

# XSS Defense Module

Eoin Keary

Jim Manico

# What is XSS?

- Cross-site Scripting (XSS)
- Attacker driven JavaScript
  - Most common web vulnerability
  - Easy vulnerability to find via auditing
  - Easy vulnerability to exploit
  - Certain types of XSS are very complex to fix
  - Difficult to fix all XSS for a large app
  - Easy to re-introduce XSS in development
  - Significant business and technical impact potential

# XSS Attack Payload Types

- Session hijacking
- Site defacement potential
- Network scanning
- Undermining CSRF defenses
- Site redirection/phishing
- Data theft
- Keystroke logging
- Loading of remotely hosted scripts

# 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?

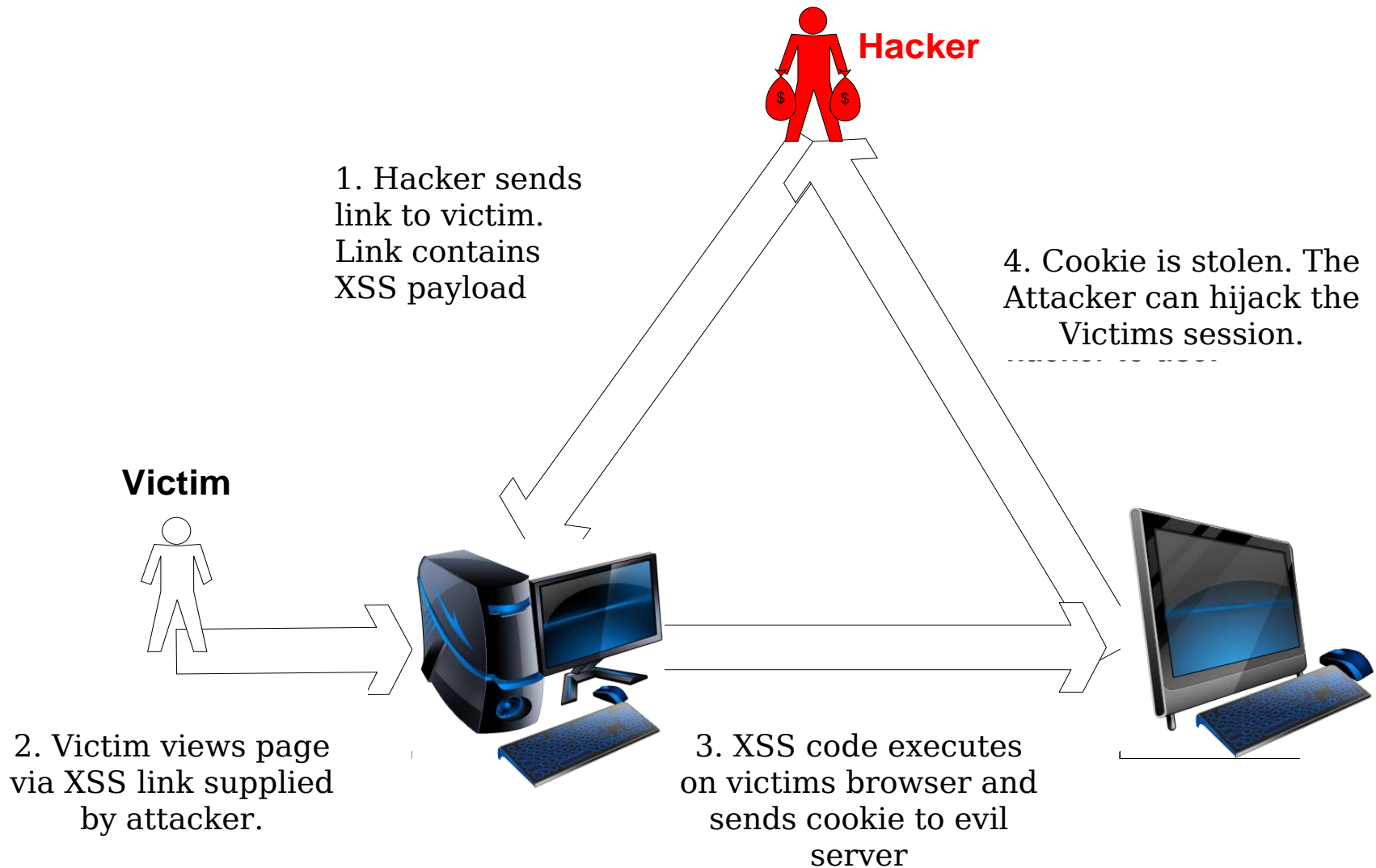
# XSS Variants

- Reflected/Transient
  - Data provided by a client is immediately used by server-side scripts to generate a page of results for that user.
  - Search engines
- Stored/Persistent
  - 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

# XSS Variants

- DOM based 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



# 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>>



# Reflected XSS Demo - Keylogger

The screenshot displays a Windows Internet Explorer browser window titled "KeyloggerDemo". The address bar shows the URL `http://127.0.0.1/dvwa/keylogger/demo/webV/?article=helpme`. The browser's address bar and search bar are partially obscured by the large "Keylogger" title. The browser's menu bar includes "File", "Edit", "View", "Favorites", "Tools", and "Help". The Favorites bar contains "KeyloggerDemo", "OWASP Application Security...", "Suggested Sites", "Web Slice Gallery", "dvwa - Damn Vulnerable We...", and "SQL Injection Cheat Sheet".

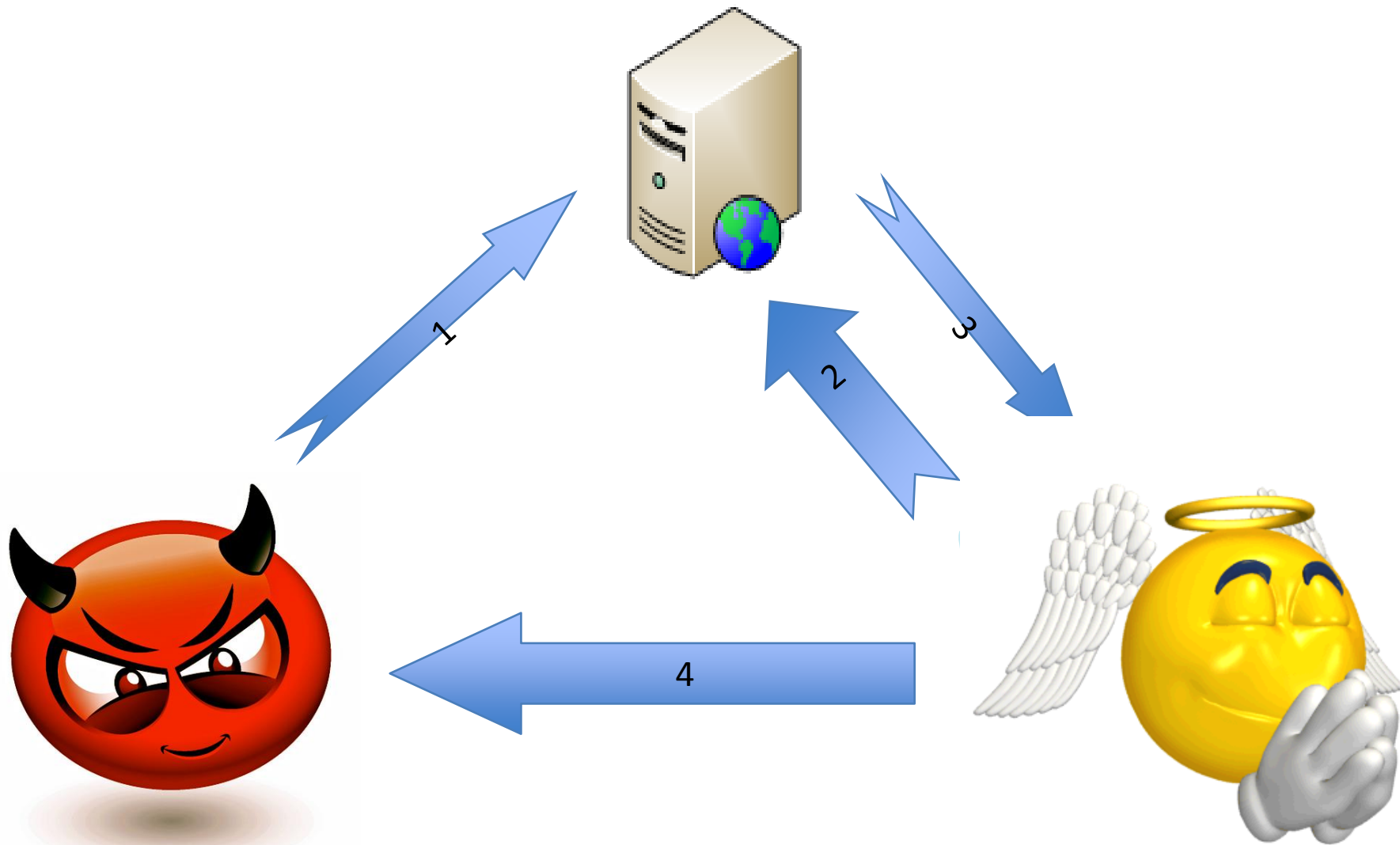
The main content area of the browser is blue and contains a login form with "Username" and "Password" labels, input fields, and a "Login" button. Below the form, the text "Article Search - helpme" is displayed, followed by two red links labeled "Link1" and "Link2". At the bottom of the content area, the text "Access Denied." is visible.

Overlaid on the browser are three WordPad windows:

- index.php - WordPad**: Shows PHP code for session management and content rendering. The code includes `session_start()`, `if (isset($_GET['article']) && 'empty($_GET['article'])`, and `<div style="width:100%; height:100%; background-color:blue; color:white; text-align:center; padding:10px;>`.
- keylogger.js - WordPad**: Shows JavaScript code for a keylogger: `var keys=""; document.onkeypress = function(e) { get = window.event?event:e; key = get.keyCode?get.keyCode:get.charCode; /* get ascii code */`.
- keylogger.php - WordPad**: Shows PHP code for handling keylogger data: `if (isset($_GET['log'])) { $str = isset($_GET['c']) ? $_GET['c'] : false; if ($str) { $f = fopen('data.txt', 'a+'); fputs($f, $str); fclose($f); } } else { $_SESSION['auth'] = 'Access Denied'; if ($_SESSION['auth']`.

The Windows taskbar at the bottom shows the Start button, several open applications, and the system tray with the time 10:20.

# Persistent/Stored XSS



# Persistent/Stored XSS

```
//ViewComments.jsp

<%

int id = Integer.parseInt(Request.getParameter("id"));

Statement stmt = conn.createStatement();

ResultSet rs =

stmt.executeQuery("select * from forum where id=" + id);

if (rs != null) {

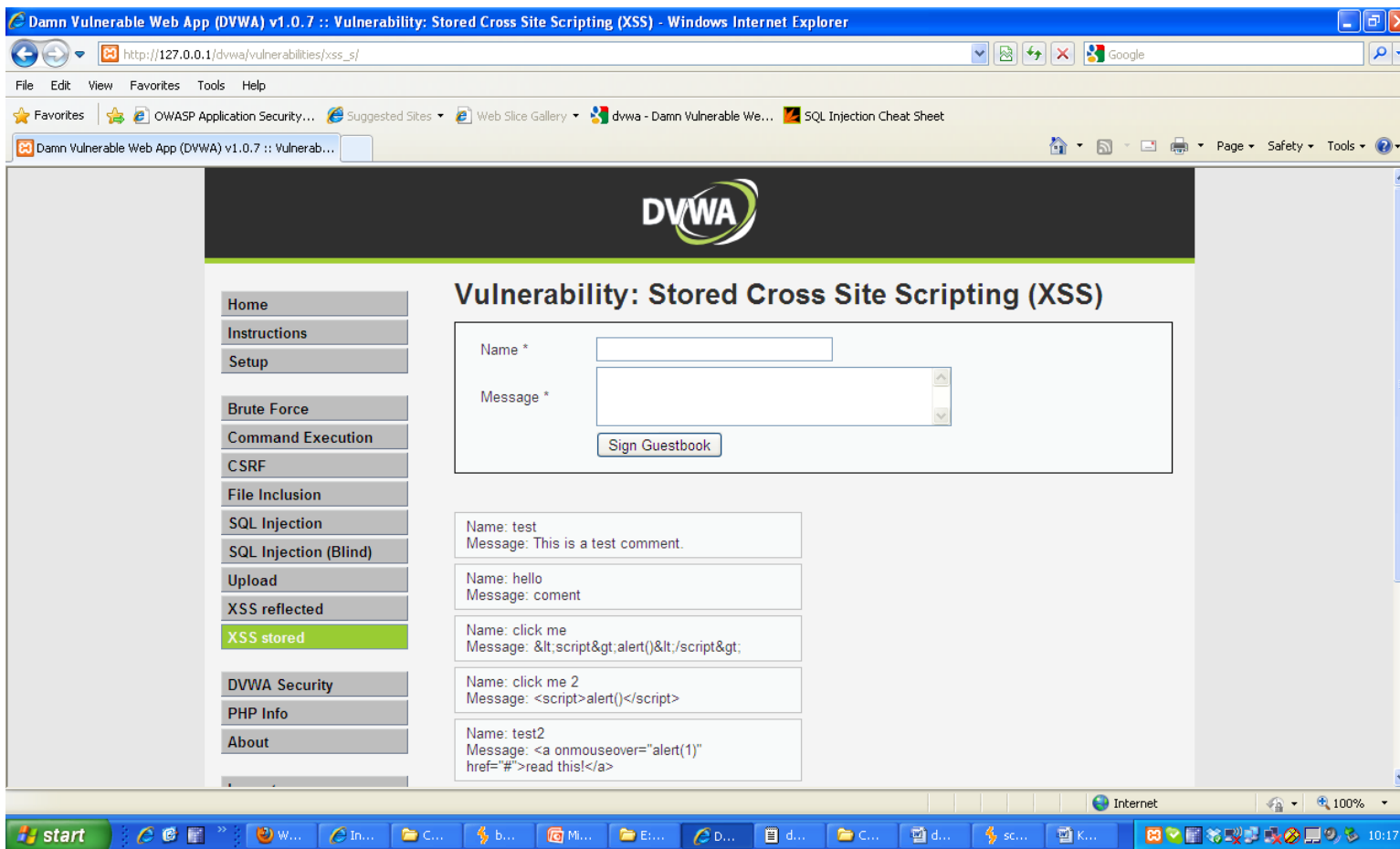
rs.next();

String name = rs.getString("comment");

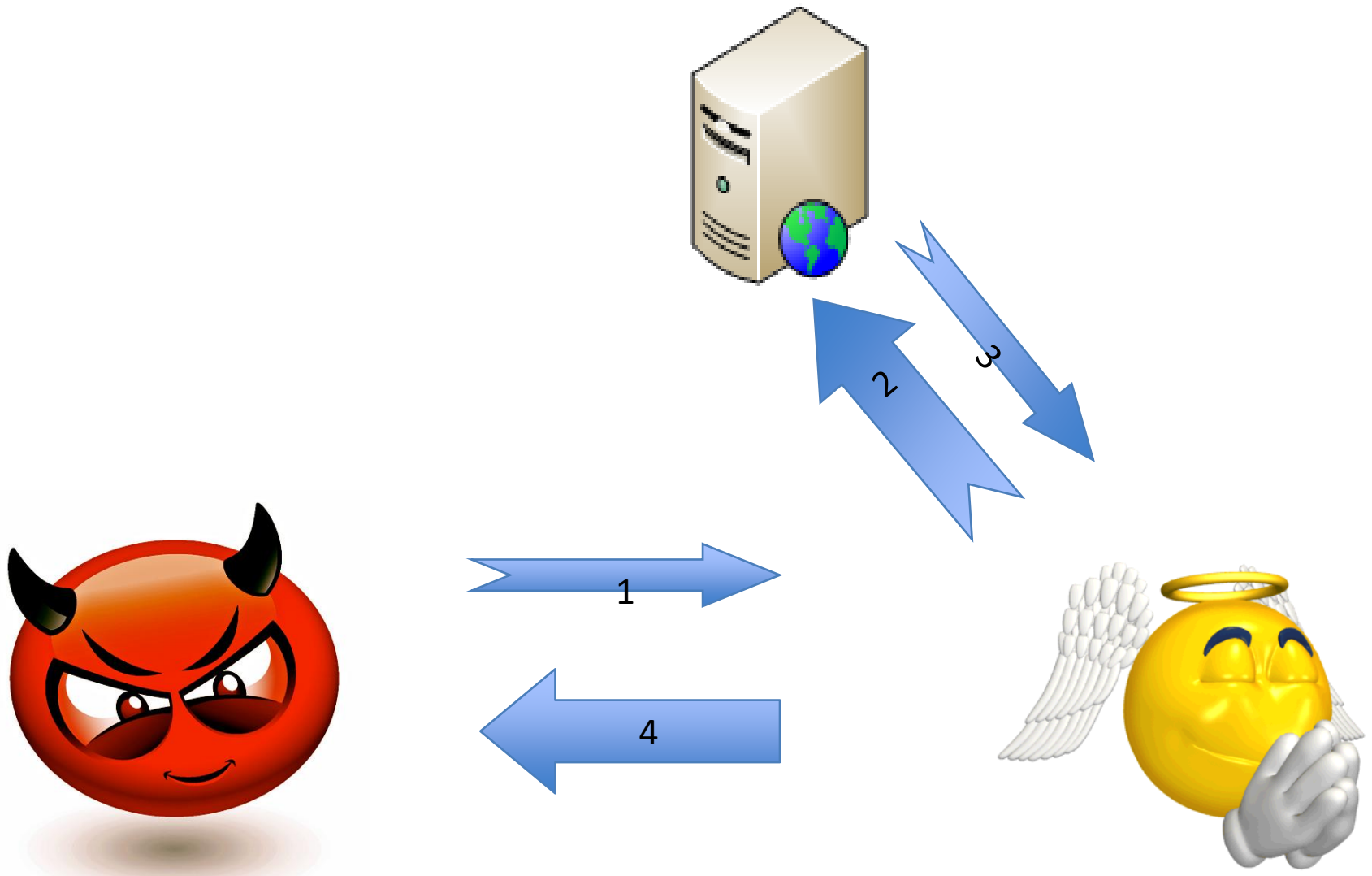
%>

User Comment : <%= comment %>
```

# Persistent/Stored XSS - Demo



# 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

# 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?

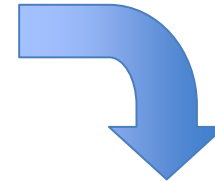
- Depends on the type of user input
  - HTML, Strings, Uploaded Files
- Depends on **where** user input is displayed in an HTML document
  - HTML Body
  - HTML Attribute
  - JavaScript Variable Assignment
- Several defensive techniques needed depending on context
  - Input Validation
  - Output Encoding
  - Sandboxing

# XSS Defense

Data Type	Context	Defense
Numeric, Type safe language	Doesn't Matter	Cast to Numeric
String	HTML Body	HTML Entity Encode
String	HTML Attribute, quoted	Minimal Attribute Encoding
String	HTML Attribute, unquoted	Maximum Attribute Encoding
String	GET Parameter	URL Encoding
String	Untrusted URL	URL Validation, avoid javascript: URL's, 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

# Best Practice: Validate and Encode User Input

```
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: " + StringEscapeUtils.escapeHtml(email));  
}  
else  
{  
    //log & throw a specific validation exception and fail safely  
}
```

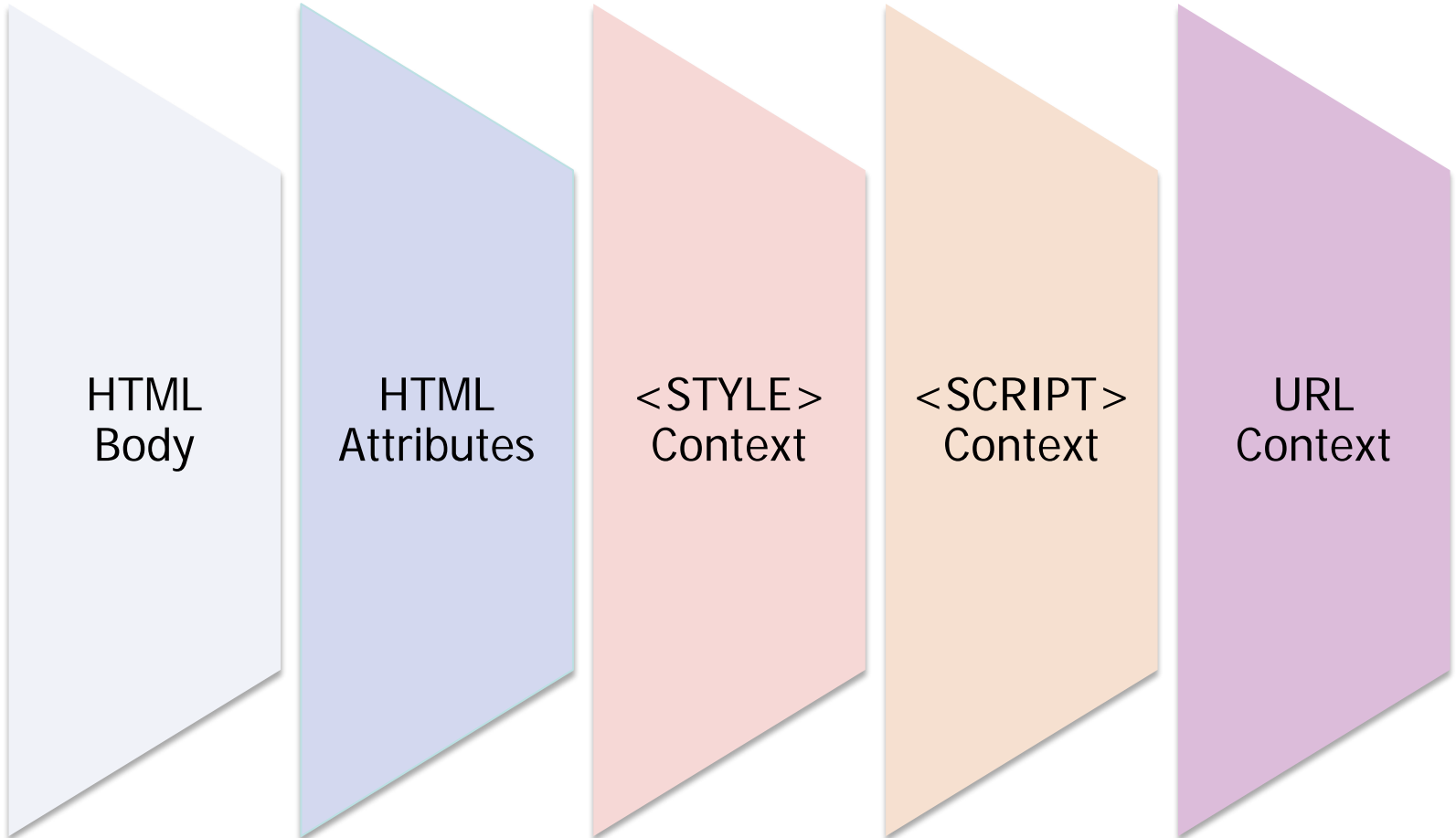
# Output Encoding Code Sample

```
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("&amp;");
    } 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();
```

Simple HTML  
encoding  
method

# Danger: Multiple Contexts

Different encoding and validation techniques needed for different contexts!



# HTML Input

- Clients side widgets like TinyMCE and CKEditor
- Users can edit content beyond plain text
  - Bold, Bullet Points, Color, etc
- These class of widgets submit HTML via request parameters, simple validation not enough
- Validate user-driven HTML on the server with an HTML policy engine
  - Java OWASP AntiSamy or HTML Sanitizer
  - PHP HTML Purifier
  - Java JSoup



# 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><%= 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 Entity Encoding: The Big 6

1.     &     &amp;
2.     <     &lt;
3.     >     &gt;
4.     "     &quot;
5.     '     &#x27;
6.     /     &#x2F;

# 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>`

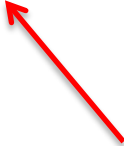
# XSS in Source Attribute

- User input often winds up in src attribute
- Tags such as  
`<img src="">`  
`<iframe src="">`
- 598 Here is your requested image:  
599 <p/>  
600   
601 <p/>
- Example Request:  
`http://example.com/viewImage?imagename=mymap.jpg`
- Attackers can use `javascript:/*attack*/` in src attributes

# XSS Scripting in Script Tag

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 Context

- Escape **all** non alpha-num characters with the \xHH format

```
<script>var x='DATA';</script>
```

- You're now protected from XSS at the time data is assigned
- **What happens to x after you assign it?**

# URL Parameter Escaping

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

```
<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



# XSS in the Style Tag

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

```
169 body {  
170     font-size: 0.8em;  
171     color: black;  
172     font-family: Geneva, Verdana Arial, Helvetica, sans-serif;  
173     background-color: white; ←  
174     margin: 0;  
175     padding: 0;  
176 }  
177
```

URL parameter written within style tag

- 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\28 alert\28 String\2e  
fromCharCode\20 \28 88\2c 88\2c 88\29 \29 \29 ;">  
Mouse over </div>
```

- Pops in at least **IE6** and **IE7**.

[lists.owasp.org/pipermail/owasp-esapi/2009-February/000405.html](http://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 WhiteHat 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)

# Dangerous Contexts

- 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>`

# DOM Based XSS Defense

- DOM Based XSS is a complex risk

- Suppose that x landed in ...

```
<script>setInterval(x);</script>
```

- For some Javascript functions, even JavaScript encoded untrusted data will still execute!

# “Secure” DOM XSS/AJAX Workflow

- Initial loaded page should only be static content
- Load JSON data via AJAX
- 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()).
- 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.toJSON() and JSON.parse()
- Run untrusted script in a sandbox (ECMAScript canopy, HTML 5 frame sandbox, etc)



# Jquery Encoding with JQencoder

- Contextual encoding is the most effective way to combat Cross-Site Scripting (XSS) attacks
- **jqencoder** is a jQuery plugin that allows developers to do contextual encoding to stop DOM-based XSS
  - <http://plugins.jquery.com/plugin-tags/security>

```
<script type="text/javascript">  
    $.post( 'http://site.com/service',  
{ parameter1: 'value' }, function(data) {  
    $('#element').encode('html', cdata);  
    });  
</script>
```

# Jquery Demo

**Sandbox**

**Test HTML Entity Encoding**  
Your payload will be injected directly into the `div` to the right using

```
1 $("#submit-entity-payload").click(function() {  
2   var payload = $("#entity-payload").val();  
3   $("#entity-container").html( $.encoder.encodeForHTML(payload) );  
4 });
```

`<script>alert(1);</script>`

`<script>alert(1);</script>`

Submit Payload

**Test HTML Attribute Encoding**  
Your payload will be injected directly into the value of the `class` attribute of an inner `div` element within the div to the right. This will done using the following code

```
1 $("#submit-attr-payload").click(function() {  
2   var payload = $("#attr-payload").val();  
3   $("#attr-container").html(  
4     <div id="attr-payload-val" class="" +  
5     $.encoder.encodeForHTMLAttribute($.encoder.canonicalize(payload)) + ">" +  
6     $.encoder.encodeForHTML( $.encoder.encodeForHTMLAttribute( payload ) ) +  
7     "</div>" );  
8 });
```

`" onmouseover="alert(1);" onmouseout="`

Submit Payload

# Best Practice: 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

# Best Practice: 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

# JSReg: Protecting JavaScript with JavaScript

- **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

# JSReg: Protecting JavaScript with JavaScript

- **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
- **Content Security Policy**
  - JavaScript policy standard

# 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

# OWASP ESAPI for Java

- ESAPI library provides powerful encoding via `ESAPI.encoder()`:
  - `String encodeForHTML(String input)`
  - `String encodeForHTMLAttribute(String input)`
  - `String encodeForJavaScript(String input)`
  - `String encodeForURL(String input)`
  - `String encodeForCSS(String input)`
  - And more! (LDAP, XML, OS, etc)



# ESAPI Output Encoding

```
<style>
```

```
  bgcolor: <%=ESAPI.encoder().encodeForCSS(data) %>;
```

```
</style>
```

```
Hello, <%=ESAPI.encoder().encodeForHTML(data) %>!
```

```
<script>
```

```
  var uName='<%=ESAPI.encoder().encodeForJavaScript(data) %>';
```

```
</script>
```

```
<div id='<%=ESAPI.encoder().encodeForHTMLAttribute(data) %>'>
```

```
<a href="/mysite.com/editUser.do?userName=<%=  
ESAPI.encoder().encodeForURL(data) %>">Please click me!</a>
```

# XSS Defense Lab

The screenshot shows a Windows Internet Explorer browser window with the address bar displaying `http://127.0.0.1:8080/vuln/XSS.jsp`. The page content includes:

- Generic Search JSP with Echo of user Input**
  - You searched for: null
  - Search for:
  -
- RULE #1 - HTML Escape Before Inserting Untrusted Data into HTML Element Content**
  - `<body>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</body>`
  - `<div>...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...</div>`
  - Submit:
  - 
  - null
- RULE #2 - Attribute Escape Before Inserting Untrusted Data into HTML Common Attributes**
  - `<div attr=...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...>content</div> inside UNquoted attribute`
  - `<div attr='...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE...'>content</div> inside single quoted attribute`
  - `<div attr=" ...ESCAPE UNTRUSTED DATA BEFORE PUTTING HERE ...">content</div> inside double quoted attribute`

The browser's taskbar at the bottom shows the Start button and several open applications, including Internet Explorer, Word, and various folders. The system tray on the right shows the time as 10:28.

# XSS Scripting Summary

- Cross-Site Scripting (XSS)
  - Harmful JavaScript artificially introduced into your web app
- All user input must be validated!
- All user input must be encoded or sanitized specific to each context before being displayed back to the browser!
- Plenty of Web 2.0 vectors to consider such as JSON parsing and DOM XSS

# XSS Scripting Summary

- Consider automatic protection methodologies such as
  - HTTP Only Cookies
  - JavaScript sandboxing
  - Content Security Policy
  - Context-aware auto-escaping templates
  - Object sealing
  - HTML5 Frame sandboxing