PHP SQL注入攻击与防护解析

SQL注入是Web安全领域最常见最具破坏性的漏洞之一，而PHP作为Web开发的主流语言，因历史版本兼容、开发习惯等因素，成为SQL注入攻击的高发区。本文由”守阙安全团队“制作，从PHP中典型的SQL注入场景入手，详解攻击者的注入手段与技巧，并结合实战案例给出系统化的防护方案。

一、PHP中SQL注入的典型场景与攻击手法

SQL注入的本质是攻击者通过构造恶意输入，篡改SQL语句的逻辑，从而实现未授权的数据访问、修改甚至服务器控制。以下是PHP中最常见的注入场景及具体攻击方式：

1.1 登录验证绕过（字符串拼接注入）

场景：登录功能中直接拼接用户输入的用户名/密码到SQL语句。

漏洞代码：

Code block

```php
<?php
$conn = mysqli_connect("localhost", "root", "root", "test");
$username = $_POST['username'];
$password = $_POST['password'];

// 直接拼接用户输入（高危）
$sql = "SELECT * FROM users WHERE username='$username' AND
password='$password'";
$result = mysqli_query($conn, $sql);

if (mysqli_num_rows($result) > 0) {
    echo "登录成功! ";
} else {
    echo "用户名或密码错误";
}
?>
```

攻击手法：

攻击者在用户名输入框输入：`admin' --`，此时拼接后的SQL语句变为：

Code block

```
SELECT * FROM users WHERE username='admin' --' AND password='任意值'
```

`--` 是SQL的注释符，会注释掉后续的 `AND password='任意值'` ，SQL逻辑变为“仅查询用户名=admin的用户”，直接绕过密码验证。

若输入： `' OR 1=1 --` ，SQL语句变为：

Code block

```
SELECT * FROM users WHERE username='' OR 1=1 --' AND password='任意值'
```

`1=1` 永远为真，将返回所有用户数据，甚至直接登录系统。

1.2 URL参数注入（数字型注入）

场景：通过URL传递ID等参数查询数据，未对参数类型做校验。

漏洞代码：

Code block

```
<?php
$conn = mysqli_connect("localhost", "root", "root", "test");
$id = $_GET['id']; // 直接获取URL中的id参数

// 数字型参数未加引号拼接
$sql = "SELECT * FROM articles WHERE id=$id";
$result = mysqli_query($conn, $sql);
$article = mysqli_fetch_assoc($result);
echo "文章标题: " . $article['title'];
?>
```

攻击手法：

攻击者构造URL： `article.php?id=1 OR 1=1` ，拼接后的SQL语句变为：

Code block

```
SELECT * FROM articles WHERE id=1 OR 1=1
```

将返回所有文章数据。若构造： `article.php?id=1 UNION SELECT 1,username,password FROM users` ，可直接窃取用户表中的账号密码（需保证前后查询字段数一致）。

1.3 搜索功能注入（模糊查询注入）

场景：搜索功能中拼接用户输入的关键词到 `LIKE` 语句。

漏洞代码：

Code block

```php
<?php
$conn = mysqli_connect("localhost", "root", "root", "test");
$keyword = $_GET['keyword'];

$sql = "SELECT * FROM goods WHERE name LIKE '%$keyword%'";
$result = mysqli_query($conn, $sql);
?>
```

攻击手法：

攻击者输入：`%' OR 1=1 --`，拼接后的SQL语句变为：

Code block

```sql
SELECT * FROM goods WHERE name LIKE '%%' OR 1=1 --%'
```

`OR 1=1` 会返回所有商品数据，甚至可进一步注入 `UNION` 语句获取其他表信息。

1.4 Cookie/Header注入（信任客户端数据）

场景：应用信任Cookie、HTTP头（如User-Agent）等客户端可控数据，并拼接至SQL语句。

漏洞代码：

Code block

```php
<?php
$conn = mysqli_connect("localhost", "root", "root", "test");
$user_id = $_COOKIE['user_id']; // 从Cookie获取用户ID

$sql = "SELECT * FROM users WHERE id='$user_id'";
$result = mysqli_query($conn, $sql);
?>
```

攻击手法：

攻击者修改Cookie中的 `user_id` 为：`1' OR '1'='1`，SQL语句变为：

Code block

```sql
SELECT * FROM users WHERE id='1' OR '1'='1'
```

可获取所有用户信息。

1.5 二次注入（存储型注入）

场景：用户输入先被存入数据库，后续读取时未过滤，导致注入。

漏洞代码：

步骤1：注册用户时存入恶意数据

Code block

```php
<?php
$conn = mysqli_connect("localhost", "root", "root", "test");
$username = $_POST['username']; // 攻击者输入：admin' OR '1'='1
$password = $_POST['password'];

$sql = "INSERT INTO users (username, password) VALUES ('$username', '$password')";
mysqli_query($conn, $sql);
?>
```

步骤2：后续查询时触发注入

Code block

```php
<?php
$conn = mysqli_connect("localhost", "root", "root", "test");
$username = $_GET['username'];

$sql = "SELECT * FROM users WHERE username='$username'";
$result = mysqli_query($conn, $sql);
?>
```

攻击手法：

攻击者注册时用户名输入 `admin' OR '1'='1`，存入数据库后，当其他功能查询该用户名时，SQL语句变为：

Code block

```sql
SELECT * FROM users WHERE username='admin' OR '1'='1'
```

触发注入并返回所有用户数据。

二、PHP中SQL注入的系统化防护方案

防护SQL注入的核心原则是**“分离SQL语句结构与用户输入数据”**，杜绝直接拼接用户输入。以下是从基础到进阶的防护手段：

2.1 核心防护：参数化查询（Prepared Statement）

参数化查询是防范SQL注入的“银弹”，其原理是先预编译SQL模板（含占位符），再将用户输入作为参数传入，数据库会将参数视为纯数据而非SQL代码执行。

（1）MySQLi扩展的参数化查询

Code block

```php
<?php
$conn = new mysqli("localhost", "root", "root", "test");
if ($conn->connect_error) {
    die("连接失败: " . $conn->connect_error);
}

$username = $_POST['username'];
$password = $_POST['password'];

// 1. 准备SQL模板（?为占位符）
$stmt = $conn->prepare("SELECT * FROM users WHERE username=? AND password=?");
// 2. 绑定参数（ss表示两个字符串类型参数，i=整数，d=浮点数，b=二进制）
$stmt->bind_param("ss", $username, $password);
// 3. 执行查询
$stmt->execute();
// 4. 获取结果
$result = $stmt->get_result();

if ($result->num_rows > 0) {
    echo "登录成功! ";
} else {
    echo "用户名或密码错误";
}

$stmt->close();
$conn->close();
?>
```

（2）PDO扩展的参数化查询（支持多数据库）

Code block

```php
<?php
try {
```

```php
    $pdo = new PDO("mysql:host=localhost;dbname=test;charset=utf8mb4", "root", "root");
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION); // 开启异常模式

    $username = $_POST['username'];
    $password = $_POST['password'];

    // 方式1: 匿名占位符 (?)
    $stmt = $pdo->prepare("SELECT * FROM users WHERE username=? AND password=?");
    $stmt->execute([$username, $password]);

    // 方式2: 命名占位符 (更易读)
    // $stmt = $pdo->prepare("SELECT * FROM users WHERE username=:username AND password=:password");
    // $stmt->execute([':username' => $username, ':password' => $password]);

    $user = $stmt->fetch(PDO::FETCH_ASSOC);
    if ($user) {
        echo "登录成功! ";
    } else {
        echo "用户名或密码错误";
    }
} catch(PDOException $e) {
    echo "错误: " . $e->getMessage();
}
?>
```

2.2 辅助防护：输入验证与过滤

参数化查询是核心，但输入验证可进一步降低风险，尤其是对参数类型、格式的校验：

（1）类型校验

对数字型参数强制转换为整数：

Code block

```php
$id = intval($_GET['id']); // 确保id为整数，即使注入恶意字符也会被转为数字
$sql = "SELECT * FROM articles WHERE id=$id"; // 此时拼接也安全
```

（2）白名单过滤

对固定值参数（如状态、类型）使用白名单：

```php
$status = $_GET['status'];
$allowed_status = ['active', 'inactive', 'deleted'];
if (!in_array($status, $allowed_status)) {
    die("非法参数! ");
}
$sql = "SELECT * FROM orders WHERE status='$status'";
```

（3）特殊字符转义（应急方案）

若无法使用参数化查询（如老旧项目），可使用 `mysqli_real_escape_string` 转义特殊字符（仅适用于字符串参数）：

Code block

```php
$username = mysqli_real_escape_string($conn, $_POST['username']);
$password = mysqli_real_escape_string($conn, $_POST['password']);
$sql = "SELECT * FROM users WHERE username='$username' AND
password='$password'";
```

⚠ 注意：该方法仅能转义单引号、双引号等字符，无法防御数字型注入，且依赖字符集设置，不能替代参数化查询。

2.3 进阶防护：使用ORM框架

ORM（对象关系映射）框架（如Eloquent、Doctrine）会自动处理SQL拼接，避免手动注入风险：

Laravel框架的Eloquent ORM示例

Code block

```php
<?php
// 查询用户，自动参数化
$user = DB::table('users')->where('username', $_POST['username'])
                          ->where('password', $_POST['password'])
                          ->first();

if ($user) {
    echo "登录成功! ";
}
?>
```

2.4 基础防护：最小权限原则

数据库账号应配置最小权限，避免攻击者通过注入获取过高权限：

- 应用使用的数据库账号仅授予 `SELECT` / `INSERT` / `UPDATE` 等必要权限，禁止 `DROP` / `ALTER` 等危险操作；
- 禁止使用root、sa等超级管理员账号连接数据库。

2.5 审计与监控：日志记录与漏洞扫描

- 记录SQL执行日志，监控异常SQL语句（如包含 `UNION` 、 `DROP` 、 `OR 1=1` 的语句）；
- 定期使用SQL注入扫描工具（如SQLMap、AWVS）检测漏洞，及时修复。

三、总结

PHP中的SQL注入攻击多源于“直接拼接用户输入”的开发习惯，攻击者通过构造恶意参数篡改SQL逻辑，实现数据泄露或权限绕过。防护的核心是**参数化查询**，辅以输入验证、ORM框架、最小权限配置等手段，形成多层防护体系。

开发者需牢记：**永远不要信任用户输入**，任何客户端传递的数据都必须经过校验和隔离处理，才能从根本上杜绝SQL注入漏洞。