Mitigating XSS Risk

Security

Otago Polytechnic
Dunedin, New Zealand

The problem

- We saw last week that XSS vulnerabilities happen when:
  1. We trust user input to be valid and safe;
  2. We output unsafe data to the browser without processing it.
- We can also make the vulnerability worse by being too accepting of various types of HTTP requests.

It's important to recognise these patterns so that you know when to be on the alert for XSS vulnerabilities.

Vulnerable code

```
<p> Hi, <?php echo($_COOKIE['user']); ?> </p>
<p><img src="picard.jpg" /></p>
<p> <?php echo($_REQUEST['secure']); ?> </p>‘
```

Step one: processing output

- Much of the problem comes from outputting things like unwanted“`<script>`” tags to the browser.
- Usually, we don’t want to pass any HTML from user input back to the browser.
- It turns out that characters like “<” and “>” have alternative representations.
- For eample, “>” can be represented as “`&gt;`”.
- These representations are safer to send to the users’ browser.

In PHP:

Instead of

```
<?php echo($unsafe-data); ?>
```

Use

```
<?php echo(htmlentities($unsafe-data)); ?>
```

This doesn't solve everything

- It's important to read the documentation[1] for `htmlentities()` to use it correctly.
- It may stil be possible to get some malicious code past `htmlentities()`.
- There are also situations where you may want to allow characters that `htmlentities` transforms to pass through to the browser.
- The bottom line is that it *improves* our vulnerability exposure considerably.

[1] `http://php.net/htmlentities`

Step two: processing input

- It's important to avoid sending malicious code to browsers, but it's probably best to avoid accepting malicious code in input.
- One approach would be to write a function that filters out everything that's "bad".
- This can work, but now we have to anticipate everything bad that could come in input.
- Another approach is to identify what we *want* in our input and to write functions that only accept that.
- This approach requires more work, but it's easier to get right.

Example

- In our sample code last week we had users enter their names.
- It's reasonable to place some conditions on user names.
  - not more than 32 characters
  - only letters, digits, - and _
- We can write a function to check that these conditions are met.

Username checker

```
function legal_username($name) {
    $length_ok = strlen($name) < 33;
    $format_ok = preg_match("/^[a-zA-Z-_]+$/", $name);
    return $length_ok and $format_ok;
}
```

Using the username checker

```
<?php
 if(legal_username($_POST['un'])) {
  setcookie("user", $_POST['un'], mktime().time() + 7200);
 }
  ?>
```