier.org/)

- .NET
  - AntiXSS.getSafeHTML/getSafeHTMLFragment

- Ruby on Rails
  - [https://rubygems.org/gems/loofah](https://rubygems.org/gems/loofah)
  - [http://api.rubyonrails.org/classes/HTML.html](http://api.rubyonrails.org/classes/HTML.html)

- Java
  - [https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project](https://www.owasp.org/index.php/OWASP_Java_HTML_Sanitizer_Project)

Eoin Keary & Jim Manico
Summary

<table>
<thead>
<tr>
<th>Stop XSS!</th>
<th>What is Cross Site Scripting (XSS)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reflective XSS</td>
</tr>
<tr>
<td></td>
<td>Stored XSS</td>
</tr>
<tr>
<td></td>
<td>Contextual Output Encoding</td>
</tr>
<tr>
<td></td>
<td>HTML Sanitization</td>
</tr>
<tr>
<td></td>
<td>Testing for XSS</td>
</tr>
</tbody>
</table>
## Part 2: Advanced XSS Defense

### Advanced XSS

- JavaScript and iFrame sandboxing
- Content Security Policy
- Avoid dangerous JavaScript API’s
- Object sealing
- Handle JSON safely
- HTTP Only Cookies
## 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/Attribute</td>
<td>HTML Entity 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: 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, noresize, noshade, nowrap, ref, rel, rev, rows, rowspan, scrolling, shape, span, summary, tabindex, title, usemap, valign, value, vlink, vspace, width
DOM Based XSS Defense

DOM Based XSS is a complex risk

Suppose that x landed in ...
<script>setInterval(x, 2000);</script>

For some Javascript functions, even JavaScript that is properly encoded will still execute!
## Dangerous JavaScript Sinks

<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()/function()/setInterval()/setTimeout(), requestAnimationFrame()</td>
<td>- elem.innerHTML = danger, elem.outerHTML = danger</td>
<td>- onload()</td>
</tr>
<tr>
<td>- script.src(), iframe.src()</td>
<td>- elem.setAttribute(&quot;dangerous attribute&quot;, danger) – attributes like: href, src, onclick, onload, onblur, etc.</td>
<td>- onblur(), etc</td>
</tr>
</tbody>
</table>

Some Safe JavaScript Sinks

- elem.innerHTML(danger)
- formfield.val(danger)

Safe JSON parsing
- JSON.parse() (rather than eval())
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/
# jQuery API’s and XSS

## Dangerous jQuery 1.7.2 Data Types

<table>
<thead>
<tr>
<th>CSS</th>
<th>Some Attribute Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML</td>
<td>URL (Potential Redirect)</td>
</tr>
</tbody>
</table>

## jQuery methods that directly update DOM or can execute JavaScript

- `$(())` or `jQuery()`
- `.attr()`
- `.add()`
- `.css()`
- `.after()`
- `.html()`
- `.animate()`
- `.insertAfter()`
- `.append()`
- `.insertBefore()`
- `.appendTo()`

Note: `.text()` updates DOM, but is safe.

## jQuery methods that accept URLs to potentially unsafe content

- `jQuery.ajax()`
- `jQuery.post()`
- `jQuery.get()`
- `load()`
- `jQuery.getScript()`

---

Don’t send untrusted data to these methods, or properly escape the data before doing so.

---

Eoin Keary & Jim Manico

Copyright 2014 – all rights reserved.
jQuery – But there’s more...

More danger
- 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)

Same safe examples
- .text(danger), .val(danger)

Some serious research needs to be done to identify all the safe vs. unsafe methods
- There are about 300 methods in jQuery

# Client Side Context Sensitive Output Escaping

<table>
<thead>
<tr>
<th>Context</th>
<th>Escaping Scheme</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML Element</td>
<td>( &amp;, &lt;, &gt;, &quot; ) → &amp;entity; ( ', / ) → &amp;#xHH;</td>
<td>$ESAPI.encoder().encodeForHTML()</td>
</tr>
<tr>
<td>HTML Attribute</td>
<td>All non-alphanumeric &lt; 256 → ­</td>
<td>$ESAPI.encoder().encodeForHTMLAttribute()</td>
</tr>
<tr>
<td>JavaScript</td>
<td>All non-alphanumeric &lt; 256 → \xHH</td>
<td>$ESAPI.encoder().encodeForJavaScript()</td>
</tr>
<tr>
<td>HTML Style</td>
<td>All non-alphanumeric &lt; 256 → \HH</td>
<td>$ESAPI.encoder().encodeForCSS()</td>
</tr>
<tr>
<td>URI Attribute</td>
<td>All non-alphanumeric &lt; 256 → %HH</td>
<td>$ESAPI.encoder().encodeForURL()</td>
</tr>
</tbody>
</table>

Encoding methods built into a jQuery-encoder: [https://github.com/chrisisbeef/jquery-encoder](https://github.com/chrisisbeef/jquery-encoder)
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', UNTRUSTED-DATA);`
“Secure” DOM XSS/AJAX Workflow

Initial loaded page should only be static content

Load JSON data via AJAX or via embedded JSON safely per
https://www.owasp.org/index.php/
XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet#JSON_entity_encoding

Only use the following methods to populate the DOM

- Node.textContent
- document.createTextNode
- Element.setAttribute

Caution: Element.setAttribute is one of the most dangerous JS methods

Caution: 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 XSS will pop
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

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()).

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

Sanitize JSON Server-side on input and output. Use JSON.toJSON() and JSON.parse() to parse JSON in the browser safely.

Run untrusted script in a sandbox (ECMAScript object sealing, HTML 5 frame sandbox, etc)
Should you trust all JSON?

"user":
{
    "name": "Jameson",
    "occupation": "Distiller",
    "location": (function() { alert("XSS 1!"); return "somewhere"})(),
    "_location_comment": "Once parsed unsafely, the location XSS will run automatically, as a self-executing function. JSON.parse can help with this, and jQuery's $.parseJSON uses it by default (as do $.ajax, etc)",
    "bio": "<script type='text/javascript'>alert('XSS!');</script>",
    "_bio_comment": "This XSS will execute once it is added to the DOM, if not properly escaped before adding it. This is more of a persistent kind of XSS attack."
}
• 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.
OWASP JSON Sanitizer Project
https://www.owasp.org/index.php/OWASP_JSON_Sanitizer
Server Side JSON Sanitization

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.
# Best Practice: 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
Best Practice: Content Security Policy

Anti-XSS W3C standard

CSP 1.1 Draft 19 published August 2012
- [https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html](https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html)

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
# Best Practice: 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 Content Security Policy              | Developers define which scripts are valid  
|                                            | Browser will only execute supported scripts  
|                                            | Inline JavaScript code is ignored  

## Best Practice: Content Security Policy

### X-Content-Security-Policy

<table>
<thead>
<tr>
<th>Application uses</th>
<th>Content-Security-Policy header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Externalize all JavaScript within web pages</td>
<td>No inline script tag</td>
</tr>
<tr>
<td></td>
<td>No inline JavaScript for onclick or other handling events</td>
</tr>
<tr>
<td></td>
<td>Push all JavaScript to formal .js files using event binding</td>
</tr>
<tr>
<td>Define Content Security Policy</td>
<td>Developers define which scripts are valid</td>
</tr>
<tr>
<td></td>
<td>Browser will only execute supported scripts</td>
</tr>
<tr>
<td></td>
<td>Inline JavaScript code is ignored</td>
</tr>
</tbody>
</table>
## Content Security Policy Directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Script-src</td>
<td>allowed script sources</td>
</tr>
<tr>
<td>Object-src</td>
<td>allowed object sources</td>
</tr>
<tr>
<td>Img-src</td>
<td>allowed image sources (html, css)</td>
</tr>
<tr>
<td>Media-src</td>
<td>allowed media sources</td>
</tr>
<tr>
<td>Style-src</td>
<td>allowed CSS sources</td>
</tr>
<tr>
<td>Frame-src</td>
<td>allowed sources for child frames</td>
</tr>
<tr>
<td>Font-src</td>
<td>allowed font sources (CSS)</td>
</tr>
<tr>
<td>Connect-src</td>
<td>allowed remote destinations</td>
</tr>
<tr>
<td>Default-src</td>
<td>allowed sources for any</td>
</tr>
</tbody>
</table>
Content Security Policy by Example 1

Source:

Site allows images from anywhere, plugin content from a list of trusted media providers, and scripts only from its server:

**X-Content-Security-Policy**: allow 'self'; img-src *; object-src media1.com media2.com; script-src scripts.example.com

Site that loads resources from a content delivery network and does not need framed content or any plugins

**X-Content-Security-Policy:** default-src https://cdn.example.net; frame-src 'none'; object-src 'none'
Best Practice: X-Xss-Protection

- Use the browser’s built in XSS Auditor

- X-Xss-Protection:
  - `[0-1] (mode=block)`
- X-Xss-Protection:
  - `1; mode=block`
XSS Defense with no Encoding!

- Advanced XSS-resistant JavaScript-centric web architecture

1) Deliver main HTML document with static data only in the HTML and embedded JSON
2) Parse JSON and populate the static HTML with safe JavaScript API's
3) Safe JavaScript API's include:
   1) Node.textContent
   2) document.createTextNode
   3) Element.setAttribute (second parameter only)

4) This is not failsafe, you can still do stupid things like:
   1) // The textContent of a <script> element is active content.
   2) var scriptElement = document.createElement('script');
   3) scriptElement.textContent = userData.firstName; // XSS!
   4) document.body.appendChild(scriptElement);

5) Adam Barth was right!
   1) http://www.educatedguesswork.org/2011/08/guest_post_adam_barth_on_three.html
## Summary

### Advanced XSS
- JavaScript and iFrame sandboxing
- Content Security Policy
- Avoid dangerous JavaScript API’s
- Object sealing
- Handle JSON safely
- HTTPOnly Cookies
<script>alert('THANK YOU!');</script>

@manicode
jim@owasp.org
jim@manico.net

http://slideshare.net/jimmanmanico