

# Cook up Web sites fast with CakePHP, Part 3: Use Sanitize for your protection

How to lock down your CakePHP applications with Sanitize and Security

Level: Intermediate

Duane O'Brien ([d@duaneobrien.com](mailto:d@duaneobrien.com)), PHP developer, Freelance

19 Dec 2006

CakePHP is a stable production-ready, rapid-development aid for building Web sites in PHP. This "[Cook up Web sites fast with CakePHP](#)" series shows you how to build an online product catalog using CakePHP. [Part 1](#) focuses on getting CakePHP up and running, and [Part 2](#) demonstrates how to use scaffolding and Bake. In this article, you will learn how to use CakePHP's Sanitize and Security components to help secure your user-submitted data. You will also learn how to handle invalid requests.

## Introduction

This series is designed for PHP developers who want to start using CakePHP to make their lives easier. In the end, you will have learned how to install and configure CakePHP, the basics of Model-View-Controller (MVC) design, how to validate user data in CakePHP, how to use CakePHP Helpers, and how to get an application up and running quickly using CakePHP. It might sound like a lot to learn, but don't worry -- CakePHP does most of it for you.

This article assumes you have already completed [Part 1](#) and [Part 2](#), and that you still have the working environment you set up for those tutorials. If you do not have CakePHP installed, you should run through Parts 1 and 2 before continuing.

It is assumed that you are familiar with the PHP programming language, have a fundamental grasp of database design, and are comfortable getting your hands dirty.

## System requirements

Before you begin, you need to have an environment in which you can work. CakePHP has reasonably minimal server requirements:

1. An HTTP server that supports sessions (and preferably `mod_rewrite`). This tutorial was written using Apache V1.3 with `mod_rewrite` enabled.
2. PHP V4.3.2 or later (including PHP V5). This tutorial was written using PHP V5.0.4
3. A supported database engine (currently MySQL, PostgreSQL or using a wrapper around ADODB). This tutorial was written using MySQL V4.1.15.

You'll also need a database ready for your application to use. The tutorial will provide syntax for creating any necessary tables in MySQL.

The simplest way to download CakePHP is to visit [CakeForge.org](#) and download the latest stable version. This tutorial was written using V1.1.8. (Nightly builds and copies straight from Subversion are also available. Details are in the CakePHP Manual (see [Resources](#)).)

## Tor, so far

At the end of [Part 2](#), you were given another opportunity to put your skills to work by enhancing *Tor*. You were to use Bake to generate views and a controller for dealers, then verify that the dealer name is unique, fix a bug in the products ACLs, and change the products view to only show **Edit** and **Delete** buttons for the users who can use them. It was a long to-do list. How did you do?

### The login view

Baking the dealers controller and views should have been straightforward. From your CakePHP installation directory, you would have run **php cake/scripts/bake.php**.

To generate the controller, you would have entered `C` in the first menu and specified the controller name `dealers`. To generate the views, you would have entered `V` in the first menu and specified the controller name `dealers`. These steps were virtually identical to the steps you followed to create the products controller and views.

You also had to alter the `add` action in the dealers controller to verify that a dealer with the submitted dealer name didn't exist. You might have done so with something like the following:

#### Listing 1. Altering the add action in dealers

```
if ($this->Dealer->findByTitle($this->data['Dealer']['title']))
{
    $this->Dealer->invalidate('title');
    $this->set('title_error', 'A dealer with that title already exists.');
```

This would have also required the setting and usage of a `title_error` variable in the `add` view for the dealer, much like you did for the user `add` view.

### Add product bug fix

Another one of your tasks was to fix a bug in the `products add` method. As the method was left at the end of [Part 2](#), anyone could add a product, even if logged out. There are several ways to fix this.

Using access control lists (ACLs), you could have created an action in the products controller and used it to grant *create* access to the users access request object (ARO) on each dealer access control object (ACO). You then would have deleted the action since you didn't need it anymore. Next, you could have modified the `dealer add` action to grant *create* permissions for the dealer ACO to the user's ARO. Finally, in the `products add` method, you could have checked to make sure the user had permissions. Or you could have done something like that shown below.

#### Listing 2. Checking if a user is logged in on users index action

```
function add() {
    $username = $this->Session->read('user');
    if ($username)
    {
        ... the rest of your add function goes here
    } else {
        $this->redirect('/users/login');
    }
}
```

Look familiar? It should. It's the same check you use to see if a user was logged in on the users index action. If the user is logged in, he is obviously a user. It's something of a trick solution, but it does show you that there's more than one way to skin a user.

Dealer ACLs

As it stands, the dealer actions are completely unprotected. Again, these should be modified so that only users can actually use them, but ACLs that are more stringent could and should be applied, though full coverage of this is too big to cover here. Consider something like this:

- Any user can add or view a dealer.
- Only the user who created a dealer can modify or delete the dealer.

This could be worked into the application further, such as:

- If a user has created a dealer, he can add another user to the dealership.
- Any user in a dealership can modify any product created for dealership.

There are many other ways this could be done. Experiment with a few. Don't be afraid to get your hands dirty.

Products view enhancements

The last thing you needed to do was modify the products views so that **Edit** and **Delete** buttons are only shown to the users who can edit or delete the product. Probably the easiest way to do this would be to remove the **Edit** and **Delete** buttons from the index view completely. You could just as easily check each product for the user's rights and show or hide the buttons, but the performance might not be what you want.

Then in the view action, you could check to see if the user has permission to update or delete the product and set variables that can be used to display or hide the buttons in the view. In the view action in the controller, you should have something like that shown below.

Listing 3. Possible view action solution

```
if ($this->Acl->check($this->Session->read('user'), $id.'-'.$product['Product']['title'],
'update'))
{
    $this->set('showUpdate', TRUE);
} else {
    $this->set('showUpdate', FALSE);
}
if ($this->Acl->check($this->Session->read('user'),
$id.'-'.$product['Product']['title'],
'delete'))
{
    $this->set('showDelete', TRUE);
} else {
    $this->set('showDelete', FALSE);
}
```

And in products/view.thtml, you need something like what is shown here:

Listing 4. Possible products/view.thtml solution

```
<li><?php if ($showUpdate) { echo $html->link('Edit Product', '/products/edit/' .
$product['Product']['id']); } ?> </li>
<li><?php if ($showDelete) { echo $html->link('Delete Product', '/products/delete/'
. $product['Product']['id'], null, 'Are you sure you want to delete: id ' .
$product['Product']['id'] . '?' ); } ?> </li>
```

Again, don't worry if your solutions don't exactly match these solutions. They are intended to be examples, not Dogma. You can download the code thus far to get on the same page.

Data security

When dealing with a Web application, the importance of data security cannot be overstated. At the risk of sounding like a credit security commercial, you have to be constantly on your guard against hackers, crackers, script kiddies, identity thieves, spammers, phishers, criminals, and just plain old bad people. Yet while it has been this way for years, data security is still frequently treated lightly. It doesn't matter if you write the most beautiful and elegant Web application in the history of mankind. Bad data security will bring your application to its knees. To understand how to deal with bad data, you should get familiar with some of the basic problems you face.

What it means to secure your data

Despite bad data security causing so much trouble, the whole problem can be summarized in five words: know what you are getting. Data security does not mean stripping out all HTML -- though you may want to. Nor does it mean stripping out any special characters -- though you may want to. The fundamental principle is that your application knows what it is getting.

This is not the same as knowing what you want. Nor is it the same as knowing what your application has requested. And it's certainly not the same thing as accepting everything and anything.

SQL injection

A SQL injection vulnerability occurs when a user is able to pass SQL code directly to the application in such a way that the code will be executed in a query. For example, the user is presented with a login screen, in which the username and password is entered. The password variable might be used in the following query.

```
"select * from users where username = '" + $username + "'"
```

```
and password = '' + $password + '''
```

Consider what would happen if the user submitted the following password: ' or '1' = '1. The final SQL statement might look something like this:

```
"select * from users where username = 'wrestler'
and password = 'secret' or '1' = '1' "
```

Not checking for SQL-specific characters can open your application up to a wide range of vulnerabilities. CakePHP's Sanitize component can help make this easy.

### Cross-site scripting

Cross-site scripting (XSS) refers to a large classification of vulnerabilities that focus on the ability to present malicious code to an unsuspecting user. This usually takes the form of malicious JavaScript, which could do anything from annoying the user to capturing data from cookies.

At the heart of the XSS vulnerability is an application that filters user data improperly once submitted. For example, suppose you're building an application with a forum and did no filtering against the user data. Anything submitted for a forum post could be displayed. As an evil user, suppose I submit the following text as a forum post:

```
<script>alert("EXPLETIVES!!!")</script>
```

If the application permits this text, anyone who viewed my post would get a JavaScript alert box that shouted expletives at them. While not particularly harmful, it's certainly not something you want the boss to see.

This is a very simple example of a harmless XSS exploitation. While this example is fairly harmless, XSS is anything but. XSS has been used to steal passwords, steal credit card numbers, forge news stories, and much more. Protecting yourself and your application from XSS is important. CakePHP can help you with Sanitize.

### Cross-Site Request Forgery

Cross-Site Request Forgery (CRSF) may not be as popular or as well known as XSS, but that doesn't make it any less dangerous.

To demonstrate, suppose your application included a forum, and the forum granted a thread's author permission to delete the thread. You have built this into the forum as a button that is only shown to the author, and you even verify that the author is the one who made the request before performing the actual delete. It might work by posting a field name called `action` with the value `delete`, and a field name called `id` that contains a unique ID for the thread. In a query string, it might look something like `http://localhost/forum.php?action=delete&id=1729`.

Now suppose we post an image, or have the ability to specify an image as a signature, and we provide as the URL to that image the same query string. In HTML, it would ultimately look something like this:

```

```

I can't go to the URL directly myself because I'm not the author, and the application knows it. But if the thread views my post, the browser will attempt to load the image by requesting `http://localhost/forum.php?action=delete&id=1729` -- and since it is the author making the request, the thread gets deleted.

This is an overly simple look at CSRF and how it works. CakePHP's Security component can help protect you.

## Sanitize

Enter Sanitize, which is CakePHP's class to help you deal with data security issues. Unlike the ACL component discussed in [Part 2](#), the Sanitize component is included by adding a line to the top of your controller. For example, if you want to use Sanitize in your products controller, the top of your controller might look like this:

### Listing 5. Using Sanitize in Products

```
<?php
uses('sanitize');
class ProductsController extends AppController
{
    ...
}
```

Sanitize provides four methods for applying varying levels of data security to user-submitted data. Each method serves a distinct purpose.

### The Sanitize paranoid method

This is the strictest of the available methods. The `paranoid` method will strip a string of all characters that are not alphanumeric (a-z, A-Z or 0-9). The `paranoid` method accepts an input string and an optional array (`$allowedChars`). The `$allowedChars` array can be used to pass an array of characters that are also permissible. For example, if you wanted data to be alphanumeric, underscores, and periods, you would use something like this:

```
$san = new Sanitize();
$clean = $san->paranoid($your_data, array('_', '.'));
```

**NOTE:** The `paranoid` method will strip out any spaces unless you pass ' ' as an allowed character in the `$allowedChars` array.

This approach to data security is known as *whitelisting*. Rather than stripping out characters you don't want (*blacklisting*), you are stripping out all characters except for those you have explicitly stated are acceptable. This works well for dealing with pieces of data that must follow specific rules (such as user names, e-mail addresses, and passwords), but a whitelisting approach can be used for almost any type of nonbinary data.

### The Sanitize html method

The `html` method of Sanitize can pass two parameters: the string being Sanitized and an optional Boolean flag referred to as `$remove`.

If `$remove` is set to `true`, the `html` method passes the string to the PHP function `strip_tags`, which will return the string minus any HTML tags. For example, `strip_tags("<p>Hello</p>", true)` will return the value `Hello`.

If `$remove` is `false` or not set, then the `html` method replaces some characters with HTML entities. Specifically:

- `&` is replaced with `&`
- `%` is replaced with `%`
- `<` is replaced with `<`

- > is replaced with >
- " is replaced with "
- ' is replaced with '
- ( is replaced with (
- ) is replaced with )
- + is replaced with +
- - is replaced with -

The `html` method uses `preg_replace` to perform these replacements.

### The Sanitize sql method

The `sql` method escapes some special characters in a string with backslashes in order to prepare the string for use in a SQL statement.

Specifically, the `sql` method first checks to see if `magic_quotes_gpc` is set to `false`. If so, the string is passed to the PHP function `addslashes`, which escapes single quotes, double quotes, backslashes, and null byte characters with backslashes. For example, `addslashes( "O'Brien" )` would return `O\'Brien`. If `magic_quotes_gpc` is set to `true`, PHP automatically escapes the special characters and, therefore, no other action is necessary.

### The Sanitize cleanArray method

The `cleanArray` method takes an array as input and returns the same array after it has been recursively "cleaned," in this case meaning that the following actions are performed:

- Using `str_replace`, spaces are normalized to " " (each space is replaced with " ", as is each occurrence of `chr(0xCA)`).
- The value is passed through the `html` method to replace HTML entities.
- Any occurrence of `\$` is replaced with `$`.
- Any newline is removed.
- `!` is explicitly replaced with `!`
- `'` is explicitly replaced with `'`
- Any occurrence of a double-encoded HTML entity is repaired (for example, `'` will be replaced with `'`).
- The value is passed through the `sql` method to prepare the string for use in a SQL statement.
- Backslashes input by the user are evaluated and, in cases where they would cause special characters to be escaped, the backslashes are fixed.

While it would have been easy enough to state "cleanArray walks an array and cleans each value," it is important to understand what is being done behind the scenes to protect your data.

You can view these methods in `cake/libs/sanitize.php` to get a better handle on what's involved. You should never modify the base CakePHP files, however. It will make upgrading to future releases difficult and could cause unexpected behavior.

## CakePHP's Security component

To use the CakePHP Security component, add it to the array of components in your controller, as you did with ACL.

```
var $components = array ( 'Acl' , 'Security' );
```

By simply including the Security component, several things happen automatically, even if you don't use it to protect an action:

- Using the core Security class, an authentication key is generated and written to the session. The authentication key will expire according to the settings in `app/config/core.php`.
- The authentication key is set as a class variable in your controller.
- If any view generated by the controller uses `$html->formTag( )` to create a form, a hidden input field named `_Token/key` will be included in the form containing the authentication key. When the form is posted, the value of this field is compared to the value of the authentication key in session, to verify that the form submission was valid. Once this validation takes place, a new authentication key is generated and set in the session.

Now that you have included the Security component, you need to create the method `beforeFilter` in the controller using the Security component. This method will be executed automatically by the controller before any method called by the user. This is the place you typically will want to perform Security checks.

There are a couple ways you can use what the Security component is doing for you here. The first is to require that forms use the `POST` method. The second is to require a valid authentication key. Using the two together creates a powerful security base for your application.

### requirePost

The Security `requirePost` method tells CakePHP to ignore any information submitted to the specified action unless `POST` was used. The `requirePost` method is passed a list of actions within the controller you want to protect. For example, if you wanted to use `requirePost` to protect the `delete` and `add` methods, your `beforeFilter` method would look like this:

```
function beforeFilter()
{
    $this->Security->requirePost('delete' , 'add');
}
```

By requiring that an action only use data from a form post, you eliminate the ability for someone to forge a request using a query string.

### requireAuth

The Security `requireAuth` method tells CakePHP to check validate any form submission with the authorization key mentioned earlier. This validation will only happen if the form was submitted via `POST`.

Like the `requirePost` method, `requireAuth` is passed a list of actions within the controller you want to protect. For example, if you wanted to use `requireAuth` to protect the `delete` and `add` methods, your `beforeFilter` method would look like this:

```
<
function beforeFilter()
{
    $this->Security->requireAuth('delete' , 'add');
}
```

To use `requireAuth` and `requirePost`, you would just specify both in the `beforeFilter` method.

**Listing 6. Specifying `requireAuth` and `requirePost`**

```
function beforeFilter()
{
    $this->Security->requireAuth('delete', 'add');
    $this->Security->requirePost('delete', 'add');
}
```

Using both `requireAuth` and `requirePost` to protect an action is a powerful combination. But you can even mix and match the two if you want varying levels of protection for different methods.

**Listing 8. Mix and match `requireAuth` and `requirePost`**

```
function beforeFilter()
{
    $this->Security->requireAuth('delete');
    $this->Security->requirePost('delete', 'add');
}
```

By requiring that a form submission contains a valid authorization key before processing, you once make it much more difficult for someone to forge a form submission for another user. Using `requireAuth` and `requirePost` together is a great way to help lock down your application.

**A word about caching**

While there are obvious advantages to using `requireAuth` to protect actions, there are also some disadvantages that should be addressed. Most of these disadvantages fall into the category of "aching problems."

The authentication key is regenerated every time a form is evaluated with `requireAuth`. This means that if a user submits a form with a key that has already been used, the form submission will be considered invalid. There are several cases in which this could occur, including but not limited to using multiple browser windows, using the **Back** button to return to a previous page, browser caching, proxy caching, and more. While you may be tempted to write off these problems as "user error," you should resist the temptation and plan on handling invalid form submissions gracefully.

**What happens to invalid form submissions?**

If a request is rejected by `requirePost` or `requireAuth`, the application exits and the user is sent a 404 page. To override this behavior, you can set the `$blackHoleCallback` property of the Security component to the name of a function within the controller.

For example, if you had an action called `invalid` and a corresponding view, you could set tell the Security component to send bad requests to the `invalid` action. You can do this by adding the following line to the beginning of your `beforeFilter` method: `$this->Security->blackHoleCallback = 'invalid';`

This gives you some control over presentation in the event that a user manages to submit an invalid request via legitimate means. A good example of this would be a user using the application, taking a lunch break, and returning to the application after the authentication key had expired.

**Filling in the gaps**

Now that you know how to use the Sanitize and Security components, put them to work in Tor.

Start off by sanitizing everything. For products, users and dealers, sanitize all data submitted. It's up to you to determine which Sanitize method is best used. Next, look at the controllers and their actions, and determine which ones need to be protected using `requireAuth`, and which ones need to be protected using `requirePost`. Implement the Security component and use it accordingly. Finally, create an action for each controller called `invalid` and use the method to inform the user that their form submission was invalid.

**Summary**

The Sanitize and Security components of CakePHP are *not* magic bullets. Using them does not mean you can forget about application security. However, using them will make your life much easier in terms of dealing with some of the more common security-related tasks. By cleaning up your data and ignoring data submitted improperly, you have made excellent first steps in securing your application.

Part 4 focuses primarily on the Session component of CakePHP, demonstrating three ways to save session data, as well as the Request Handler component to help you manage multiple types of requests (mobile browsers, requests containing XML or HTML, etc.).

**Download**

Description	Name	Size	Download method
Part 3 source code	s-php-cake3.source.zip	9KB	<b>HTTP</b>

→ [Information about download methods](#)

**Resources**

**Learn**

- Visit [CakePHP.org](#) to learn more about it.
- The [CakePHP API](#) has been thoroughly documented. This is the place to get the most up-to-date documentation for CakePHP.

- There's a ton of information available at [The Bakery](#), the CakePHP user community.
- [CakePHP Data Validation](#) uses PHP Perl-compatible regular expressions.
- Read a tutorial titled "[How to use regular expressions in PHP](#)."
- Want to learn more about design patterns? Check out [Design Patterns: Elements of Reusable Object-Oriented Software](#).
- Check out some [Source material for creating users](#).
- Check out the [Wikipedia Model-View-Controller](#).
- Here is more useful background on the [Model-View-Controller](#).
- [Here's a whole list](#) of different types of software design patterns.
- Read about [Design Patterns](#).
- Visit IBM developerWorks' [PHP project resources](#) to learn more about PHP.
- Stay current with [developerWorks technical events and webcasts](#).
- Check out upcoming conferences, trade shows, webcasts, and other [Events](#) around the world that are of interest to IBM open source developers.
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.
- To listen to interesting interviews and discussions for software developers, be sure to check out [developerWorks podcasts](#).

#### **Get products and technologies**

- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

#### **Discuss**

- The developerWorks [PHP Developer Forum](#) provides a place for all PHP developer discussion topics. Post your questions about PHP scripts, functions, syntax, variables, PHP debugging and any other topic of relevance to PHP developers.
- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

#### **About the author**

Duane O'Brien has been a technological Swiss Army knife since the Oregon Trail was text only. His favorite color is sushi. He has never been to the moon.