Recall that for the purposes of preventing XSRF and XSSI attacks, it is *not* feasible to rely on `Referer` headers because they are not always present, and at the same time can in certain circumstances be suppressed by a malicious site (see Section 10.3). However, for limiting "bandwidth leeching," it is sufficient. It is generally not possible for a page to include an image, script, or style sheet, and suppress the `Referer` in the resulting request; at the same time, most users' browsers will send `Referer` headers. A third party that relies on our resource would only "work" for users whose browsers or proxies suppress `Referer` headers, which makes it impractical for them to do so. And even if a few requests slip through, we are only exposed to a small fraction of the cost we could otherwise expect.

10.5. Preventing XSS

At a first glance, preventing XSS appears fairly straightforward. We have to ensure that our application never sends any untrusted data to the user's browser such that this data could cause execution of script in the browser. Usually, this can be done by either suppressing certain characters (such as the characters < and > that delimit HTML tags), or replacing them with an appropriate escape sequence (such as `<` and `>`).

Unfortunately, the solution is not as simple as just escaping <, >, and "; there are a multitude of situations and contexts in which a string could be interpreted by the browser in a way that causes execution of script. In the following, we examine each such context in turn, and then also cover a number of special situations.

Most situations in which a particular string could cause script execution involve placement into a specific context within an HTML document. We give examples for these contexts in the form of HTML snippets with variable substitution placeholders such as `%(variable)s` (this syntax will be familiar to Python programmers). We refer to HTML snippets containing such placeholders as HTML *template snippets*.

For example, the HTML template snippet

```
<title>Example document: %(title)s</title>
```

is intended to illustrate a template snippet that results in the following HTML to be emitted to the browser if the variable `title` has value `Cross-Site Scripting`:

```
<title>Example document: Cross-Site Scripting</title>
```

We chose this syntax to keep the examples independent from any particular templating or HTML rendering infrastructure; the equivalent notation for the preceding would be `<?= title ?>` in PHP and `<%=title %>` in ASP or JSP.

The examples in this document are based on JavaScript. Of course, similar examples could be constructed using other scripting languages supported by the victim's browser (such as VBScript).

To keep the examples short, we use the placeholder `evil-script;` to denote a sequence of JavaScript statements the attacker might inject (possible exploit payloads were discussed in more detail in Section 10.2.3).

For each class of XSS vulnerability (which is based on the context within an HTML document in which the injection occurs, such as "simple text," "within an `href` attribute," etc.), we provide the following:

- An example that shows how the injection can be exploited—that is, how the attacker could inject strings into the HTML document such that script of his choosing would execute in the user's browser
- Prevention techniques for avoiding XSS in each particular context (e.g., "escape a specific set of characters")
- Rationale explaining why these guidelines are necessary and why they indeed prevent XSS in the given context

10.5.1. General Considerations

Before we discuss prevention of script injection in specific HTML contexts, let's make some general observations.

Input Validation vs. Output Sanitization

It is sometimes suggested that XSS is an issue that occurs due to lack of proper input validation.

Strictly speaking, this is not so, and relying on input validation alone can result in incomplete solutions. There are a number of reasons that support this observation:

- It is often not possible to restrict input strings to a set of characters that are safe to output in all HTML contexts. For example, we may have to allow angle-bracket and double-quote characters for an input field representing an e-mail address to allow addresses of the form `"Alice User" <alice@learnsecurity.com>`.
- Input validation is usually applied at the outer boundaries of the overall system—for example, to query parameters received with HTTP requests, or data retrieved from a back-end data feed. However, back-end applications and databases are usually considered "inside" the input validation boundary.

  As such, it is common that data read from a database is not passed through an input validation layer. At the same time, another application component may have very reasonably decided that writing a string containing HTML metacharacters to the database is safe; the developer of that component may not be aware that the string is later read back from the database and displayed to the user as part of an HTML document.
- At a very basic level, we make the observation that no risk arises from strings containing HTML (and potentially JavaScript) while they are being passed around within a web application, written to databases, used within database queries, and so forth. Problems only arise when the string is sent to a user's browser and interpreted as HTML.

  It is in many cases not architecturally reasonable to enforce that all strings be HTML-safe (free of HTML metacharacters or suitably escaped) throughout the system.

As such, we often cannot avoid addressing the problem by sanitizing strings at the time they are inserted into an HTML document, i.e., by performing *output sanitization*.

This, of course, does not imply that you should be lenient with respect to input validation! You must always apply the strongest possible input validation constraints within the parameters given by your feature specifications. Strict input validation will often prevent a missing

output sanitization step from resulting in an exploitable XSS vulnerability (and will of course also reduce the risk of other classes of vulnerabilities that can be exploited based on untrusted input).

HTML Escaping

In many of the contexts covered in the following sections, XSS is prevented by escaping certain metacharacters such that strings are treated as uninterpreted literals, rather than HTML markup.

In HTML, escaping is accomplished by replacing characters with their corresponding *HTML character reference* (see the HTML 4.01 Specification, Section 5.3, for more information). Character references can be numeric in the form `&#`*D*`;` where *D* is the decimal character number, or `&#x`*H*`;` where *H* is the character number in hexadecimal notation. For example, the character reference `&#x41;` represents the letter *A*. Alternatively, so-called *entity references* are available, which refer to the character with a mnemonic name. For example, the entity reference `<` refers to the less-than symbol, <.

We assume in the following discussion that an HTML escape function is available that escapes at least the characters listed in Table 10-1 into their corresponding HTML character references.

Table 10-1. *Minimum Set of Characters to Be Escaped by HTML Escape Function*

Character	Reference
&	`&`
<	`<`
>	`>`
"	`&quot;`
'	`&#39;`

Ready-to-use HTML escape functions are available in libraries in many programming languages. It is advisable to carefully check the documentation or source code (if available) that all necessary characters are indeed escaped. For example, it is easily overlooked that the Python function `cgi.escape` (available in the library included in the standard Python distribution) does not escape quote characters unless requested via an optional argument, and even then only escapes double-quote characters.

In certain contexts, we require other escaping functions (e.g., for JavaScript string literals); we introduce these functions in the appropriate context.

10.5.2. Simple Text

This is the most straightforward and common situation in which XSS can occur.

Example

Suppose our application is producing output based on the following template fragment:

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

If the attacker is able to cause the variable `query` to contain, for example

```
<script>evil-script;</script>
```

the resulting HTML snippet would render as

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

and the attacker's script would execute in the browser, and could, for example, steal the victim user's cookies.

Prevention Techniques

Any string that is possibly derived from untrusted data and is inserted into an HTML document must be HTML-escaped using the HTML escape function introduced in Section 10.5.1.

Rationale

The less-than and greater-than characters need to be escaped because they delimit HTML tags. If not escaped, these tags (including `<script>` tags) would be evaluated by the browser.

If the ampersand were not escaped in this context, this would not result in a security issue, but could result in a rendering bug because the browser may interpret the ampersand as the beginning of an entity reference and not display it as intended.

It is not strictly necessary to escape the quote characters in this context; however, this is necessary in other contexts, and it is convenient to use the same escaping function everywhere.

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

Many HTML tags can have (usually optional) attributes that specify or modify how a tag is interpreted by a browser. For example, in the HTML snippet

```
<form method="POST" action="/do">
```

the `<form>` tag has two attributes, named `method` and `action`. The value of the attribute `method` is the string `POST`, and the value of the attribute `action` is the string `/do`.

This and the following subsection cover several variations of contexts in which data is inserted into the values of attributes of HTML tags. In *this* section, we discuss concerns that apply to all attributes. The examples consider a form field that is prefilled with data. However, the considerations in this section apply to other attributes as well (such as `style`, `color`, `href`, etc).

Example

Consider a template fragment of the form

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

If an attacker is able to cause the variable `query` to contain, for example

```
cookies"><script>evil-script;</script>
```

then, after substitution, this will result in the HTML

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

That is, the attacker is able to "close the quote" and insert a script tag that will be executed by the browser.

Attribute-Injection Attacks

A variation of the attack in the previous section is possible if the attribute's value is not enclosed in quotes in the template. Consider a template fragment in which the attribute's value is not enclosed in quotes, for example

```
<img src=%(image_url)s>
```

Suppose the attacker is able to cause the variable `image_url` to contain

```
http://www.examplesite.org/ onerror=evil-script;
```

After substitution, this will result in the HTML fragment

```
<img src=http://www.examplesite.org/ onerror=evil-script;>
```

Browsers are usually lenient in their parsing of HTML attributes, and assume that an attribute whose value is not enclosed in quotes ends at the first whitespace character or the end of the tag. Thus, the preceding HTML will be parsed as an `<img>` tag with *two* attributes (i.e., the attacker was able to inject an additional attribute).

The ability to inject an arbitrary attribute can often be exploited to execute arbitrary script. In the preceding example, the attacker arranged to inject an `onerror` attribute, which specifies an error handler in the form of a JavaScript snippet that the browser evaluates if evaluation of the tag resulted in an error condition. In the example, the attacker forces the error condition by supplying a URL that does not resolve into an image document (i.e., the URL can be a valid, resolvable URL that returns an HTML or other non-image document).

Besides the `onerror` handler, other handler attributes, such as `onload`, or handlers for various DOM events, such as `onmouseover`, may be usable in an exploit (though the latter usually requires user interaction to be triggered).

It should be noted that this attribute-injection attack did not require the injection of any HTML metacharacters (angle brackets or quotes) that would be commonly escaped or filtered. We also note that it is quite possible to craft malicious script payloads without using quote characters (it may be tempting to assume that it is difficult for an attacker to do anything damaging without being able to specify string constants—for instance, to refer to their server's URL).

Prevention Techniques

Any string that is possibly derived from untrusted data and is inserted into the value of an HTML tag's attribute must be HTML-escaped using the HTML escape function introduced in Section 10.5.1.

Furthermore, the attribute's value must be enclosed in *double* quotes.

Rationale

The entire attribute value must be enclosed in quotes to prevent attribute-injection attacks.

First, it is necessary to escape the quote character that is used to delimit the attribute's value to prevent the "closing the quote" attack. While the HTML specification allows either double or single quotes to be used to enclose attributes, it is advisable to decide on a convention and use one type of quote throughout the application. It is nevertheless advisable to use an HTML escaping function that escapes both types of quotes, in case of deviation from the convention.

Second, it is necessary to escape the ampersand character. Older versions of the Netscape browser support so-called *JavaScript entities* (see Netscape's "JavaScript Guide"). This allows a string of the form `&{javascript_expression};` to be used within attributes; the expression is evaluated and the entire entity expression is replaced with the result of this evaluation. An attacker who is able to inject ampersand and curly-brace characters into an attribute could be able to execute malicious script.

While non-escaped angle brackets in attribute values do not result in XSS vulnerabilities in popular browsers, it is safest to escape them nevertheless. This also ensures that the resulting HTML is well-formed and allows you to use the same HTML-escaping function as elsewhere.

10.5.4. URL Attributes (href and src)

Attributes such as `href` and `src` take URLs as arguments. Depending on the tag they are associated with, the URL may be interpreted, de-referenced, or loaded at the time the browser interprets the tag (e.g., `<img src=...>` tags), or loaded only when the user performs an action (e.g., `<a href=...>` tags).

If the value of the URL attribute is computed dynamically and may be influenced by a attacker, the attacker can make the URL refer to a resource that we did not intend. This could result in all kinds of problems (e.g., page spoofing), but may in particular result in injection of malicious script.

Script and Style Sheet URLs

The attacker can easily cause script to execute if he can manipulate the source of a `<script>` or `<style>` tag (as we will discuss in Section 10.5.5, CSS style sheets can cause script to execute). Take the following example:

```
<script src="%(script_url)s">
```

If the attacker can make `script_url` point to `http://hackerhome.org/evil.js`, his malicious script will execute in the context of the page containing this script tag.

javascript: URLs

Furthermore, most browsers interpret URLs with the `javascript:` scheme such that the remainder of the URL is evaluated as a JavaScript expression, as in the following example:

```
<img src="%(img_url)s">
```

If the attacker can set `img_url` to `javascript:evil-script;`, the resulting HTML will be

```
<img src="javascript:evil-script;">
```

When the browser attempts to load the image, `evil-script;` will execute.

Other Resource URLs

Other resources can be very dangerous (e.g., code bases for ActiveX objects or Java applets), or at least highly annoying (e.g., background music) if their URLs can be influenced by an untrusted source. They need to be validated carefully, and should generally not be allowed to refer to a URL that is not under your control.

Prevention Techniques

The attribute's value must be escaped and enclosed in quotes, as described in Section 10.5.3.

For script, style sheet, and code base URLs, it must be enforced that the data is served from a web server under our control. If such URLs are allowed to be absolute paths without a host part (e.g., `src="/path/file"`), then it is important that the attacker does not have control over the `BASE` attribute of the page, if any. For URLs (e.g., image URLs and the like) that may refer to third-party sites, it must be verified that the URL is a relative or absolute HTTP URL (i.e., one that starts with `/`, `http://`, or `https://`).

Rationale

The escaping is necessary to prevent the general attribute-based script-injection attacks described in the previous section. Enclosing the URL in quotes is necessary to prevent attribute-injection attacks (see Section 10.5.3). We note that it may be safe to omit the quotes around the attribute's value if you are sure that the URL does not contain any whitespace characters. However, we advise against such cutting of corners; the savings of two characters of page size per attribute rarely justifies the additional risk due to the special casing in code and/or coding conventions.

To prevent `javascript:` injection attacks, one might be tempted to just disallow URLs that start with `javascript:`. However, there are many rather obscure variations of this injection, and it is much safer to apply a positive filter. For example, some versions of Internet Explorer will ignore a `0x08` (`\010` octal) character at the beginning of the string. Similarly, if the `:` character is escaped as an HTML character reference—such as `javascript&#58;evil-script;`—both Internet Explorer and Firefox still execute the script. Internet Explorer executes `vbscript:` URLs. The `data:` scheme can also be used to cause script execution—for example, `data:text/html,<script>evil-script;</script>`. It is quite possible (or rather, likely) that there are additional browser behaviors that similarly can be used to cause script execution. This example vividly illustrates the virtues of the general security paradigm of preferring whitelisting over blacklisting. Validating a parameter with anything but the most trivial semantics is almost

always safer by testing inclusion in a known-safe subset of possible values, rather than trying to exclude the set of values you think are problematic—it is often very difficult to reliably characterize the set of "bad values."

10.5.5. Style Attributes

Style attributes can be dangerous if an attacker can control the value of the attribute, since CSS styles can cause script to execute in various ways.

Example

For example, consider the following template fragment:

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

If the attacker can cause `color` to contain

```
green; background-image: url(javascript:evil-script;)
```

after substitution, the HTML evaluated by the browser would be

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

This does result in `evil-script;` being executed (at least in Internet Explorer 6.0; Firefox version 1.5, for example, apparently does not de-reference `javascript:` URLs in this context).

Prevention Techniques

It is very important to validate the value that is to be inserted into the style attribute using a whitelist approach (i.e., we must test that the string in question is in a set of strings that are sure to be safe in the given context).

Regular expressions provide a convenient and efficient mechanism for the specification of sets of strings (and testing the inclusion therein).

For example, we can specify the set of strings consisting of (a safe superset of) syntactically valid CSS color specifications with the following regular expression:

```
^([a-z]+)|(#[0-9a-f]+)$
```

This regular expression defines the set of strings that consist of either a sequence of one or more lowercase letters (a "color name"), or the symbol # followed by one or more hexadecimal digits (a numeric color specification). The ^ and $ regular expression operators force a match at the beginning and end of the string. Failing to require a match of the entire string is a common mistake when using regular expressions for data validation.

We note that the preceding regular expression does not match the exact set of strings that are valid CSS color specifications, but rather a superset thereof. For example, the strings `brfff` and `#7` are matched, but represent a nonexistent color name and a numeric color specification

with insufficient digits. However, it is reasonable to assume that providing such strings to the browser as a CSS color specification will not result in JavaScript execution.

An even safer alternative would be to not derive the value to be inserted into the HTML document directly from user-controlled input, but rather externally expose a different parameter that is mapped to a fixed set of values. For example, we could externally expose an integer-valued parameter, `color_id`, and look up the corresponding CSS color specifier in a table (of course, not forgetting to bounds-check the index).

As with all attributes, it is important to ensure that style attributes do not contain any (non-escaped) quote characters, and that their value is enclosed in quotes in the template.

Rationale

Using a whitelist approach is very important. CSS is a fairly complex language and there are several ways in which a style specification could cause JavaScript to execute. It would be fairly difficult to reliably filter out malicious style specifications using a blacklist approach.

10.5.6. Within Style Tags

The previous section's considerations regarding `style=` attributes also apply to `<style>` tags. Data must be validated very carefully using a whitelist approach before it is inserted into an HTML document in a `<style>` context.

10.5.7. In JavaScript Context

For obvious reasons, one has to be very careful with regard to embedding dynamic content in `<script>` tags or other contexts that are evaluated as script (such as `onclick`, `onload`, and `onerror` handler attributes). If an attacker can cause arbitrary strings to be injected into a JavaScript context within a document in our application's domain, he can very likely cause malicious script to execute. Note that HTML-escaping the data is not sufficient, since the attacker does not need to inject any HTML tags.

Dynamic content within `<script>` tags should generally be avoided as much as possible, with the exception of situations in which *data* is emitted to a client in JavaScript syntax. For example, it is sometimes useful to initialize variables with dynamically computed values in the context of a `<script>` tag.

In Ajax-style web applications (see 10.2.2), it is common for the server to return documents containing data in JavaScript syntax—for example, in the form of arrays or object literals. When writing the server-side code of an Ajax application, you have to be careful about how you control untrusted data that will appear within a JavaScript context in the user's browser.

Example

For example, consider the following template fragment:

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

If the attacker can cause `msg_text` to contain

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

after substitution, the HTML evaluated by the browser would be

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

which would cause `evil-script;` to execute.

Prevention Techniques

Do not insert user-controllable strings into contexts that will be evaluated as JavaScript, including the following:

- Within `<script>` tags in HTML documents
- Within handler attributes such as `onclick`
- Within JavaScript code intended to be sourced by a `<script>` tag or evaluated using `eval()`

Exceptions to this rule are situations in which data is used to form literals of elementary JavaScript data types such as strings, integers, and floating-point numbers.

For string literals, it is necessary to enclose the string with single quotes, and to ensure that the string itself is JavaScript string-escaped, as shown in Table 10-2 (we use the *U+<hex-digits>* notation to refer to nonprintable Unicode code points). In addition, it is advisable to escape all characters less than 32 and greater than 127, especially if the document encoding of the resulting document cannot be relied upon to match the one used during processing of strings.

Table 10-2. *Character Escapes for JavaScript String Literals*

Character	Escape	Comment
U+0009	\t	Tab
U+000a	\n	Line feed
U+000d	\r	Carriage return
U+0085	\u0085	Next line
U+2028	\u2028	Line separator
U+2029	\u2029	Paragraph separator
'	\x27 or \u0027	Single quote
"	\x22 or \u0022	Double quote
\	\\	Backslash
&	\x26 or \u0026	Ampersand

Continued

Table 10-2. *Continued*

Character	Escape	Comment
<	\x3c or \u003c	Less than
>	\x3e or \u003e	Greater than
=	\x3d or \u003d	Equals

Ensure that the string literal is not later used in a context in which it could be interpreted as script (such as a JavaScript `eval()`).

Non-string literals (such as integers and floats) need to be formatted appropriately to ensure that the resulting string representation cannot result in malicious JavaScript.

Rationale

Embedding JavaScript statements that are dynamically derived from user input into `<script>` tags would be extremely risky; it is essentially impossible to reliably distinguish harmless snippets of code from dangerous ones.

Enclosing in quotes and backslash-escaping the inserted string ensures that the JavaScript parser interprets the string as a single string literal as intended. We must escape quotes and line feed characters because they could be interpreted as the end of the string literal and permit an "escape from the quote" attack. We also must escape the backslash; otherwise the attacker could provide a single backslash, which would escape the quote that was intended to end the string literal. After that, the sense of "inside" and "outside" string literals is reversed, and the attacker may be able to cause script execution if he controls another string that is inserted later on.

The escaping of the angle bracket characters is necessary because otherwise an attacker could cause arbitrary script execution by injecting the following (into the `msg_text` variable in the example at the beginning of this section):

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

After substitution, the HTML evaluated by the browser would be the following (the extra new lines were inserted for formatting reasons):

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

Somewhat surprisingly, this HTML document does in fact result in the execution of `evil-script;`. The reason is that the browser first parses the document as HTML, and only later passes text enclosed in `<script>` tags to the JavaScript interpreter—in other words, the HTML parser does not respect or care about the delimiters of JavaScript string literals. Thus, the HTML fragment will be parsed into three separate `<script>` tags. The first script tag contains invalid JavaScript and would result in a syntax error. However, most browsers will

evaluate separate `<script>` tags separately, and indeed execute the second (syntactically correct) tag containing the malicious script (the third tag will again result in an error).

Finally, we escape the `=` and `&` characters for defense-in-depth, preventing attacker-provided strings from being interpreted as tag attributes or HTML entities, respectively.

Numeric literals are generally safe if their string representations were obtained by the appropriate conversion from a native numeric data type (e.g., via `snprintf("%d",...)` or `Integer.toString()`). Note that in weakly typed languages such as Perl, PHP, and Python, it is important to enforce type conversion to the appropriate numeric type.

10.5.8. JavaScript-Valued Attributes

In addition to the considerations in the previous section, we have to keep in mind an additional complication that arises with JavaScript in the context of a JavaScript-valued tag attribute such as an `onLoad` or `onClick` handler: the values of such attributes are HTML-unescaped before they are passed to the JavaScript interpreter.

Example

For example, consider the following template fragment:

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

Suppose an attacker injects the value

```
foo&amp;quot;);evil_script(&amp;quot;
```

for `targetUrl`, and our application does not apply any escaping of HTML metacharacters to this variable. Note that this situation might appear perfectly safe because the string contains neither non-escaped JavaScript quote characters nor HTML metacharacters.

However, this scenario results in the following document to be evaluated by the browser:

```
<input ...
       onclick='GotoUrl("foo&amp;quot;);evil_script(&amp;quot;");'>
```

The browser HTML-unescapes the value of the `onclick` attribute before passing it to the JavaScript interpreter, which then evaluates the expression

```
GotoUrl("foo");evil_script("");
```

Thus, the JavaScript interpreter will in fact invoke `evil-script`.

Prevention Techniques

When inserting user-controllable strings into the context of an attribute that is interpreted as a JavaScript expression (such as `onLoad` or `onClick`), ensure that the string literal is JavaScript escaped using a function that satisfies the criteria defined in the previous section, and is enclosed in *single* quotes. Then HTML-escape the entire attribute, and ensure that the attribute is enclosed in *double* quotes.

As an additional safety measure, JavaScript escape functions should escape the HTML metacharacters `&`, `<`, `>`, `"`, and `'` into the corresponding hexadecimal or Unicode JavaScript character escapes (see Table 10-2).

Rationale

The additional HTML-escaping step ensures that the JavaScript expression passed to the JavaScript interpreter is as intended, and an attacker cannot sneak in HTML-encoded characters.

Using different style quotes for JavaScript literals and attributes provides a safety measure against one type of quote accidentally "ending" the other.

The use of a JavaScript-escaping function that escapes HTML metacharacters into numeric JavaScript string-escapes provides an additional safety measure in cases in which a programmer forgets to use both escapes in sequence and only uses the JavaScript escape. Note that for this measure to be effective, it is important to use the numeric escape for quote characters (i.e., `\x22` instead of `\"`) since the HTML parser does not consider the backslash an escape character, and therefore a non-HTML-escaped `\"` would actually end the attribute.

10.5.9. Redirects, Cookies, and Header Injection

Problems can also arise if user-derived input is not properly validated or filtered before it is inserted into HTTP response headers.

Example

Consider a servlet that returns an HTTP redirect and allows the attacker to control the variable `redir_url` via a request parameter:

```
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>
```

Suppose an attacker is able to set the redirect URL to the string

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

Note that `\n` and `\r` denote newline and carriage return characters, respectively; and that the string has been wrapped to fit the page. The attacker would likely submit the newline characters in URI-encoded form—that is, `'oops:foo%0d%0aSet-Cookie...'`.

The resulting HTTP response would be as follows:

```
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

&lt;script&gt;evil()&lt;/script&gt;'>here</a>
```

This will cause the cookie of the attacker's choosing to be set in the user's browser, and may also execute the malicious script (e.g., Firefox would determine that the `Location:` header of this HTTP response is not valid and would then just render the HTML in the response's body).

A similar scenario could occur for servlets that emit `Set-Cookie` headers and derive the cookie's name or value from user input.

Aside from the potential for XSS, the ability for an attacker to influence the user's cookies can under certain circumstances be problematic in itself. For example, the attacker might be able to overwrite cookies that embody user preferences, which is a (albeit in most cases minor) DoS issue. Or, the attacker may be able to set a cookie that is used to protect the application against XSRF attacks (see Section 10.3.3) to a known value, which might permit the attacker to circumvent the protection.

Prevention Techniques

When setting `Location:` headers, ensure that the URL supplied is indeed a well-formed `http` URL. In particular, if it starts with a scheme (*`xxxx:`*), the scheme must be `http` or `https`. Furthermore, it must consist only of characters that are permitted to occur non-escaped in a URL as specified in the relevant standard, RFC 2396 (Berners-Lee, Fielding, and Masinter 2005).

When setting cookies, ensure that the cookies' names and values contain only characters allowed by the relevant standard, RFC 2965 (Kristol and Montulli 2000).

When setting other headers (e.g., `X-Mycustomheader:`) ensure that the header values (as well as header names) contain only characters allowed by the HTTP/1.1 protocol specification in RFC 2616 (Fielding et al. 1999).

Rationale

Restricting character sets to the characters allowed by the various specifications ensures that the HTTP response will be parsed by the browser correctly and as intended.

Validating the URL ensures that only redirects to valid HTTP URLs can occur—not to, for instance, a `javascript:` URL. While modern browsers will not execute script in a redirect to a `javascript:` URL, older browsers might. As always, we follow the "whitelist, not blacklist" paradigm.

10.5.10. Filters for "Safe" Subsets of HTML

There are situations in which some "safe" subset of HTML should be allowed past filters and rendered to the user. An example would be a web-based e-mail application that allows "harmless" HTML tags (such as `<h1>`) in HTML e-mails to be rendered to the user, but does not allow the execution of malicious script contained in an e-mail.

The general recommended approach to this problem is to parse the HTML with a strict parser, and completely remove all tags and attributes that are not on a whitelist of tags and attributes that are known to not allow arbitrary script execution.

Getting this right is fairly difficult; we highly recommend that designers and developers of such applications consult a security expert versed in web application and cross-domain security issues.

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

To render a document received from a web server, a browser must know what character encoding to assume when interpreting the raw stream of octets received from the server as a sequence of characters. For HTML documents, a server can specify the encoding via the `charset` parameter of the `Content-Type` HTTP header, or in a corresponding `<meta http-equiv>` tag (the terminology around charsets, document character sets, and character encodings is somewhat confusing—see the HTML 4.01 Specification, Section 5.2).

If no charset is specified by the server, browsers generally assume a default—for example, `iso-8859-1`. In addition, some browsers can be configured to guess the correct charset to use for a given document.

The latter behavior can lead to XSS vulnerabilities, because character sequences that were interpreted in a certain way under an assumed charset on the server (and, in particular, not escaped or filtered) can be interpreted as different character sequences under a different, guessed encoding in the browser.

Example

For example, suppose a server renders an HTML document based on the following template, and the document is returned without an explicitly specified `charset`:

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

Suppose an attacker can cause `query` to contain

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

Note that this string does not contain any characters that would usually be filtered out by an input filtering framework. Neither would any of the characters be escaped by an application that follows the usual guidelines for HTML-escaping strings documented in Section 10.5.2.

The resulting HTML snippet would render as

```
<p>Error: Your query
   '+ADw-script+AD4-alert(document.domain);+ADw-/script+AD4-'
   did not return any results.</p>
```

If the user is using Internet Explorer (as of version 6.0) configured to auto-select encodings (set in menu View ➤ Encoding ➤ Auto-Select), Internet Explorer will guess UTF-7 as the encoding for this document (Firefox appears not to guess UTF-7 encodings, even with autodetect enabled). However, under UTF-7 encoding, the octet sequence corresponding to the ASCII characters `+ADw-` is actually an encoding of the less-than character (i.e., `<`), and `+AD4-` corresponds to the greater-than character (`>`). Therefore, the browser will interpret and execute the script tag.

Prevention Techniques

All pages rendered by a web application must have an appropriate charset explicitly specified, which can be done using one of two mechanisms:

- Via the `charset` parameter of the HTTP `Content-Type` header—for example

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

- Via a corresponding `<meta http-equiv>` tag:

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

  When using this solution, it is important to ensure that the `<meta>` tag appears in the document *before* any tag that could contain untrusted data—for example, a `<title>` tag.

  It is important to specify an *appropriate* charset that reflects the encoding assumptions made by the application when sanitizing/filtering inputs and HTML-encoding strings for output.

10.5.12. Non-HTML Documents and Internet Explorer Content-Type Sniffing

In general, browsers are expected to honor the MIME type of the document as specified in the `Content-Type` HTTP header. In particular, one would expect that a browser would always render a document with `Content-Type: text/plain` as plain text, without interpreting HTML tags in the document.

Content-Type Sniffing

This is not the case for Internet Explorer, which has a feature, referred to as *content-type sniffing* or *MIME-type detection*, where it scans the beginning of a document for HTML tags. If it finds substrings that appear to be HTML tags, it assumes that the publisher of the document meant to serve an HTML document, and ignores the document's specified content type and interprets the document as HTML. (For more information, see the Microsoft MSDN article, "MIME Type Detection in Internet Explorer," at `http://msdn.microsoft.com/workshop/networking/moniker/overview/appendix_a.asp`).

This can result in considerable headaches for the developer of an application that renders non-HTML documents from untrusted sources. For example, a web-based e-mail application may have a feature that allows users to view an e-mail's attachments in a separate browser window. A malicious attachment of content type `text/plain` that contains script tags near the beginning would cause the malicious script to be executed if viewed by a user in Internet Explorer, even if the server served the document with the correct `Content-Type: text/plain` header.

More problematically, some versions of Internet Explorer even do this for documents with image content types. If an image is not really a valid image file, but rather contains HTML tags near the beginning, Internet Explorer will reinterpret the image document as HTML, and execute any script incorporated in the document. Note that it does so only if the image is

accessed as an entire separate document (i.e., if the browser accesses a link that leads to this document), but apparently not (at least in current versions) when the image is embedded into another document via an `<img>` tag. Applications that let users upload images that may be displayed to other users have to worry about this.

Prevention Techniques

With respect to XSS, there are a number of ways to deal with this feature:

- Validate the content to be displayed to indeed be a document of the intended MIME type. This strategy would be most appropriate for images. To be on the safe side, it would be best to actually process the image using an image-manipulation library; that is, read the image file, convert it to a bitmap, and then convert it back to an image file in the appropriate format. Then, the file that is displayed to users is actually produced by code under your control, which would help ensure that the image file is well formed and does not contain any artifacts that may fool the browser into interpreting it as a different MIME type (or try to exploit vulnerabilities in browser-side image parsing, for that matter).

  When following this approach, you have to be aware that your application will parse and process image files from untrusted sources. Image file formats are often rather complex, and it is not uncommon for third-party image-parsing libraries to themselves contain bugs and exploitable security vulnerabilities that might be exploited by a malicious, malformed image file. As such, you have to be careful, especially if you are using an image library written in a non-type-safe language such as C or C++.

- Ensure that there are no HTML tags in the first 256 bytes of the document, which could be done for example by prepending 256 whitespace characters. It is important to note that while we have empirically confirmed this number, and it is also consistent with Microsoft online documentation (see "MIME Type Detection in Internet Explorer," at `http://msdn.microsoft.com/workshop/networking/moniker/overview/appendix_a.asp`), there are no guarantees that future or very old versions of Internet Explorer may not behave differently.

- For plain text documents, one could also render the entire document as HTML (e.g., in `<pre>` tags) and HTML-escape the entire document. However, this may not always be appropriate (e.g., if the user should have the option of saving the document as a plain text file).

10.5.13. Mitigating the Impact of XSS Attacks

In the preceding sections, we discussed how to prevent XSS by eliminating its root cause: the injection of unvalidated or non-escaped strings that cause the execution of attacker-controlled script within a victim's browser. In the following discussion, we consider two strategies to mitigate the impact of XSS attacks in case your application is vulnerable to XSS despite your best efforts.

HTTP-Only Cookies

Internet Explorer implements an extension to the HTTP Cookie specification that allows a web server to add an additional attribute, `HttpOnly`, to cookies that it sets in the user's browser. When Internet Explorer receives a cookie with the `HttpOnly` attribute, it will not expose this cookie to client-side script (e.g., in the `document.cookie` DOM property); rather, such a cookie will only be sent to the server as part of HTTP requests. In case of an XSS attack against the web application, HTTP-only cookies cannot be accessed by the injected malicious script, and therefore cannot be sent to the attacker (for details, see the MSDN article "Mitigating Cross-Site Scripting with HTTP-Only Cookies," at `http://msdn.microsoft.com/workshop/author/dhtml/httponly_cookies.asp`).

While setting the `HttpOnly` attribute for session cookies in many cases prevents traditional session hijacking (i.e., scenarios in which an attacker obtains the victim's session cookies and uses them to access the victim's session with his own browser), you should not rely on this feature as your only protection mechanism against XSS attacks. There are several reasons why HTTP-only cookies provide incomplete protection:

- The `HttpOnly` attribute is at this time only supported by Internet Explorer, but not other popular browsers.
- Even with `HttpOnly` session cookies, XSS attacks with payloads that execute malicious actions directly within the user's browser are still possible. For example, using `HttpOnly` would generally not prevent XSS worms from propagating.
- If it is possible for an HTTP request to elicit a response that includes cookies sent as part of the request, injected script can extract HTTP-only session cookies from this response. For example, the response to the HTTP `TRACE` method, which is supported by many popular web servers, includes a copy of all the original HTTP request's headers, including cookies. Thus, if a web server supports `TRACE`, then session cookie hijacking is possible even if cookies are marked `HttpOnly`. It is therefore recommended to disable `TRACE` requests (as well as other debug requests whose responses might contain cookies) in the web server configuration if `HttpOnly` is used.

Binding Session Cookies to IP Addresses

If your web application receives multiple requests with the same session token, but from different IP addresses (especially if those IP addresses are known to be in far-apart geographic locations), then you have a strong indication that this session token has been hijacked. Based on this observation, it is worth considering mechanisms to mitigate session hijacking attacks by binding session cookies to IP addresses.

In the simplest case, your application could record the user's IP address at the time a session is initiated (e.g., when the user logs in using her username and password), and associate this IP address with this session and corresponding session cookie. If at a later time a request with this session cookie is received from a different IP address, the application would reject the request.

It is important to note that, like the `HttpOnly` cookie attribute, this scheme does not prevent XSS attacks whose payload does not rely on stealing session cookies, but rather executes immediately within the victim's browser.

In addition, there are a number of challenges in implementing a scheme that ties sessions to IP addresses without adversely affecting usability. There are a number of situations in which a user's session may legitimately result in requests originating from different IP addresses. For example, a user who accesses the Internet using a dial-up connection may be assigned a different IP address every time he connects. Similarly, it is common for laptop users to access the Internet via several different providers (and hence with different IP addresses) in a single day (e.g., from home, at work, or via the wireless network in a coffee shop).

For less frequently used applications involving sensitive data or high-value transactions (e.g., online banking applications), it may be appropriate to indeed tie sessions to a single IP address at login time and require the user to re-authenticate whenever his computer's IP address changes. For other applications, this may not be an acceptable user experience. For such applications, a heuristic approach might be more appropriate; the application might reject a request only if there is a very strong indication that it includes a stolen session cookie (e.g., a case in which a session is created based on a login request that originates from an IP address in California, and the corresponding session cookie appears 5 minutes later in a request that originates in Eastern Europe).