Preventing XSS

XSS stands for cross-site scripting and is a type of vulnerability that allows one to inject a client-side script  
(typically JavaScript) in a page viewed by other users. Considering the power of client-side scripting, this  
can lead to very serious consequences such as bypassing security checks, getting other user’s credentials,  
or data leaks.

In this recipe, we will see how to prevent XSS by escaping the output with both \yii\helpers\Html and  
\yii\helpers\HtmlPurifier.

Getting ready

1. Create a new application by using the Composer package manager, as described in the official  
   guide at [http://www. yiiframework. c om/doc-2.0/guide -start-installation .html](http://www.yiiframework.com/doc-2.0/guide-start-installation.html)**.**
2. Create controllers/XssController.php:

<?php

namespace app\controllers;  
use Yii;

use yii\helpers\Html;  
use yii\web\Controller;

/\*\*

* Class SiteController.
* @package app\controllers  
  \*/

class XssController extends Controller  
{

/\*\*

\* @return string  
\*/

public function actionIndex()

{

$username = Yii::$app->request->get('username', 'nobody');

return $this->renderContent(Html::tag('h1',

'Hello, ' . $username . '!'

));

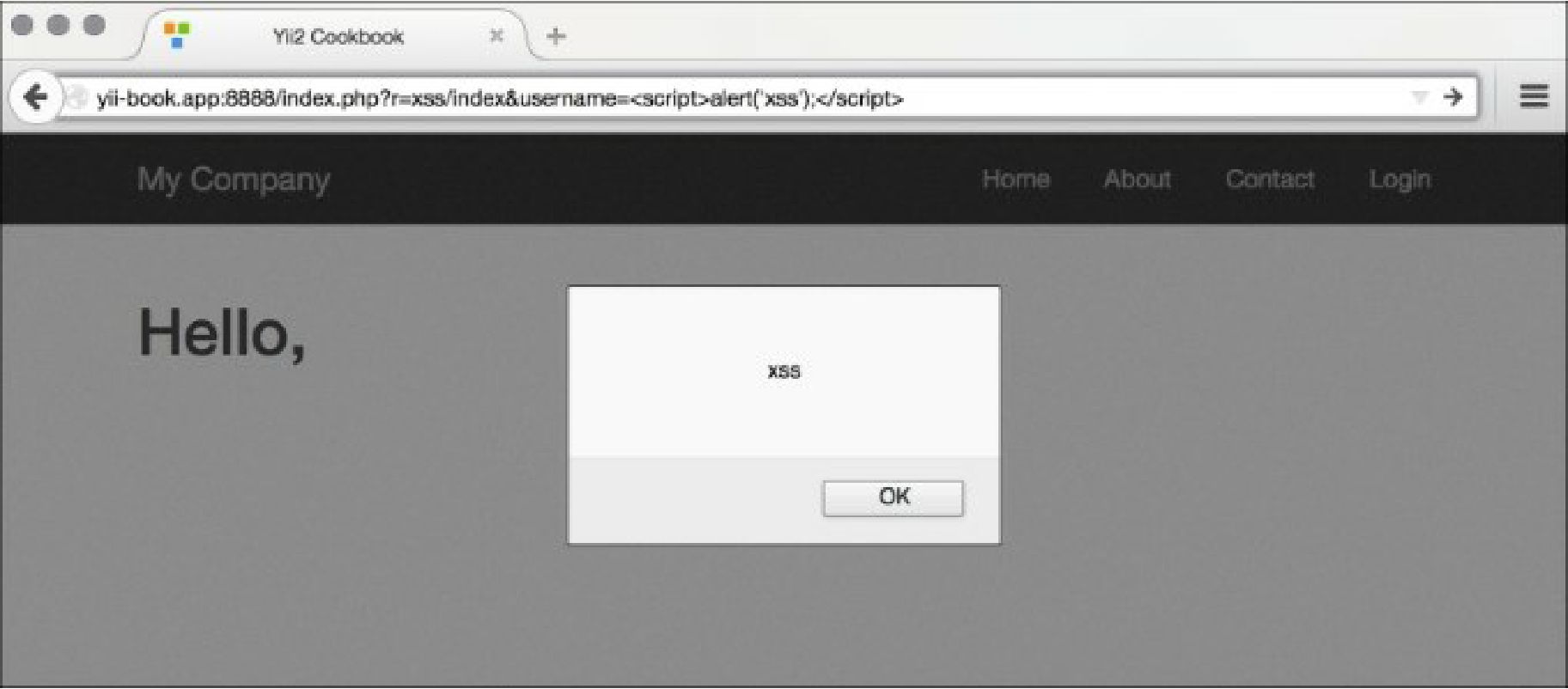
}

}

1. Normally, it will be used as /xss/simple?username=Administrator. However, as the main  
   security principle **filter input, escape output** was not taken into account, malicious users will be  
   able to use it in the following way:

/xss/simple?username=<script>alert('XSS');</script>

1. The previous code will result in a script execution, as shown in the following screenshot:



How to do it...

Carry out the following steps:

1. In order to prevent the XSS alert shown in the previous screenshot, we need to escape the data  
   before passing it to the browser. We do this as follows:

<?php

namespace app\controllers;  
use Yii;

use yii\helpers\Html;  
use yii\web\Controller;

/\*\*

* Class SiteController.
* @package app\controllers  
  \*/

class XssController extends Controller  
{

/\*\*

\* @return string  
\*/

public function actionIndex()

{

$username = Yii::$app->request->get('username', 'nobody');

return $this->renderContent(Html::tag('h1',

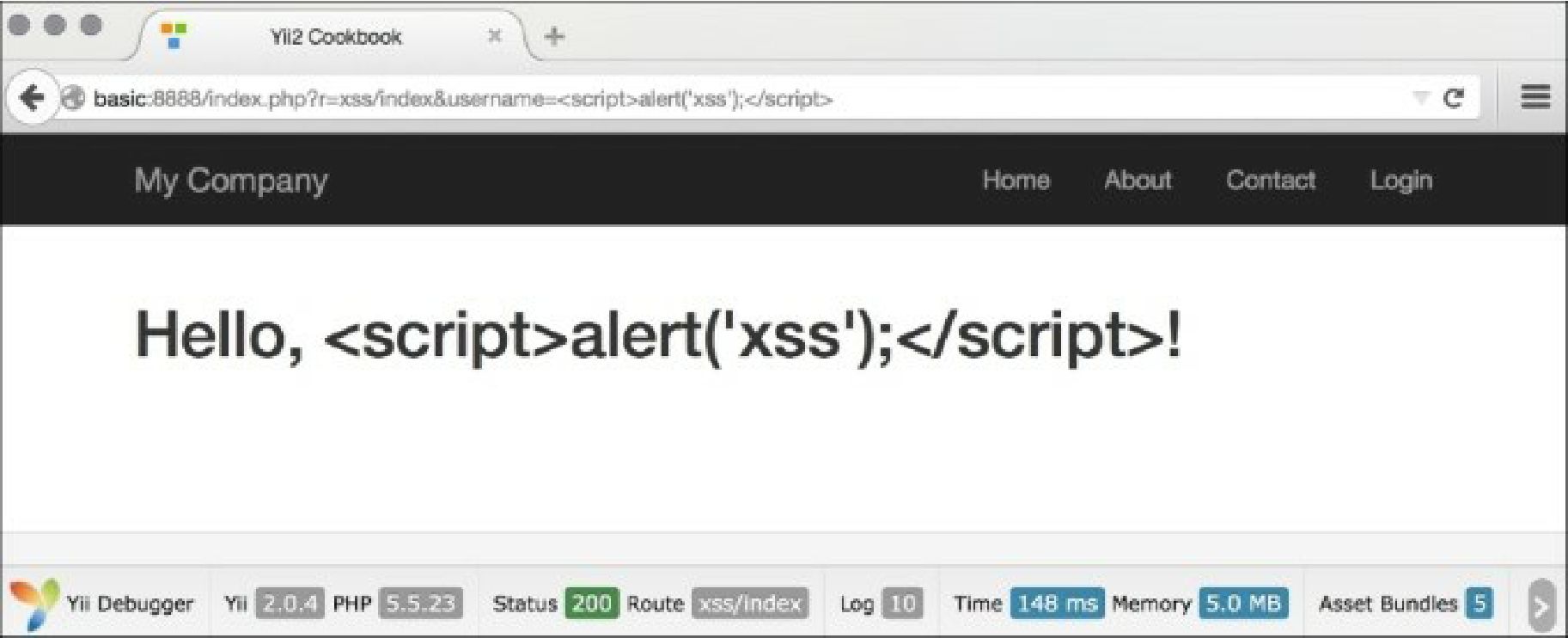
Html::encode('Hello, ' . $username . '!')

));

}

}

1. Now instead of an alert, we will get properly escaped HTML, as shown in the following screenshot:



3. Therefore, the basic rule is to always escape all dynamic data. For example, we should do the same  
for a link name:

use \yii\helpers\Html;

echo Html::a(Html::encode($\_GET['username']), array());

That’s it. You have a page that is free from XSS. Now, what if we want to allow some HTML to pass?  
We cannot use \yii\helpers\Html: : encode anymore because it will render HTML as just a code and  
we need the actual representation. Fortunately, there is a tool bundled with Yii that allows you to filter the  
malicious HTML. It is named html Purifier and can be used in the following way:

<?php

namespace app\controllers;  
use Yii;

use yii\helpers\Html;

use yii\helpers\HtmlPurifier;

use yii\web\Controller;

/\*\*

* Class SiteController.
* @package app\controllers  
  \*/

class XssController extends Controller  
{

/\*\*

\* @return string  
\*/

public function actionIndex()

{

$username = Yii::$app->request->get('username', 'nobody');

$content = Html::tag('h1', 'Hello, ' . $username . '!');

return $this->renderContent(

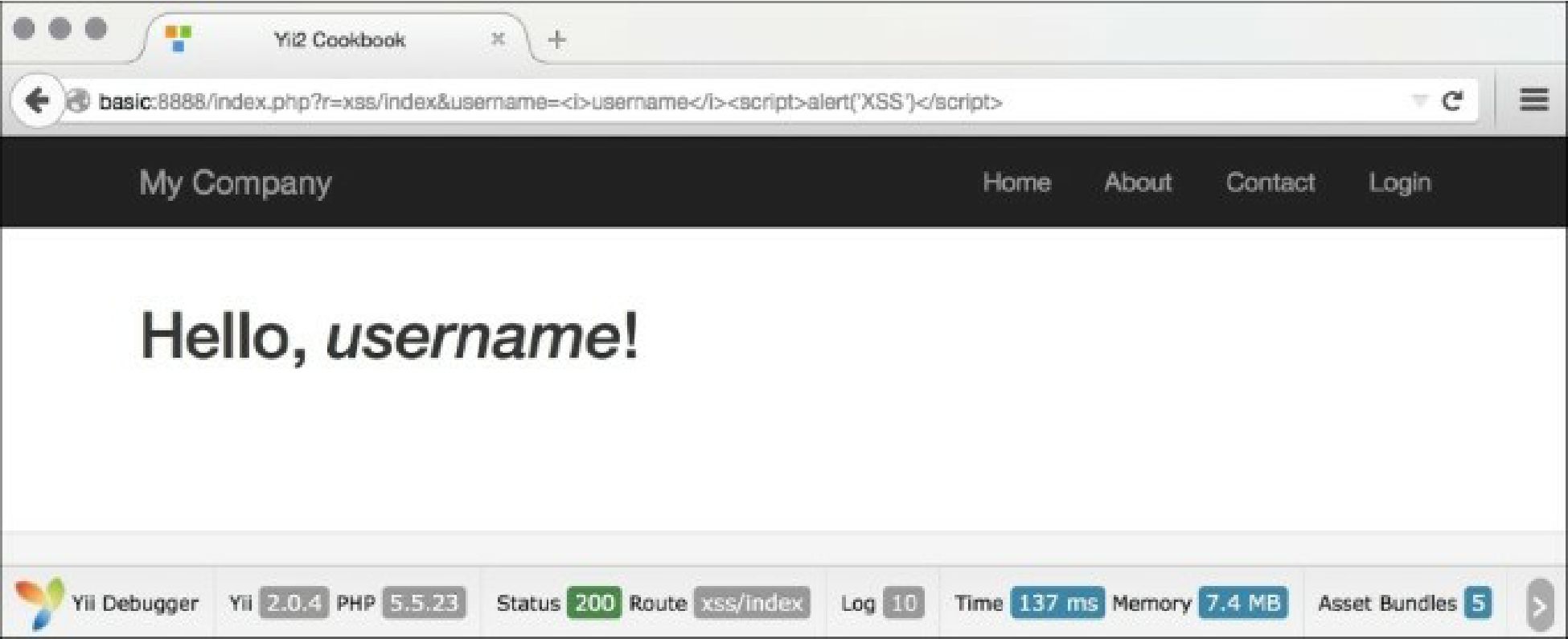
HtmlPurifier::process($content)

);

}

}

Now if we access the HTML action using a URL such as /xss/index?username=<i>username</i> !  
<script>alert( 'xss' )</script>, HTML Purifier will remove the malicious part and we will get the  
following result:



How it works...

1. Internally, \yii\helpers\Html: :encode looks like the following:

public static function encode($content, $doubleEncode = true)

{

return htmlspecialchars($content, ENT\_QUOTES | ENT\_SUBSTITUTE, Yii::$app ?  
Yii::$app->charset : 'UTF-8', $doubleEncode);

}

1. So basically, we use PHP’s internal htmlspecialchars function, which is pretty secure if one does  
   not forget to pass the correct charset in the third argument.

\yii\helpers\HtmlPurifier uses the HTML Purifier library, which is the most advanced solution out  
there to prevent XSS inside of HTML. We have used its default configuration, which is okay for most  
user-entered content.

There’s more.

There are more things to know about XSS and HTML Purifier; they are discussed in the following section.

XSS types

There are two main types of XSS injections, which are as follows:

* Non-persistent
* Persistent

The first type is the one we have used in the recipe and is the most common XSS type; it can be found in  
most insecure web applications. Data passed by the user or through a URL is not stored anywhere, so the  
injected script will be executed only once and only for the user who entered it. Still, it is not as secure as it  
looks. Malicious users can include XSS in a link to another website and their core will be executed when  
another user follows the link.

The second type is much more serious, as the data entered by a malicious user is stored in the database  
and is shown to many, if not all, website users. Using this type of XSS, malicious users can literally destroy  
your website by commanding all users to delete all data to which they have access.

See also

In order to learn more about XSS and how to deal with it, refer to the following resources:

* [http://htmlpurifier. org/docs](http://htmlpurifier.org/docs)
* <http://ha.ckers.org/xss.html>
* [http://shiflett.org/blog/2nn7/may/character-encoding-and-xss](http://shiflett.org/blog/2007/may/character-encoding-and-xss)