

Cross-site Scripting Prevention

CS487

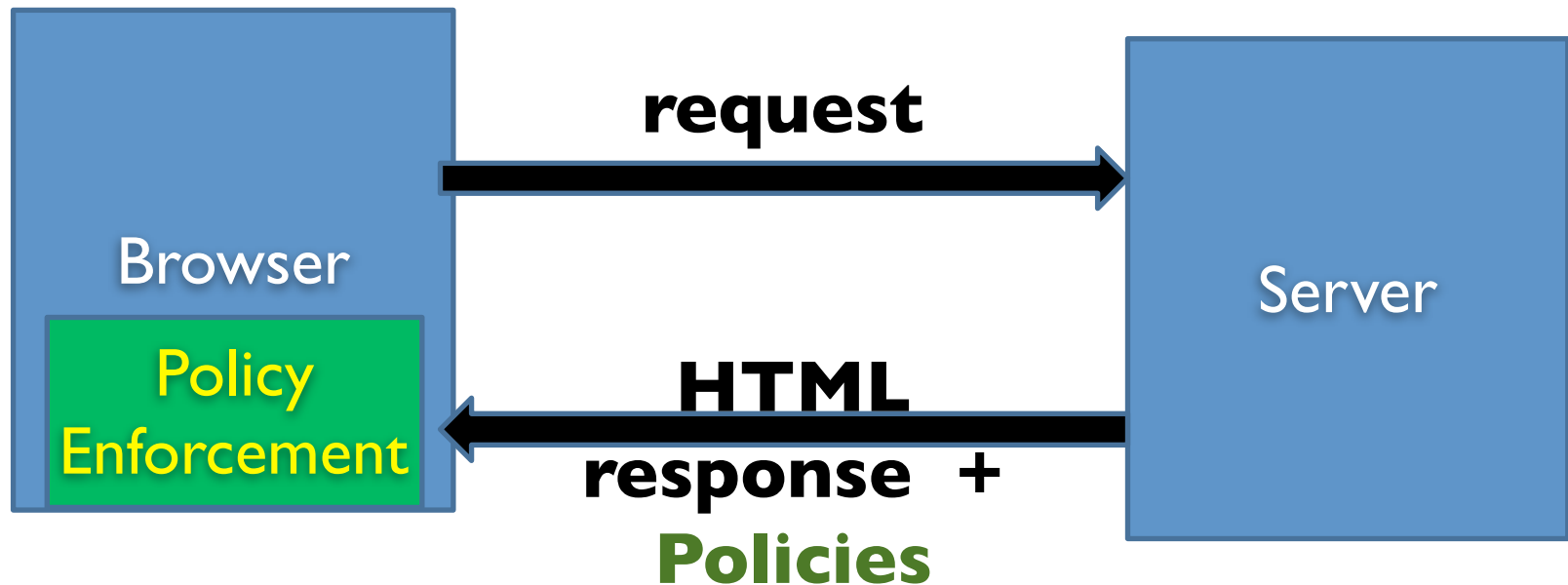
State of the art Research

- **Server-Client Collaborative Protection**
 - Solutions leverage collaboration between Server and Client to prevent attacks.
- **Server Side Protection**
 - Solutions mainly based on “Filtering” hostile contents.
- **Client Side Protection**
 - Solutions based on Firewall like constructions/Information Flow to prevent malicious actions at Client.

Collaborative solution: Browser Enforced Embedded Policies (BEEP)

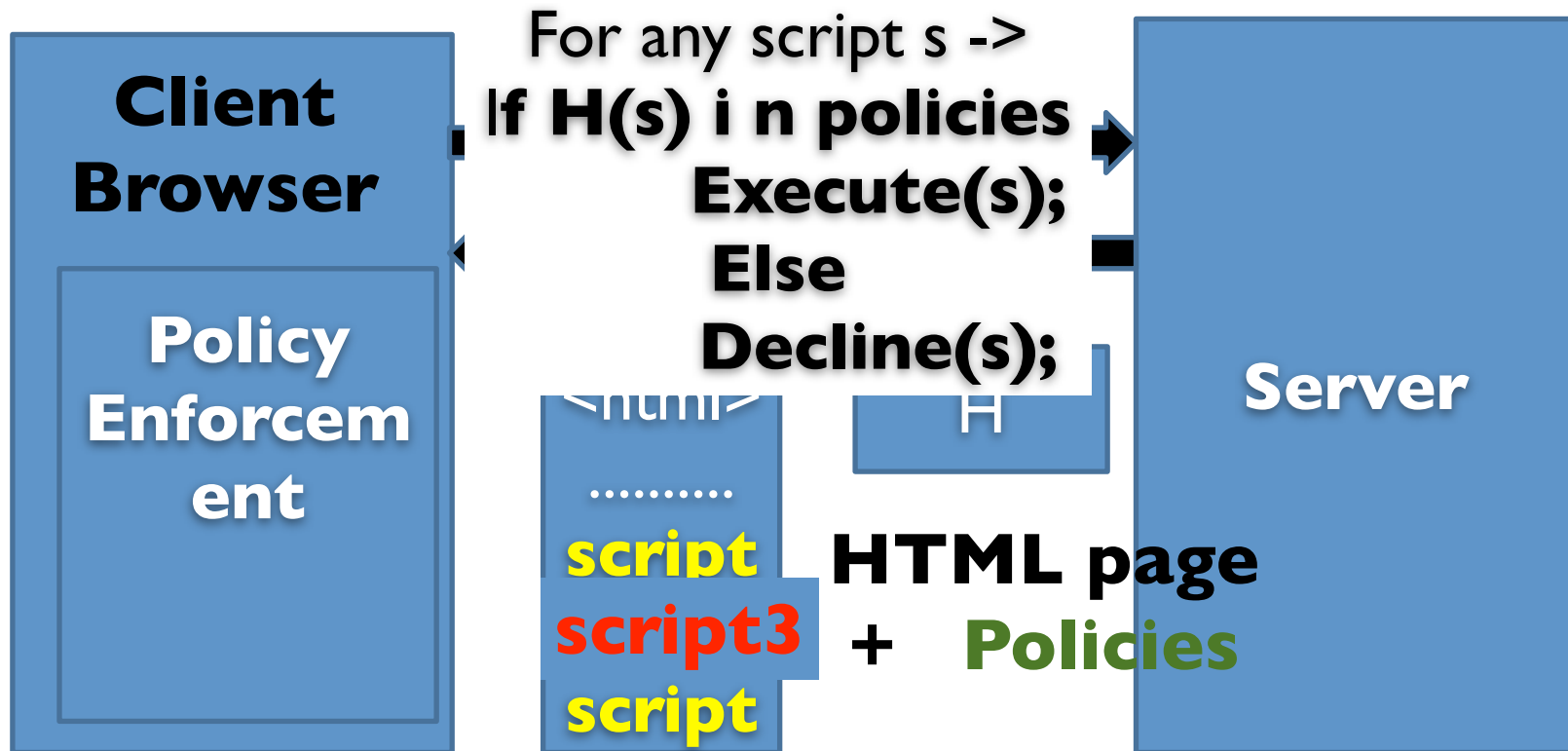
BEEP Solution

- Key Points
 - Application programmer exactly knows which scripts to execute.
 - Client browser exactly knows all “scripts” present in a given page.



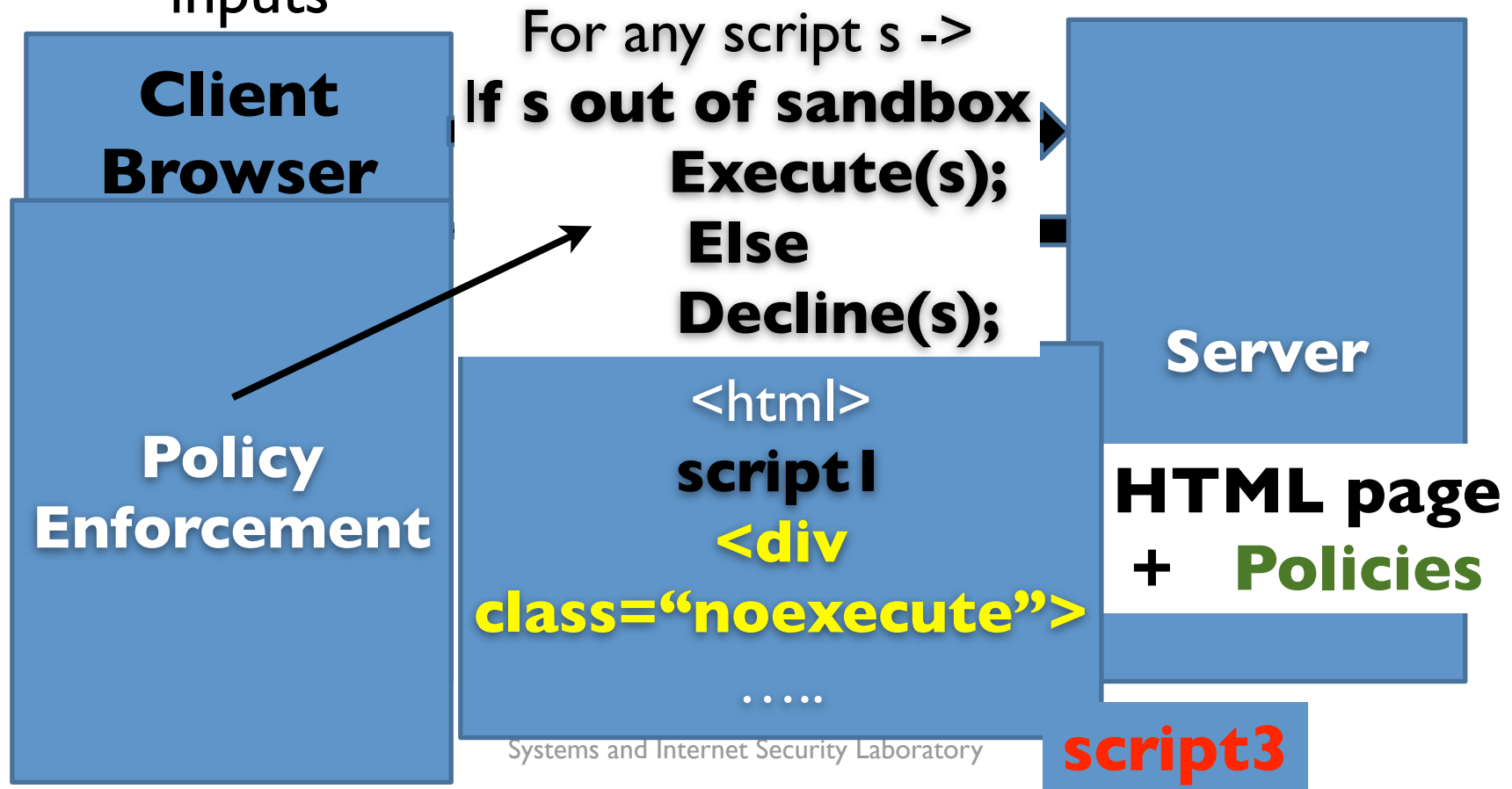
BEEP Policies

- Whitelisting
 - List known scripts and disallow anything else!



BEEP Policies (contd..)

- DOM Sandboxing
 - Sandbox the DOM node which could contain user inputs



BEEP Implementation

- Browser Side Modifications for Policy Enforcement
 - Konquerer and Safari
 - Partial Support for Opera
 - Firefox and IE under investigation
- Web Application Modifications
 - Whitelisting
 - Simple tool to identify scripts, calculate hashes
 - DOM Sandboxing
 - Manually added sandboxing support in applications

Browser server collaborations

- In general, require standards modifications..
why?
- No easy way using the existing standards
to realize BEEP (ref. my paper in W2SP'08)

Issues in realizing BEEP

- Biggest issue: requires browser and standards modifications
 - more of a long term solution
 - near term users remain unprotected
- In general, takes a long time for standards to be changed and browsers to be implemented

Server Side protection

Slides adapted from "Foundations of Security: What Every Programmer Needs To Know" by Neil Daswani, Christoph Kern, and Anita Kesavan (ISBN 1590597842; <http://www.foundationsofsecurity.com>). Except as otherwise noted, the content of this presentation is licensed under the Creative Commons 3.0 License.



Server side protection

- Mostly input filtering
- Identifying characters that appear in scripts and filter them out
 - `< > /`
- Difficult to do if user can input HTML
 - Formatting tags
 - Wikis, blogs
- Solution is to look for input that may result in scripting content

Use of tainting for server side filtering

<HTML>

<Title>Welcome!</Title>

Hi Joe Hacker

taint[33,42]

Welcome to our system

...

</HTML>

How can this be abused??

Preventing XSS

- Never send untrusted data to browser
 - Such that data could cause execution of script
 - Usually can just suppress certain characters
- We show examples of various contexts in HTML document as *template snippets*
 - Variable substitution placeholders: `% (var) s`

General Considerations

- Input Validation vs. Output Sanitization
 - XSS is not just a input validation problem
 - Strings with HTML metachars not a problem until they're displayed on the webpage
 - Might be valid elsewhere, e.g. in a database, and thus not validated later when output to HTML
 - Sanitize: check strings as you insert into HTML doc
- HTML Escaping
 - a.k.a entity reference encoding
 - escape some chars with their literals
 - e.g. & = & < = < > = > " = "
 - Library functions exist

Simple Text

- Most straightforward, common situation
- Example Context:

```
<b>Error: Your query '%(query)s' did not return any results.</b>
```

- Attacker sets `query = <script>evil-script;</script>`
- HTML snippet renders as

```
<b>Error: Your query '<script>evil-script;</script>'  
did not return any results.</b>
```

- Prevention: HTML-escape untrusted data
- Rationale: If not escaped
 - `<script>` tags evaluated, data may not display as intended

Tag Attributes (e.g., Form Field Value Attributes)

- Contexts where data is inserted into tag attribute

- Example HTML Fragment:

```
<form ...><input name="query" value="% (query) s"></form>
```

- Attacker sets `query = cookies"><script>evil-script;</script>`

- Renders as

```
    <form ...>  
    <input name="query" value="cookies">  
    <script>evil-script;</script>">  
    </form>
```

- Attacker able to “close the quote”, insert script

More Attribute Injection Attacks

- **Image Tag:** ``
- **Attacker sets** `image_url = http://www.examplesite.org/onerror=evil-script;`
- **After Substitution:** ``
 - **Lenient browser:** first whitespace ends `src` attribute
 - `onerror` attribute sets handler to be desired script
 - Attacker forces error by supplying URL w/o an image
 - Can similarly use `onload`, `onmouseover` to run scripts

Preventing Attribute Injection Attacks

- HTML-escape untrusted data as usual
 - Escape &, ', ", <, >
- Also attribute values must be enclosed in " "
- Must escape the quote character to prevent “closing the quote” attacks as in example
- Decide on convention: single vs. double quotes

URL Attributes (href and src)

- Dynamic URL attributes vulnerable to injection
- Script/Style Sheet URLs: ``
 - Attacker sets `script_url = http://hackerhome.org/evil.js`
- javascript: URLs - ``
 - By setting `img_url = javascript:evil-script;` we get
``
 - And browser executes script when loading image

Preventing URL Attribute Injection

- Escape attribute values and enclose in " "
 - Follow earlier guidelines for general injection attacks
- Only serve data from servers you control
 - For URLs to 3rd party sites, use absolute HTTP URLs (i.e. starts with http:// or https://)
- Against `javascript`: injection, whitelist for good URLs (apply positive filter)
 - Not enough to just blacklist, too many bad URLs
 - Ex: even escaping colon doesn't prevent script

Style Attributes

- Dangerous if attacker controls style attributes

```
<div style="background: %(color)s;">I like colors.</div>
```

- Attacker injects: `color = green; background-image: url(javascript:evil-script;)`

- Browser evaluates:

```
<div style="background: green;
background-image: url(javascript:evil-script;);">
  I like colors. </div>
```

- In IE 6 (but not Firefox 1.5), script is executed!

- Prevention: whitelist through regular expressions

- Ex: `^([a-z]+)|#[0-9a-f]+$` specifies safe superset of possible color names or hex designation

Within Style Tags

- Injections into `style=` attributes also apply for `<style>` tags
- Validate data by whitelisting before inserting into HTML document `<style>` tag
- Apply same prevention techniques as in earlier.

In JavaScript Context

- Be careful embedding dynamic content

- `<script>` tags or handlers (onclick, onload, ...)

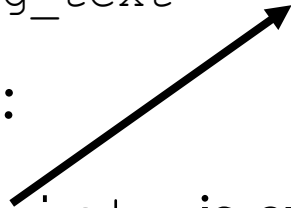
```
<script>
var msg_text = '%(msg_text)s';
// do something with msg_text
</script>
```

- **Attacker injects:**

- **And `evil-script;` is executed!**

```
msg_text = oops'; evil-script; //
```

```
<script>
var msg_text = 'oops';
evil-script; //'
// do something with msg_text
</script>
```



Preventing JavaScript Injection

- Don't insert user-controlled strings into JavaScript contexts
 - `<script>` tags, handler attributes (e.g. `onclick`)
 - w/in code sourced in `<script>` tag or using `eval()`
 - Exceptions: data used to form literal (strings, ints, ...)
 - Enclose strings in `'` & backslash escape (`\n`, `\t`, `\x27`)
 - Format non-strings so that string rep is not malicious
 - Backslash escaping important to prevent “escape from the quote” attack where notions of “inside” and “outside” string literals is reversed
 - Numeric literals ok if from `Integer.toString(), ...`

Another JavaScript Injection Example

- From previous example, if attacker sets

```
msg_text = foo</script><script>evil-script;</script><script>
```

- the following HTML is evaluated:

```
<script>var msg_text = 'foo</script>  
  <script>evil-script;</script>  
<script>'// do something with msg_text</script>
```

- Browser parses document as HTML first
 - Divides into 3 `<script>` tokens before interpreting as JavaScript
 - Thus 1st & 3rd invalid, 2nd executes as `evil-script`

JavaScript-Valued Attributes

- **Handlers inside `onload`, `onclick` attributes:**

- HTML-unesaped before passing to JS interpreter

- **Ex:** `<input ... onclick='GotoUrl("%(targetUrl)s");'>`

- **Attacker injects:** `targetUrl = foo");evil_script("`

- **Browser Loads:**
`<input ... onclick='GotoUrl("foo");evil_script("");'`

- **JavaScript Interpreter gets** `GotoUrl("foo");evil_script("");`

- **Prevention: Two Rounds of Escaping**

- JavaScript escape input string, enclose in `' '`

- HTML escape entire attribute, enclose in `" "`

JavaScript-valued Attributes Prevention Rationale

- HTML-escaping step prevents attacker from sneaking in HTML-encoded characters
- Different quotes
 - Single for JavaScript literals
 - Double for HTML attributes
 - Avoid one type accidentally “ending” the other
- JavaScript escape function should escape HTML metachars (&, <, >, ", ') as well
 - Escaped into hex or unicode
 - Additional security measure if second step forgotten

Redirects, Cookies, and Header Injection

- Need to filter and validate user input inserted into HTTP response headers
- Ex: servlet returns HTTP redirect

```
HTTP/1.1 302 Moved
Content-Type: text/html; charset=ISO-8859-1
Location: %(redir_url)s
```

```
<html>
  <head><title>Moved</title></head>
<body>Moved <a href='%(redir_url)s'>here</a></body>
</html>
```

- **Attacker Injects:** `oops:foo\r\nSet-Cookie: SESSION=13af..3b; domain=mywwwservice.com\r\n\r\n<script>evil()</script>`
(URI-encodes newlines)

Header Injection Example

- Resulting HTTP response:

```
HTTP/1.1 302 Moved
Content-Type: text/html; charset=ISO-8859-1
Location: oops:foo
Set-Cookie: SESSION=13af..3b; domain=mywwwservice.com

<script>evil()</script><html><head><title>Moved</title>
</head><body>
  Moved <a href='oops:foo
Set-Cookie: SESSION=13af..3b; domain=mywwwservice.com
</script>evil()</script>'>here</a></body></html>
```

- Attacker sets desired cookies

Preventing Header Injection

- Ensure URLs for `Location`: headers are well-formed `http:` or `https:`
 - Only consists of characters permitted to be non-escaped according to standard (e.g. RFC 2396)
 - Checks that it's not `javascript:` URL for example
- Check that cookie names and values within standard (e.g. RFC 2965)
- Setting other headers: ensure values contain only characters allowed by HTTP/1.1 protocol spec (RFC 2616)

Filters for “Safe” Subsets of HTML

- Allow “safe” subset of HTML and render to user
- Ex: web-based e-mail app
 - Can allow “harmless” HTML tags (e.g. `<h1>`)
 - But don’t allow execution of malicious scripts
- Use strict HTML parser
 - Strip tags and attributes that are not whitelisted (i.e. known to not allow arbitrary scripting)

Unspecified Charsets, Browser-Side Charset Guessing, and UTF-7 XSS Attacks

- Browser needs to know what character encoding to use to render HTML document
 - Server can specify through `charset` parameter of `Content-Type` HTTP header or `<meta http-equiv>`
 - Default: `iso-8859-1`
 - Or may try to guess the charset
- Example: attacker injects UTF-7 text

```
+ADw-script+AD4-alert(document.domain);+ADw-/script+AD4-
```

- No characters that are normally filtered

Preventing Charset XSS Attacks

- Explicitly specify *appropriate* charset

- Ex: `Content-Type: text/html; charset=UTF-8`

- Or through tag:

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
```

- Meta-tag should appear before untrusted tags

- Appropriate: the one that reflects encoding assumptions used by app for filtering/sanitizing input and HTML encoding output strings

Non-HTML Documents & IE Content-Type Sniffing

- Browsers may ignore MIME type of document
 - Specifying `Content-Type: text/plain` should not interpret HTML tags when rendering
 - But not true for IE: mime-type detection
- AKA Content-Type Sniffing: ignores MIME spec
 - IE scans doc for HTML tags and interprets them
 - Even reinterprets image documents as HTML!

Preventing Content-Type Sniffing XSS Attacks

- Validate that content format matches MIME type
 - Especially for image files: process through library
 - Read image file, convert to bitmap, convert back
 - Don't trust image file format,
- Ensure no HTML tags in first 256 bytes of non-HTML file
 - Could prepend 256 bytes of whitespace
 - Empirically determined # (also in docs), could be different for other versions

Mitigating the Impact of XSS Attacks

- HTTP-Only Cookies: incomplete protection
 - HTTPOnly attribute on cookie in IE prevents it from being exposed to client-side scripts
 - can prevent traditional session hijacking
 - But only for IE and doesn't prevent direct attacks
 - Should also disable TRACE requests
- Binding Session Cookies to IP Address
 - check if session token is being used from multiple IP addresses (especially geographically distant)
 - could cause user inconvenience, use only for high-value

Types of XSS Attacks Recap

Context	Examples (where to inject evil-script)	Prevention Technique
Simple Text	<code>'%(query)'/></code>	HTML Escaping
Tag Attributes (Attribute-Injection)	<code><input ... value = "%(query)"/></code>	HTML Escaping (attrib values in " ")
URL Attributes (href, src attribs.)	<code><script src = "%(script_url)"></code>	HTML Escaping (attrib values in " ")
HTTP Header	<code>HTTP/1.1 302 Moved... Location: %(redirUrl)</code>	Filter Bad URLs (check format)

Types of XSS Attacks Recap

Context	Examples (where to inject evil-script)	Prevention Technique
Style Attributes (or <style> tags)	<pre><div style="background: %(color);"></pre>	Whitelist (Use RegExps)
JavaScript (JS)	<pre><input... onclick=''></pre>	Escape JS/HTML
HTTP Header	<pre>HTTP/1.1 302 Moved... Location: %(redirUrl)</pre>	Filter Bad URLs (check format)

References

1. Defeating Script Injection Attacks with Browser Enforced Embedded Policies: Jim, Swamy and Hicks <http://www2007.org/paper595.php>
2. Technical Explanation of The MySpace Worm <http://namb.la/popular/tech.html>
3. Malicious Yehooligans www.symantec.com/avcenter/reference/malicious.yehooligans.pdf
4. XSS (Cross Site Scripting) Cheat Sheet Esp: for filter evasion <http://ha.ckers.org/xss.html>