《安全程序设计与实践》
实验报告

实验名称：更新密码——二阶注入攻击

实验类型：实操型

指导教师：孙海峰

专业班级：

姓　　名：

学　　号：

实验地点：东 11C227

实验日期：2025/5/7

一、 实验目的

1. 深入理解二阶 SQL 注入攻击原理
2. 掌握二阶 SQL 注入攻击测试方法
3. 二阶 SQL 注入攻击防护技术

二、 实验过程

任务一：建立具有密码更新功能的网站。

1-1 任务实现

通过修改密码更新功能的代码，构造一个存在二阶注入隐患的场景。

首先先将 updatepasswd.php 中

$username = mysqli_real_escape_string($con,$_SESSION['username']);

更改为$username = $_SESSION['username'];

修改后直接从$_SESSION 获取未转义的用户名，若会话中存储的用户名包含恶意字符（如 admin'#），后续 SQL 拼接时会破坏查询逻辑，从而为后续二阶注入埋下隐患。

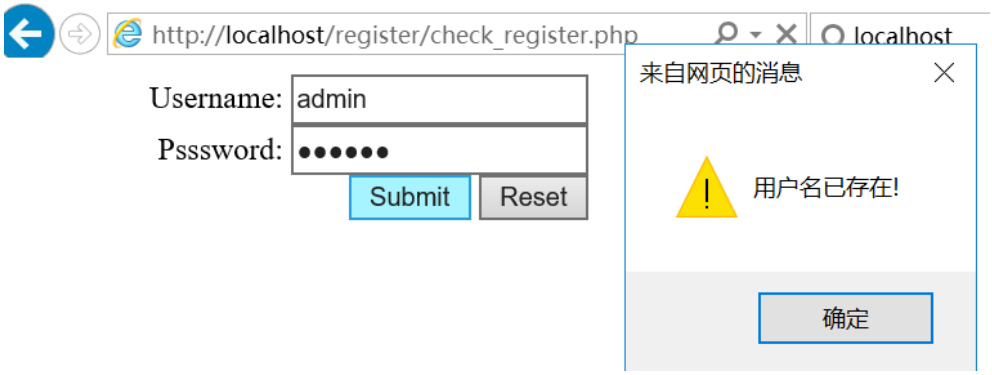
```php
if (isset($_POST['Submit'])){
    //包含数据库连接
    include("con_database.php");

    $username = $_SESSION['username'];
    //$username = mysqli_real_escape_string($con,$_SESSION['username']);
    $curr_pass= mysqli_real_escape_string($con,$_POST['current_passwd']);
    $pass= mysqli_real_escape_string($con,$_POST['passwd']);
    $sql = "UPDATE users SET passcode = '$pass' WHERE username = '$usernar
    $res = mysqli_query($con,$sql) or die('SQL执行失败 :'.mysqli_error($con
    $row = mysqli_affected_rows($con);
```

将 updatepasswd.html 中 action 更改为 action="updatepasswd.php"

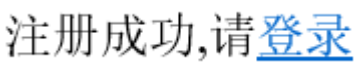
```
<form name="form_register" method="post" action="updatepasswd_mysqli.php">
```

1-2 测试过程

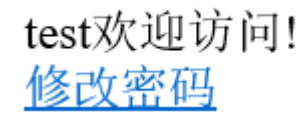
注册数据库已存在的用户，输入用户名 admin，提示 “用户名已存在”，验证注册功能的唯一性校验正常。



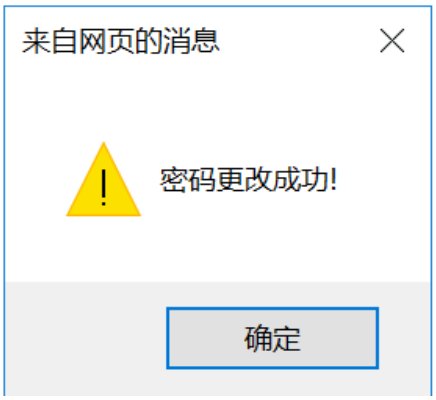
输入用户名 test、密码 123456，注册成功后登录。



登录完成后进入密码修改页面。



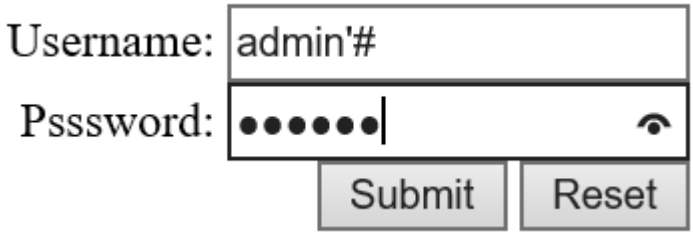
输入当前密码 123456 和新密码 123，提示 “密码更改成功”，验证正常流程功能。



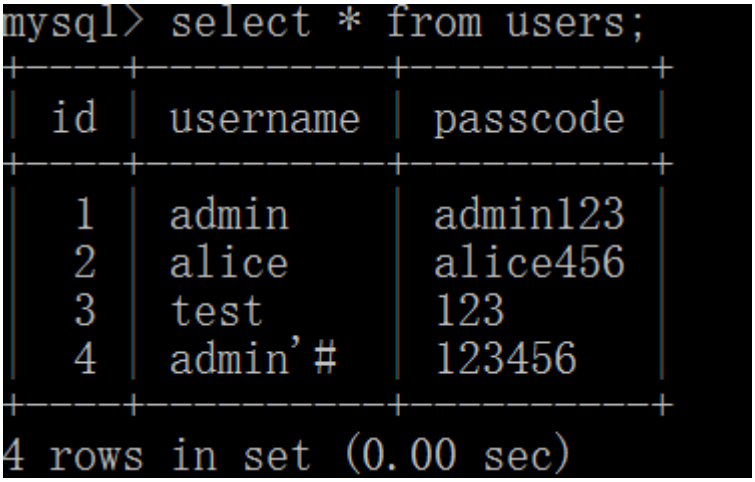
任务二：二阶注入攻击测试

2-1 测试过程

输入用户名 admin'#、密码 123456，注册成功。



数据库存储结果：



```
mysql> select * from users;
+----+----------+----------+
| id | username | passcode |
+----+----------+----------+
|  1 | admin    | admin123 |
|  2 | alice    | alice456 |
|  3 | test     | 123      |
|  4 | admin'#  | 123456   |
+----+----------+----------+
4 rows in set (0.00 sec)
```

用 admin'#登录成功，进入密码修改页面。

admin'#欢迎访问!
修改密码

输入当前密码 123456，新密码 admin，提交更新。

发现数据库中 admin 用户的密码被修改为 admin：

```
mysql> select * from users;
+----+----------+----------+
| id | username | passcode |
+----+----------+----------+
|  1 | admin    | admin    |
|  2 | alice    | alice456 |
|  3 | test     | 123      |
|  6 | admin'#  | 123456   |
+----+----------+----------+
4 rows in set (0.00 sec)
```

用密码 admin 登录 admin 账户，成功登录，证明攻击生效。

admin欢迎访问!
修改密码

2-2 测试分析

二阶注入的核心在于“存储-触发”机制：

注册用户名为 admin'#，其中'用于闭合 SQL 字符串，#用于注释后续内容。

因使用转义函数，实际执行的 SQL 为：

INSERT INTO users (username, passcode) VALUES ('admin\'#', '123456');

因此注册和登录都不会破坏 sql 语句结果，admin’#正常存储进入数据库。

但是 updatepasswd.php 中，没有使用转义函数，系统从数据库读取 admin'#作为$username，更新 admin’#密码时语句为：

UPDATE users SET passcode='admin' WHERE username='admin'#' and passcode='123456';

#注释后续内容，实际执行：

UPDATE users SET passcode='admin' WHERE username='admin';

因此更新的为 admin 密码。

任务三：二阶注入攻击防护

3-1 使用 PHP 转义函数：

通过 mysqli_real_escape_string 转义$username 中的特殊字符，确保 SQL 拼接时恶意字符被转义为字面量，无法破坏查询逻辑。

我们在更新密码（ipdatepasswd.php）中使用转义函数，

```php
$username = mysqli_real_escape_string($con,$_SESSION['username']);
```

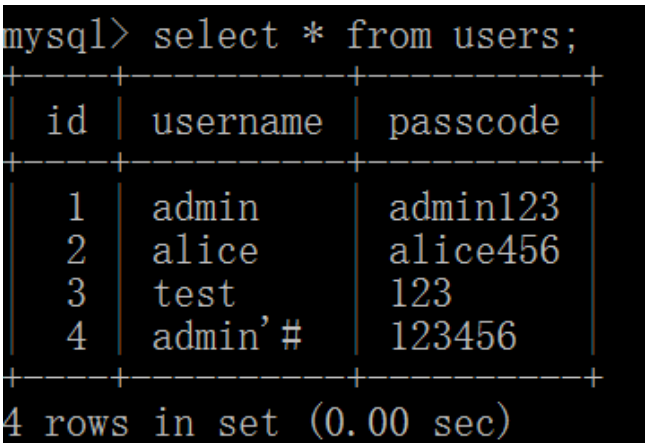
这样我们对 admin’#更新密码时’会被转义为/’，#就不会绕过单引号，不会注释掉后续内容。

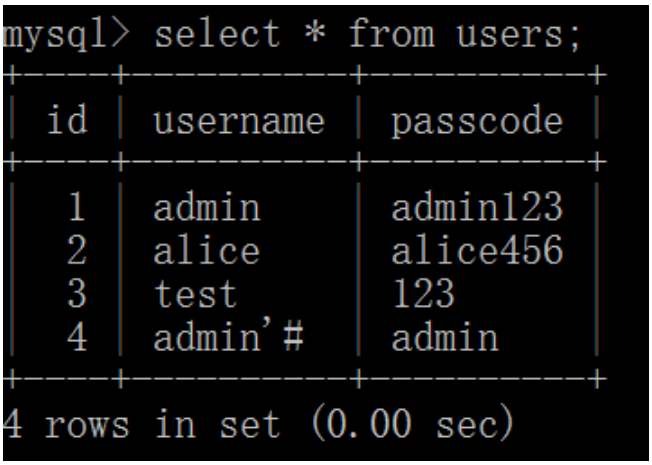
我们进行测试，更改 admin'#密码为 admin。

实际执行 UPDATE users SET passcode='new_pass' WHERE username='admin\'#' and passcode='admin';

#作为普通字符存在，无法注释后续内容，因此仅更新 admin'#用户的密码，admin 用户不受影响。

数据库验证：





admin'#密码变了，admin 密码没变，达到防护作用。

3-3 MySQLi 参数化更新

参数化查询通过预编译 SQL 语句，将用户输入作为参数绑定，彻底分离数据与 SQL 逻辑，避免恶意输入篡改查询结构。

```php
if (isset($_POST['Submit'])){
    //包含数据库连接
    include('con_database.php');

    $username = $_SESSION['username'];
    $curr_pass= $_POST['current_passwd'];
    if($curr_pass == '') exit("旧密码不能为空");
    $pass= $_POST['passwd'];
    if($pass == '') exit("新密码不能为空");
    if($curr_pass == $pass) exit("密码相同");

    $sql = "UPDATE users SET passcode = ? WHERE username = ? and passcode = ? ";
    $stmt = $con->prepare($sql);
    if (!$stmt) exit("prepare 执行错误");
    $stmt->bind_param("sss",$pass, $username, $curr_pass);
    $stmt->execute();

    if($stmt->affected_rows >0)
    {
        echo "<script>alert('密码更改成功!'); history.go(-1);</script>";
    }
    else
    {
        echo "<script>alert('当前密码错误!'); history.go(-1);</script>";
    }
    $stmt->close();
}
mysqli_close($con);
```

将 updatepasswd.html 中 action 定向为 updatepasswd_mysqli.php

```
<form name="form_register" method="post" action="updatepasswd_mysqli.php">
```

进行测试：

更新 admin'#，参数化查询将$username 作为纯数据处理，SQL 逻辑固定为：

UPDATE users SET passcode=? WHERE username=? AND passcode=?；

恶意字符无法改变查询结构，仅更新 admin'#用户的密码，admin 用户密码保持不变。

数据库验证：

```
mysql> select * from users;
+----+----------+----------+
| id | username | passcode |
+----+----------+----------+
|  1 | admin    | admin    |
|  2 | alice    | alice456 |
|  3 | test     | 123      |
|  6 | admin'#  | 123456   |
+----+----------+----------+
4 rows in set (0.00 sec)
```

```
mysql> select * from users;
+----+----------+----------+
| id | username | passcode |
+----+----------+----------+
|  1 | admin    | admin    |
|  2 | alice    | alice456 |
|  3 | test     | 123      |
|  6 | admin'#  | abc      |
+----+----------+----------+
4 rows in set (0.00 sec)
```

admin'#密码变了，admin 密码没变，达到防护作用。

三、 实验结论

成功复现二阶 SQL 注入攻击，证实了攻击者可利用应用程序对数据处理的差异，绕过常规防护手段实现攻击。

实践了 PHP 转义函数、MySQLi 参数化查询防护方法。PHP 转义函数通过对特殊字符转义，有效阻止了攻击语句的执行；参数化查询技术则从根本上分离数据与代码，将用户输入作为参数传递，彻底杜绝了 SQL 注入风险。经测试，这些防护措施均能有效抵御二阶 SQL 注入攻击，保障了密码更新功能的安全性。

这要求在软件开发过程中，对涉及用户输入与数据库交互的操作进行严格的安全设计与审查，采用参数化查询等安全编程方式，从源头防范安全漏洞。

四、 思考题

1. 如何使用参数化查询来防护二阶 SQL 注入攻击？

   PDO 参数化查询通过以下机制防护二阶注入：

   预编译阶段：SQL 语句中的占位符（:new_pass、:user）被数据库预编译，确定查询结构，后续无法被恶意输入修改。

   参数绑定：用户输入作为参数绑定到占位符，无论内容如何（包括'、#），均作为纯数据处理，不影响 SQL 逻辑。

```php
$sql = "UPDATE users SET passcode = :new_pass WHERE username = :user and passcode = :curr_pass";
$stmt = $con->prepare($sql);
if (!$stmt) {
    die('prepare 执行错误');
} else {
    // 绑定参数
    $stmt->bindParam(':new_pass', $pass, PDO::PARAM_STR);
    $stmt->bindParam(':user', $username, PDO::PARAM_STR);
    $stmt->bindParam(':curr_pass', $curr_pass, PDO::PARAM_STR);
    $stmt->execute();

    if ($stmt->rowCount() > 0) {
        echo "<script>alert('密码更改成功!'); history.go(-1);</script>";
    } else {
        echo "<script>alert('当前密码错误!'); history.go(-1);</script>";
    }
    $stmt->closeCursor();
}
```

将密码 abc 更改为 123456，成功修改。
数据库验证：

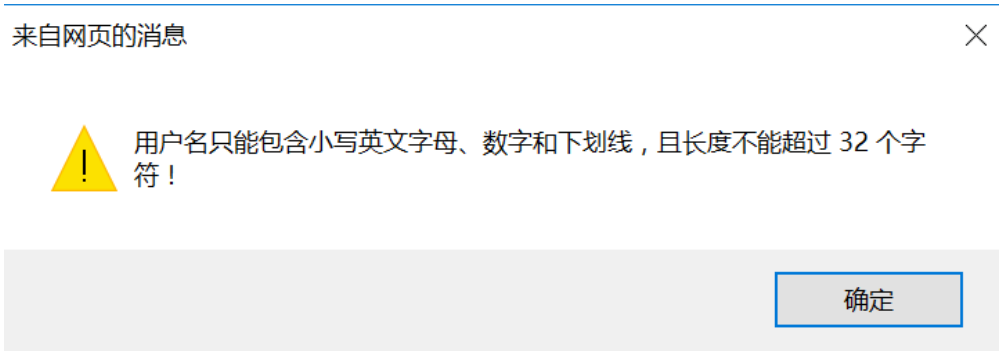
```
mysql> select * from users;
+----+----------+----------+
| id | username | passcode |
+----+----------+----------+
|  1 | admin    | admin    |
|  2 | alice    | alice456 |
|  3 | test     | 123      |
|  6 | admin'#  | abc      |
+----+----------+----------+
4 rows in set (0.00 sec)
```

```
mysql> select * from users;
+----+----------+----------+
| id | username | passcode |
+----+----------+----------+
|  1 | admin    | admin    |
|  2 | alice    | alice456 |
|  3 | test     | 123      |
|  6 | admin'#  | 123456   |
+----+----------+----------+
4 rows in set (0.00 sec)
```

2. 如何在 check_register.php 中利用正则表达式实现对注册的用户名限制？要求为：只能使用小写英文字母、数字和下划线，长度不能超过 32 个字符。
先制定正则表达式规则：$pattern = '/^[a-z0-9_]{1,32}$/';
表示使用小写英文字母、数字和下划线，长度 1-32 个字符。
然后使用 preg_match 进行匹配规则即可。

```php
$pattern = '/^[a-z0-9_]{1,32}$/';
if (!preg_match($pattern, $username)) {
    echo "<script>alert('用户名只能包含小写英文字母、数字和下划线，且长度不能超过 32 个字符！'); history.go(-1);</script>";
    exit;
}
```

输入 1!（含非法字符!）或 11111111111111111111111111111111111111111（长度 33）时，弹出提示并阻止注册。



3. 还有哪些情况可能出现二阶 SQL 注入攻击？

- 用户资料修改：如果只是简单地将用户输入的数据存储到数据库，在后续的其他操作中使用这些数据来构造 SQL 语句，且未对数据进行严格的过滤和安全处理，就会导致二阶 SQL 注入。
- 评论或留言功能：用户在网站上发表评论或留言，这些内容通常会存储在数据库中。如果在展示评论或进行相关数据分析时，使用未经过安全处理的评论内容来构造 SQL 语句，攻击者可以在评论中插入恶意 SQL 代码，从而实现二阶 SQL 注入。
- 文件上传功能：允许用户上传文件并将文件相关信息存储在数据库中。如果在后续的文件管理操作中，使用了这些存储在数据库中的文件信息来构造 SQL 语句，且没有对其进行安全验证，攻击者可以通过上传包含恶意 SQL 代码的文件名，在后续操作中触发 SQL 注入。
- 日志记录与查询：应用程序记录用户的操作日志并存储在数据库中。当管理员或其他功能模块查询这些日志信息时，如果日志内容在 SQL 查询语句中被直接使用，而没有进行安全处理，攻击者可以通过构造特殊的操作行为，使日志记录包含恶意 SQL 代码，进而在查询日志时引发二阶 SQL 注入攻击。
- 数据导入与导出：在数据导入功能中，用户可能上传包含大量数据的文件，应用程序将这些数据解析后存储到数据库。如果在后续的数据库操作中，使用了这些导入的数据来构造 SQL 语句，且未对数据进行严格检查，攻击者可以在导入的数据中嵌入恶意 SQL 代码。