Preventing SQL injections

SQL injection is a type of code injection that uses vulnerability at the database level and allows you to  
execute arbitrary SQL, allowing malicious users to carry out actions such as deleting data or raising their  
privileges.

In this recipe, we will see examples of vulnerable code and fix them.

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. Execute the following SQL:

DROP TABLE IF EXISTS 'user';

CREATE TABLE 'user' (

'id' int(11) unsigned NOT NULL AUTO\_INCREMENT,

'username' varchar(100) NOT NULL,

'password' varchar(32) NOT NULL,

PRIMARY KEY ('id')

);

INSERT INTO 'user'('id','username','password') VALUES (

'1','Alex','202cb962ac59075b964b07152d234b70');

INSERT INTO 'user'('id','username','password') VALUES (

'2','Qiang','202cb962ac59075b964b07152d234b70');

1. Generate a User model using Gii.

How to do it...

1. First, we will implement a simple action that checks whether the username and password that came  
from a URL are correct. Create app/controllers/Sqlcontroller. php:

<?php

namespace app\controllers;

use app\models\User;  
use Yii;

use yii\base\Controller;  
use yii\base\Exception;  
use yii\helpers\ArrayHelper;  
use yii\helpers\Html;

/\*\*

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

class SqlController extends Controller  
{

protected function renderContentByResult($result)

{

if ($result) {

$content = "Success";

} else {

$content = "Failure";

}

return $this->renderContent($content);

}

public function actionSimple()

{

$userName = Yii::$app->request->get('username');

$password = Yii::$app->request->get('password');

$passwordHash = md5($password);

$sql = "SELECT \* FROM 'user'"

." WHERE 'username' = '".$userName."'"

." AND password = '".$passwordHash."' LIMIT |1";

$result = Yii::$app->db->createCommand($sql)->queryOne();

return $this->renderContentByResult($result);

}

}

1. Let’s try to access it using the /sql/simple?username=test&password=test URL. As we are  
   unaware of both the username and password, it will, as expected, print **Failure**.
2. Now try /sql/simple?username=%27+or+%271%27%3D%271%27%3B+--&password=whatever. This  
   time, it lets us in, even though we still don’t know anything about the actual credentials. The  
   decoded part of usernamevalue looks like the following:

' or '1'='1'; --

1. Close the quote so that the syntax stays correct. Add or 'i'='i', which makes the condition  
   always true. Use ; -- to end the query and comment the rest.
2. As no escaping was done, the whole query executed was:

SELECT \* FROM user WHERE username = '' or '1'='1'; --' AND password =

'008c5926ca861023c1d2a36653fd88e2' LIMIT 1;

1. The best way to fix this is to use a prepared statement, as follows:

public function actionPrepared()

{

$userName = Yii::$app->request->get('username');

$password = Yii::$app->request->get('password');

$passwordHash = md5($password);

$sql = "SELECT \* FROM 'user'"

." WHERE 'username' = :username"

." AND password = :password LIMIT 1";

$command = Yii::$app->db->createCommand($sql);

$command->bindValue(':username', $userName);

$command->bindValue(':password', $passwordHash);

$result = $command->queryOne();

return $this->renderContentByResult($result);

}

1. Now check /sql/prepared with the same malicious parameters. This time everything was fine and

we received the **Failure** message. The same principle applies to ActiveRecord. The only  
difference here is that AR uses other syntax:

public function actionAr()

{

$userName = Yii::$app->request->get('username');

$password = Yii::$app->request->get('password');

$passwordHash = md5($password);

$result = User::findOne([

'username' => $userName,

'password' => $passwordHash

]);

return $this->renderContentByResult($result);

}

1. In the previous code, we used the username and password parameters like an array key with a  
   value style. If we had written the previous code by using only the first argument, it would be  
   vulnerable:

public function actionWrongAr()

{

$userName = Yii::$app->request->get('username');

$password = Yii::$app->request->get('password');

$passwordHash = md5($password);

$condition = "'username' = '".$userName." AND 'password' = '".$passwordHash."'";  
$result = User::find()->where($condition)->one();  
return $this->renderContentByResult($result);

}

1. If used properly, prepared statements can save you from all types of SQL injections. Still, there are  
   some common problems:

° You can only bind one value to a single parameter, so if you want to query where in(1, 2,  
3, 4), you will have to create and bind four parameters.

° Prepared statements cannot be used for table names, column names, and other keywords.

1. When using ActiveRecord, the first problem can be solved by adding where, as follows:

public function actionIn()

{

$names = ['Alex', 'Qiang'];

$users = User::find()->where(['username' => $names])->all();

return $this->renderContent(Html::ul(

ArrayHelper::getColumn($users, 'username')

));

}

1. The second problem can be solved in multiple ways. The first way is to rely on active record and  
   PDO quoting:

public function actionColumn()

{

$attr = Yii::$app->request->get('attr');

$value = Yii::$app->request->get('value');

$users = User::find()->where([$attr => $value])->all();

return $this->renderContent(Html::ul(

ArrayHelper::getColumn($users, 'username')

));

}

12. But the most secure way is to use the whitelist approach, as follows:

public function actionWhiteList()

{

$attr = Yii::$app->request->get('attr');

$value = Yii::$app->request->get('value');

$allowedAttr = ['username', 'id'];

if (!in\_array($attr, $allowedAttr)) {

throw new Exception("Attribute specified is not allowed.");

}

$users = User::find()->where([$attr => $value])->all();

return $this->renderContent(Html::ul(

ArrayHelper::getColumn($users, 'username')

));

}

How it works...

The main goal when preventing SQL injection is to properly filter the input. In all cases except table  
names, we have used prepared statements—a feature supported by most relational database servers.  
They allows you to build statements once and then use them multiple times, and they provide a safe way  
of binding parameter values.

In Yii, you can use prepared statements for both Active Record and DAO. When using DAO, it can be  
achieved by using either bindValue or bindParam. The latter is useful when we want to execute multiple  
queries of the same type while varying parameter values:

public function actionBind()

{

$userName = 'Alex';

$passwordHash = md5('password1');

$sql = "INSERT INTO 'user' ('username', 'password') VALUES (:username, :password);";  
// insert first user

$command = Yii::$app->db->createCommand($sql);

$command->bindParam('username', $userName);

$command->bindParam('password', $passwordHash);

$command->execute();

// insert second user  
$userName = 'Qiang';

$passwordHash = md5('password2');

$command->execute();

return $this->renderContent(Html::ul(

ArrayHelper::getColumn(User::find()->all(), 'username')