**Basic Yii Security**

* Validate the user input (see below for details).
* Protect (escape) your application output according to context (see below for a few output types, mostly HTML and SQL).
* Test your application in debug mode.  
  Set the constant YII\_DEBUG to true (by default, it is defined in index.php) and put alongside error\_reporting(E\_ALL);. Then errors and warnings will stop the execution and Yii will display the message, the source code and the call stack. Even an undefined key in an array (which is just a "E\_NOTICE" level) can cause security problems.
* Disable the debug mode in production.  
  Make sure your error messages don't contain sensitive information.
* In production, keep logs. Parse them regularly for warnings and errors.  
  There are two levels of logs : application logs (handled by Yii) and server logs (handled by PHP and usually Apache). Yii logs are described in [The Definitive Guide to Yii, Logging](http://www.yiiframework.com/doc/guide/1.1/en/topics.logging). PHP logs are usually on by default. Please check your server configuration and your rights on the file system for accessing these log files.

**Validating User Input**

#### Validating through a model

* Validating through controller

## HTML output and XSS

* If the application prints unfiltered user input inside a HTML page, then it allows a malicious user to change the display of this page, and to inject client code (usually JavaScript) that can be run by other users. One typical use of these XSS attacks is to steal user sessions.

## SQL Injections

* When some user data is put unfiltered in a SQL query, it allows a malicious user to send its own SQL in the query.

<?php

// warning, dangerous code

Yii::app()->db

->createCommand("DELETE FROM mytable WHERE id = " . $\_GET['id'])

->execute();

$comments = Comment::model->findAll("user\_id = " . $\_GET['id']);

## Cross-site Request Forgery (CSRF)

## Please note that HTTP requests that modify the server state (create, update, delete) should be with the POST protocol. This is a good practice, as recommended by REST, and it helps web browser to prevent accidental re-send of these requests.

## Authorization

Authorization is ensuring users only have access to the resources they have permissions on. This is a lengthy subject, and Yii provides many useful classes to handle permissions and roles. To learn about this, please read The Definitive Guide to Yii from [Access Control Filter](http://www.yiiframework.com/doc/guide/1.1/en/topics.auth#access-control-filter) to [Using Business Rules](http://www.yiiframework.com/doc/guide/1.1/en/topics.auth#using-default-roles).

## Authentication

### Password strength

The validation rule must reject any weak password. Writing its own validation method is easy: just require a minimum size, and check that different classes of characters are present.

### Encrypting passwords

This section considers only internal authentication, i.e. through passwords managed by the application. It does not consider LDAP, SSO, OpenID, or any other external service.

If the authentication process is internal, then of course you shouldn't store the passwords in plain text. The easiest solution for encryption is to use the well-known library [PHPass](http://www.openwall.com/phpass/). With Yii, it can be as simple as the following "User" model:

**Massive assignment**

Once your model's validators have approved all the fields, it's time to make use of the data produced by the form, in bulk. This happens during form submission by calling the controller's action.

Here we'll look at the post/update code:

// protected/controllers/CommentController.php

public function actionUpdate()

{

$model = $this->loadModel();

if (isset($\_POST['Comment']))

{

$model->attributes = $\_POST['Comment']; // Massive Assignment

....

When ever we use massive assignment we must assing the default value in corresponding controller.