15.2 Cross-Site Scripting (XSS)

In this section we will examine one particular kind of security attack called cross-site scripting, or XSS for short, that is perhaps the most commonly seen in all of web programming. In 2007, XSS attacks accounted for over 80% of all web-based security attacks. Some XSS attacks are a mere nuisance, while others can lead to actual damage of data and other significant security problems.

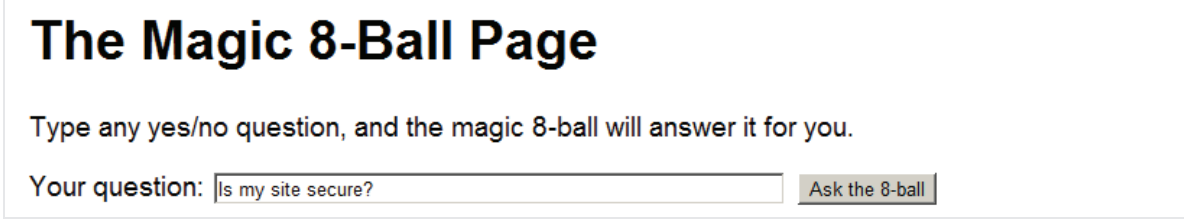
Cross-site scripting is the act of inserting malicious code into a web page to be viewed by others. XSS is a specific case of the concept of malicious *code injection*, the insertion of program code into an unexpected or undesired place in an application.

Most XSS attacks involve submitting HTML or JavaScript code into a form on a web site and causing that code to appear on the form's response page. The basic idea with a cross-site scripting attack is that if you, the attacker, can insert arbitrary content into a page on the server, you can insert anything from slanderous or embarrassing HTML content to malicious JavaScript code that executes functionality the web site's developer did not intend.

15.2.1 A Typical XSS Attack

Most of the examples we've shown in this textbook involving HTML forms and PHP code have been vulnerable to XSS attacks. Let's look at a particular example of a "Magic 8-Ball" page. The site is a PHP page that shows a form asking the user to type a question. In response, the page shows a picture of a Magic 8-Ball toy that gives the user a yes/no answer to their question, similar to the popular children's toy. The partial HTML source for the front page is shown in Example 15.1.

```
<h1>The Magic 8-Ball Page</h1>
<p>Type any yes/no question, and the magic 8-ball will answer it for you.</p>
<form action="8ball-answer.php" method="post">
  Your question: <input type="text" name="question" size="60" />
  <input type="submit" value="Ask the 8-ball" />
</form>
```



Example 15.1 Magic 8-ball page with form

The front page submits its data to a PHP response page named 8ball-answer.php. The answer page reads a file of one-line responses named sayings.txt where each line contains a single response that the 8-ball might give, such as "It is decidedly so" or "Outlook not so good". The answer page and a sample output from it are shown in Example 15.2.

```
<?php
$question = $_POST["question"];
$choices = file("sayings.txt");
$answer = $choices[rand(0, count($choices) - 1)];   # pick random answer
?>
...
<p>The answer to, <strong><q><?= $question ?></q></strong>, is:</p>
<div id="eightball"> <?= $answer ?> </div>
```

Example 15.2 Magic 8-ball form response page (not secure)

This simple site is actually not secure and is vulnerable to cross-site scripting attacks. The reason it is vulnerable is that it grabs the query parameter, `$_POST["question"]`, and directly inserts that parameter's value into the response page. Example 15.3 demonstrates. The attacker types a string containing HTML tags such as `<em>`. The answer page displays the content as formatted HTML, with an italic font and a green dotted border.

Example 15.3 Injecting HTML content into 8-ball page

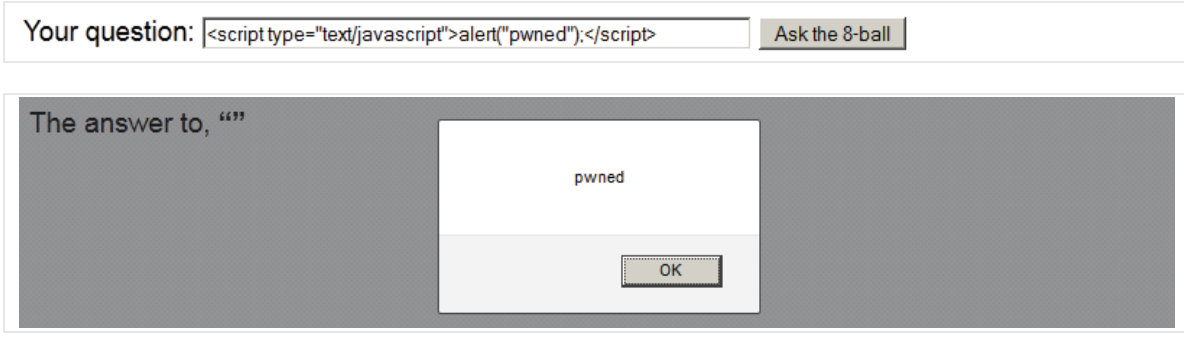
Injecting a few HTML tags for formatting is not very harmful. But there are many malicious things the attacker can do if it's possible to inject arbitrary HTML content into the page. One simple thing we could do is to inject something offensive or slanderous into the page, making it look like this site supports such a statement. This is shown in Example 15.4. The example input inserts tags to pre-

maturely close the "The answer to" paragraph, then begin a new paragraph containing an embarrassing false advertisement claiming that the site is written by terrorists (!), followed by a few final tags to hide the previous ", is:" text from appearing on the page. The response page with its phony claims about our web site is shown in the example. How annoying!

```
hi</q></strong>, is:</p><p>(This site was written by evil terrorists!)</p>
<p style="display: none"><strong><q>
```

Example 15.4 Injecting embarrassing HTML content into 8-ball page

Even worse than injecting embarrassing HTML content is to inject arbitrary JavaScript code. Our next attack, the true XSS attack, is shown in Example 15.5. The attacker types in a `<script>` tag containing some inline JavaScript code to be executed. When the answer page shows, the code is executed and the script `alert` window appears.



Example 15.5 Injecting JavaScript code into 8-ball page (XSS attack)

It may not be immediately clear why it is damaging for the attacker to be able to run JavaScript code like this. What's so bad about the attacker being able to pop up an `alert` box? The real problem is that JavaScript code executes with a certain amount of trust in a web browser. Users that visit web sites have the expectation that any code running on the site is written by the site authors themselves and therefore can be trusted as much as we trust that particular author. If the attacker can run arbitrary JavaScript code on your page, they can potentially insert commands that will run this code when other users visit the site. The attacker could cause code to run that makes malicious bank transfers from one account to another, or changes or discloses users' account data, or changes the page dynamically to the attacker's whim. XSS attacks can also lead to theft of session data, which can allow the attacker to perform identity theft and other damage. It is simply unacceptable to give attackers this kind of ability to modify the content of your site.

15.2.2 Defending Against XSS

Function	Description
`htmlspecialchars`	Replaces certain characters with HTML character entities
`htmlspecialchars_decode`	Converts all HTML entities from encoded forms back into plain text; opposite of `htmlspecialchars`
`htmlentities`	Replaces *all* characters that have an equivalent HTML character entity with that entity
`html_entity_decode`	Converts all HTML entities back to regular characters; the opposite of `htmlentities`

Table 15.1 PHP functions for encoding and decoding HTML entities

The proper way to deal with most types of code injection, including cross-site scripting, goes back to our discussion of the security mindset. We must not trust the input sent to us by the user. In particular, we must never directly insert that input into our own pages without first processing it to make sure that it is free of HTML or JavaScript or other malicious code.

Security Note

Never insert user input directly into an HTML page. (It must be encoded first.)

The real problem in this case is that the user's input can contain characters like `<` or `&` that have meaning in HTML's syntax. We could write code of our own to look for such characters and encode them into safer character entity equivalents. But that isn't necessary, because the authors of PHP have already taken care of this for us. There are several provided functions in PHP that accept a string as a parameter and return a safely HTML-encoded version of that string. The most commonly used function is `htmlspecialchars`. When you call this function and pass it a string, it returns a new string with each potentially harmful character replaced by an equivalent HTML character entity reference. For example, `<` is replaced by `<` in the encoded string. This process of replacing HTML content with character entities is also called *HTML-encoding* a string. Example 15.6 shows a sample call.

```
$text = "<p> hi 2 u & me </p>";
$text = htmlspecialchars($text);   # "&lt;p&gt; hi 2 u &amp; me &lt;/p&gt;"
```

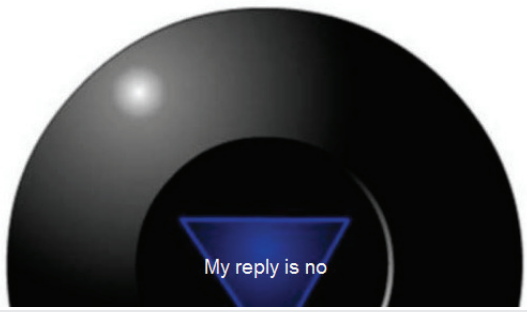
Example 15.6 Using `htmlspecialchars` to encode text

Another function `htmlentities` is like `htmlspecialchars` but more aggressive. The `htmlentities` function replaces every character that has an equivalent entity with that entity, even if the character was not potentially harmful. For example, `htmlentities` replaces all apostrophe `'` characters with `&#039;`. But we don't need this extra level of encoding to protect against XSS attacks, and it leads to larger and less readable strings, so we recommend using `htmlspecialchars` instead.

Example 15.7 shows our Magic 8-ball answer page modified to use `htmlspecialchars`. The sample output shown demonstrates that now if the attacker tries to insert HTML or JavaScript content into the page, the tags are simply echoed (because they are encoded now with entities like `<`) rather than actually being inserted as HTML or JavaScript code in the page. The page is now secure against cross-site scripting attacks.

```
<?php
$question = $_POST["question"];
$choices = file("sayings.txt");
$answer = $choices[rand(0, count($choices) - 1)];   # pick random answer
?>
...
<p>The answer to,
   <strong><q><?= htmlspecialchars($question) ?></q></strong>, is:</p>
<div id="eightball"> <?= htmlspecialchars($answer) ?> </div>
```

The answer to, **"<script type="text/javascript">alert("Did I pwn you?");</script>"**, is:



Example 15.7 Magic 8-ball form response page (secure)

Note that `htmlspecialchars` is not being called until the moment that the input values are inserted with PHP expression blocks (`<?= ... ?>`). This is a stylistic choice; you could call `htmlspecialchars` as soon as you read the query parameters into variables, but that can be annoying if you want to analyze the parameter's value to look for some particular text or value. If you're going to insert the value into the page many times, it may be better to encode it early so that you won't forget to do so later.

Another interesting note about this code is that we also call `htmlspecialchars` on the line of text being inserted from our input file, sayings.txt. This may seem unnecessary; the attacker doesn't have access to that file, so why encode the text that comes out of the file? It's mostly defensive programming. Who knows if there is some vulnerability somewhere else in our site or server that allows the attacker to gain access to that file and modify its contents? If so, the attacker could put lines into that file that contain malicious HTML or JavaScript code. Or even if no attacker is involved, the person who manages the sayings file might innocently insert a saying with a special character in it such as `<` or `&`, not knowing that these characters have special meanings in HTML. By being safe and encoding the line from the file that we inject into the page, we reduce the risk that the file's contents could lead to a code injection attack on this page. Better safe than sorry.

Self-Check

7. What mistake in a web site's code leads to it being vulnerable to XSS attacks?
8. What kinds of content can an attacker insert into a page in an XSS attack, and what sort of damage can result from inserting such content?
9. What changes are made to a string when it is HTML-encoded? Why do these specific changes make the string safer to use in an HTML page?
10. Why might you want to HTML-encode a piece of data that did not come from user input?