**SQL注入漏洞发现与检测**

1. **SQL注入漏洞的定义**

SQL是结构化查询语言（Structured Query Language）的缩写，是专用于数据库查询、更新、管理的语言。CWE将SQL注入漏洞命名为“用于SQL查询的特殊元素的不当过滤”。当Web应用接收用户输入来创建部分或全部SQL查询时，没有过滤或没有正确过滤用户输入中可能会修改原有SQL查询含义的特殊元素即将SQL查询发送给数据库，就会触发SQL注入漏洞。

1. **SQL 注入漏洞的产生原理**

SQL 注入漏洞的形成过程如图 2.7 所示，其产生有五个条件：

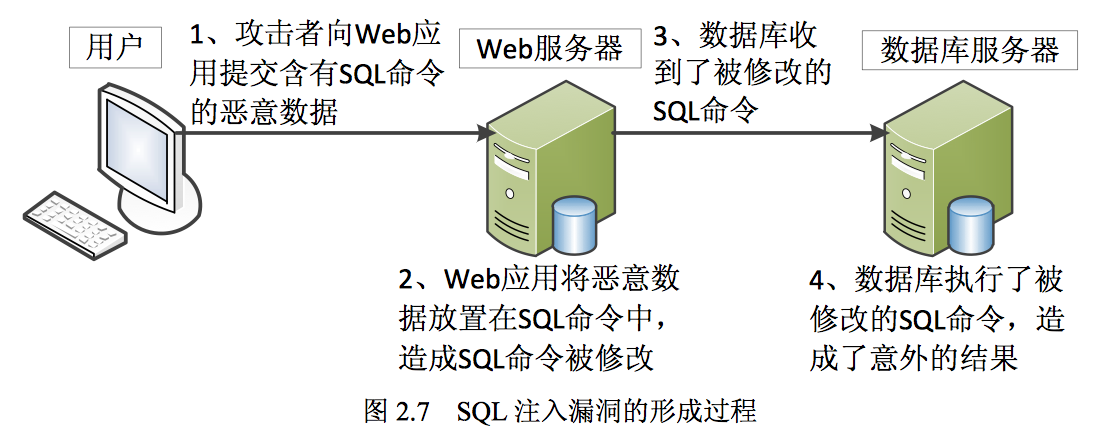
1、来自用户的不受信任数据进入了 Web 应用；

2、不受信任的数据参与了 Web 应用生成 SQL 查询语句的过程；

3、Web 应用在生成 SQL 查询时没有过滤或转义引起歧义的特殊字符；

4、Web 应用将被污染的 SQL 查询语句用于数据库查询；

5、SQL 查询由于特殊字符的存在语义改变，数据库在解析执行 SQL 查询时 执行了设计之外的命令。



比如有一个 ASP 编写的简单应用，使用 MS SQL Server 数据库。该应用需要 对用户提交的用户名 username 和密码 password 进行验证，通常会这样处理：

1:username = request.form("username")

2:password = request.form("password")

3:query="SELECT \* FROM userList WHERE username ='" & username & "' AND password ='"&password&"'"

4:set conn=server.createObject("adodb.connection")

5:conn.open "Provider = SQLOLEDB ; Server = SQL ; Database = DB ; Uid = sa ; Pwd = sa"

6:set rs=server.createObject("adodb.recordset")

7:rs.open query,conn,1,1

……

这段代码的前两行接收了用户提交的username和password，已符合存在SQL注入漏洞的条件1；代码第三行将username和password与SQL查询中的固有部分直接拼接起来构成SQL查询，并且未对这两个参数进行任何过滤处理，符合条件2和3；代码后续部分与数据库建立连接，并将查询提交给数据库，符合条件4。如果一切正常运作，则不会满足条件5，比如正常用户使用时，会输入合法的用户名和密码，此时应用不会出现异常。假设用户名和密码分别是“alice”和“123456”，那么拼接后的SQL查询是：

SELECT \* FROM userList WHERE username = 'alice' AND password = '123456'

该条SQL查询将按照开发者的设定，从userList表中选出用户名为“alice”并且密码为“123456”的用户，应用运转正常。但是攻击者可以通过构造特别的用户名或者密码来操纵SQL查询的拼接结果，从而触发条件5。假设攻击者已经发现此处存在SQL注入漏洞，为了利用该漏洞获取更高的数据库操作权限，攻击者试图在SQLServer中添加新用户。为了达到这一攻击目的，攻击者在用户名处填入“hack';exec sp\_addlogin 'hacker' , 'hacker123'--”，密码处为“123456”，并提交。此时Web应用所拼接成的SQL查询语句为：

SELECT \* FROM userList WHERE username='hack'; exec sp\_addlogin 'hacker' , 'hacker123' --' AND password = '123456'

该条SQL查询被分号“;”和双横线“--”分成了三个部分。分号是SQL中一条查询语句的终结符，即分号前后是两条语句；双横线“--”是单行注释符，其后所有字符直至该行结束都将被解释为注释。所以该条查询语句被分解为：

语句一：SELECT \* FROM userList WHERE username='hack'

语句二：exec sp\_addlogin 'hacker', 'hacker123'

注 释：--' AND password='123456'

语句一和语句二语法都完全正确，因而解释器在执行完第一条语句后会自动执行第二条语句，此时数据库已经执行了开发者设计之外的命令，满足了前文中的条件5。语句二的意义是向SQL Server中添加一个用户名为hacker密码为hacker123的新用户。所以攻击者利用此次攻击成功向SQL Server中添加了新用户，如图2.8所示，图中红框部分即为利用SQL注入添加的数据库用户。如果SQL Server开启了cmdshell功能，攻击者甚至可以操纵cmdshell对SQL Server运行的系统进行操作，比如添加系统管理员账号。

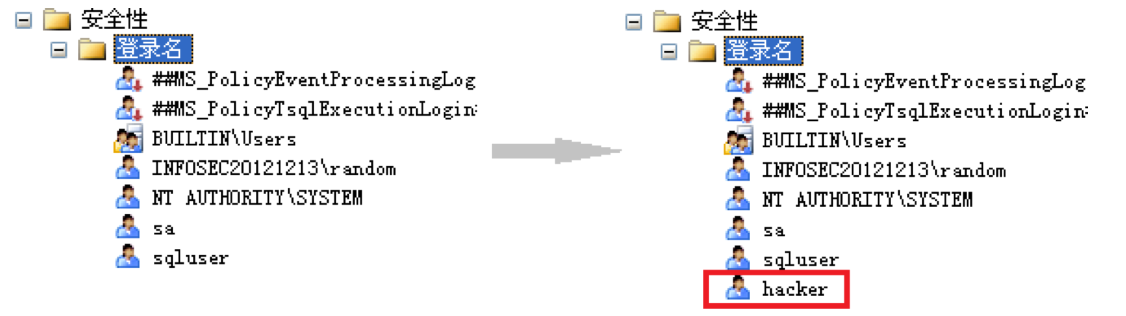


图 2.8 利用 SQL 注入添加数据库用户

1. **SQL 注入漏洞的危害**

SQL注入漏洞危害极为严重。SQL注入漏洞通常会导致被攻击数据库中存储的信息被窃取，甚至被删除、篡改。严重的将导致数据库登录帐户密码被篡改，使得Web应用无法正常运转。最为严重的结果是Web应用所运行的服务器被攻击者完全控制，攻击者利用服务器控制权可以修改Web应用，进行“网站挂马”等恶意活动。

比如在2011年爆发的名为“丽莎月亮”（LizaMoon）的攻击事件，仅仅数天内就利用SQL注入漏洞席卷全球，攻击了超过一百万个网页并在其中留下恶意代码。

1. **基于浏览器引擎的跨站脚本漏洞检测思路**

跨站脚本漏洞的检测效果可以从两方面进行提升，其一是设计好的测试用例，其二是改进漏洞判别方法。换而言之，即是从优化攻击输入和响应分析这两方面入手。本章提出了一种跨站脚本漏洞检测技术，在响应分析阶段加入浏览器引擎，以提高跨站脚本漏洞检测的准确率。

**4.1 反射型跨站脚本漏洞检测思路**

根据跨站脚本漏洞的产生机理，在响应分析阶段判定存在反射型跨站脚本漏洞需要确定三个条件，分别是输入参数的反射性、攻击代码的反射完整性和攻击代码的可执行性。

（1）输入参数的反射性

输入参数反射性是指经由当前测试的参数入口点进入的参数确实能够被回显在用户的浏览器上，这是反射性跨站脚本漏洞存在的先决条件。如果该条件不成立，则说明该参数点不存在反射型跨站脚本漏洞。

（2）攻击代码的反射完整性

该条件成立与否取决与攻击代码在经过反射后结构是否仍然完整。由于Web应用通常都有一定的防御机制，所以即使攻击代码可以被反射，但可能其中的某些关键字符被应用过滤掉了，破坏了代码原有的正确结构。假设某Web应用中使用了下面这段PHP代码，该段代码的功能仍然是将用户的用户名进行回显，但是相比于第二章中的“welcome.php”多出了使用htmlspecialchars函数对收到的参数进行处理的步骤。

<?php

$uname = $\_GET['uname'];

$unameF = htmlspecialchars($uname);

echo "Welcome, $unameF!";

?>

对其提交测试参数“uname=<script>alert('xss')</script>”，浏览收到的响应如图 4.1 所示，脚本未被浏览器执行。

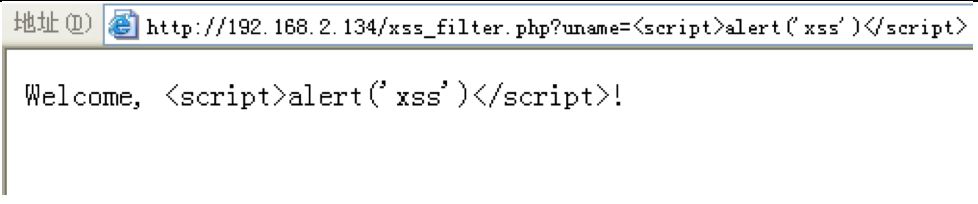


图 4.1脚本未被浏览器执行

将该网页源代码（如图4.2所示）与不加过滤步骤的网页源代码（如图4.3所示）进行对比可以发现，“<”和“>”这两个字符被htmlspecialchars函数转换为了HTML实体，“<”从原本的“%3C”变成了“&lt;”，“>”从“%3E”变成了“&gt;”。这两个字符被成功过滤，因而攻击代码虽然被反射，但是结构被破坏，无法被作为脚本执行。

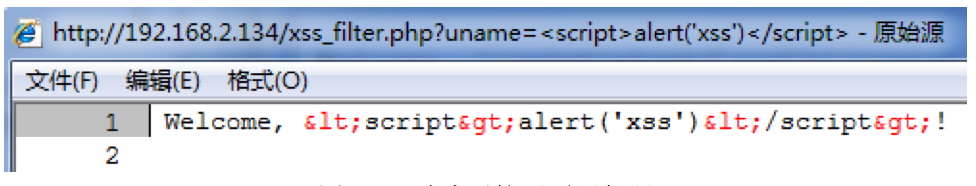


图 4.2 过滤后的网页源代码

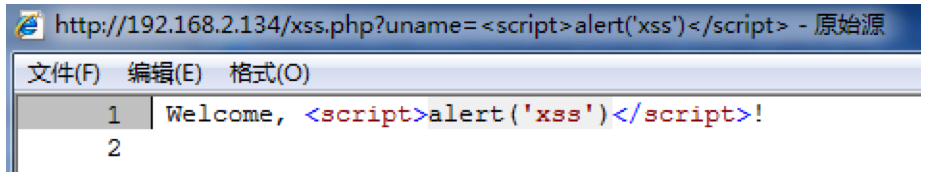


图 4.3 未过滤的网页源代码

所以在确定了参数的反射性后，需要测试Web应用在该参数点设置的过滤机制是否强健。如果所有攻击代码经过反射后都失去了原有的正确结构，那么此次检测未能发现反射型跨站脚本漏洞。

（3）攻击代码的可执行性

在确定了前两个条件后，为了降低误报率，需要确定页面中的攻击代码确实是可以被浏览器执行。攻击者不能控制攻击代码被反射后的位置，如果攻击代码被反射在网页中的数据部分（如双引号之间），即使有正确的结构，也无法被浏览器执行。比如下面这段PHP代码，对于从用户处得到的参数未进行任何形式的过滤。

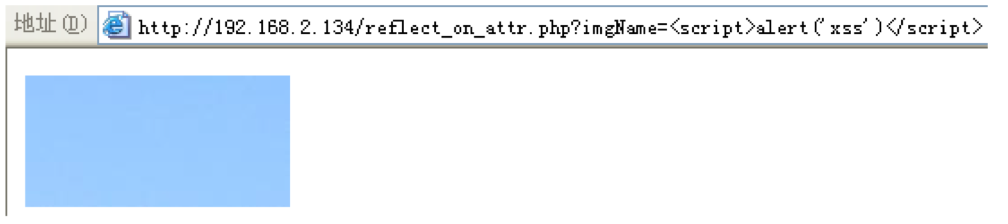
<?php

$imgName = $\_GET['imgName'];

echo '<img src="blue.jpg" name="'.$imgName.'"/>';

?>

向其提交测试用例“imgName=<script>alert('xss')</script>”，所得网页如图4.4上半部分所示，源代码在下半部分。从源代码部分可以看出攻击代码被完全的反射在了网页上，但是在网页中并未观测到有告警框弹出，说明攻击代码没有被执



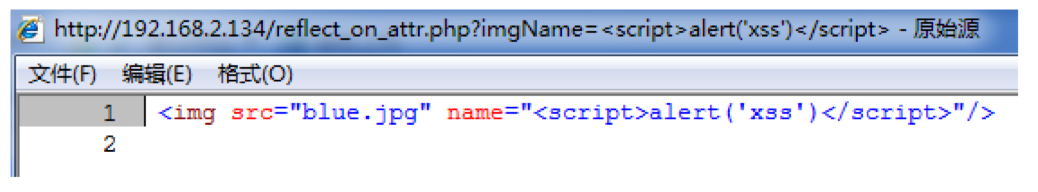


图 4.4 攻击代码被反射在 HTML 数据部分

**4.2 存储型跨站脚本漏洞检测思路**

依据对存储型跨站脚本漏洞原理的分析，可以得出与反射型跨站脚本漏洞相类似的三个条件：输入参数存储后可再用性、攻击代码存取完整性以及攻击代码的可执行性。

首先，由于存储型跨站脚本漏洞的攻击代码将会被数据库存储，所以当其再次被取出时才是漏洞的触发时机。因而条件一是为了确保用户向该入口提交的数据将会进入数据库并且会在某时刻根据用户的请求而被取出放在网页上。

其次，同样是由于Web应用防御机制的存在，需要确定攻击代码再次出现在网页上时结构是否完整，亦即Web应用在数据库存取的过程中没有对攻击代码进行足够的过滤处理。

最后，为了降低误报，同样需要确定页面上的攻击代码确实能够被执行。

由于存储型跨站脚本漏洞的形成涉及到Web应用的表示层、逻辑层、数据层全三层，并且攻击所产生的效果无法即时获得，所以检测方法较为复杂，主要有两点问题：问题一是响应观测点不确定，由于数据库参与了漏洞的形成，参数的入口和出口通常不在同一位置，导致无法事先确定攻击代码再次出现的位置，从而无法直接观察所提交测试用例对应的响应；问题二是测试用例的驻留，由于测试代码在存储型跨站脚本漏洞触发成功时会驻留在Web应用中，对Web应用造成损伤，因此不能采用损害性较大的攻击代码。

**4.3 基于浏览器引擎的跨站脚本漏洞检测思想**

无论是反射型XSS还是存储型XSS，在其检测条件中都需要对攻击代码的可执行性进行验证。就这一验证过程而言，如果检测过程是由测试人员人工操作，则只需要对浏览器的行为进行观察即可。根据前文的论述，如果存在跨站脚本漏洞，攻击者可以通过注入脚本操控用浏览器的行为。假设在安全测试中，测试人员使用JavaScript的alert()函数（该函数会在浏览器中弹出一个告警框）作为测试脚本进行反射型跨站脚本漏洞检测，那么一旦测试人员观察到告警框，则可以确定存在反射型跨站脚本漏洞。但是现有检测算法无法直接观察注入脚本后的浏览器行为，因此本章提出了基于浏览器引擎的跨站脚本漏洞检测算法来解决这一问题。

浏览器引擎是浏览器的核心。使用浏览器引擎，将可以达到对浏览器行为进行监控的效果。譬如控制浏览器弹出告警框的alert函数，在浏览器引擎中将作为一个变量存在。因此本算法使用浏览器引擎处理注入攻击代码后的Web应用响应，通过观测其变量变化以确定跨站脚本漏洞的存在。

浏览器引擎的引入，将消除跨站脚本漏洞检测的误报现象。这是由于跨站脚本漏洞检测的落脚点在于浏览器的行为，在形成了对浏览器行为的准确监控后，将可以消除误判。

1. **小结**

本文对跨站脚本漏洞的检测进行了研究，提出了基于浏览器引擎的跨站脚本 漏洞检测技术。首先分析了该技术的检测思路，即跨站脚本漏洞的检测条件。为 了在检测中实现对这些条件的验证，针对反射型和存储型两类跨站脚本漏洞分别 设计了检测算法和测试用例。