# 常见Web漏洞演示 实验报告

## 尝试使用防御方法，防范XSS攻击

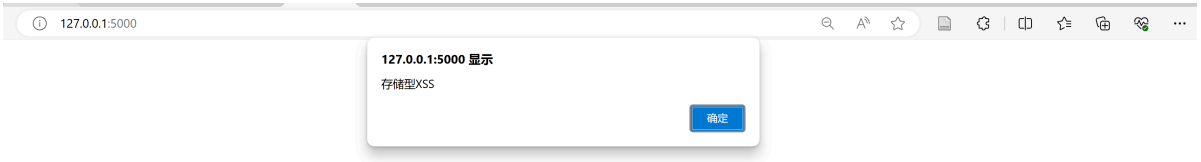
XSS（跨站脚本攻击）是指攻击者将恶意脚本注入到网页中，然后在用户的浏览器中执行。一旦用户浏览受感染的页面，这些恶意脚本就会在用户的浏览器中执行，可能导致用户会话被劫持、个人信息被窃取，甚至在用户不知情的情况下执行其他恶意操作。

XSS攻击分为两种：

1. 存储型XSS：攻击者将恶意脚本注入到数据库中，当用户访问到对应数据的时候，执行恶意脚本
2. 反射型XSS：攻击者通过欺骗用户向网站发送特定的的内容，网站将恶意脚本作为响应的一部分返回给用户的浏览器，最终在用户的浏览器中执行。

### 存储型XSS

将 `<script>alert('存储型XSS')</script>` 作为评论发到网站中以后，再浏览评论，此时前端会把该攻击脚本作为“评论”展示出来，而在html中，这条评论被视作可自动执行的脚本，跳出相应提示




### 反射型XSS

将 `<script>alert('反射型XSS')</script>` 作为查询评论发送后，在前端html页码中 `<h3>包含 "{{ search_query }}" 评论如下:</h3>` 会包含查询内容，而浏览器会把它解析为脚本执行，立即显示"反射型XSS"



### 防御措施

通过在前端使用 Jinja2 的 escape 函数来转义HTML标签，可以强制将内容以字符串方式展现，而不作为html代码执行

```
<!-- Comments -->
{% if not search_query %}
  <h3>所有的评论如下:</h3>
{% else %}
  <h3>包含 "{{ search_query | escape  }}" 评论如下:</h3>
{% endif %}

{% for comment in comments %}
  <div>
    <p>{{ comment | escape }}</p>
  </div>
{% endfor %}
```

效果展示:

可以看到，html转义确保了两种XSS攻击注入的html脚本无法执行，起到了防御XSS攻击的效果


# 增加一个登陆功能，设计有SQL注入隐患的代码，进行攻击。并且展示如何进行防范

首先我为网站增加了登录功能，函数设计如下，其整体思路为首先确定用户在数据库中，然后再验证用户名和其密码的正确性

```
def login(user_id, password):
    # 连接数据库
    print(f"user_id:{user_id}")
    db = connect_db()
    sql_request = f"SELECT * FROM users WHERE user_id = '{user_id}'"
    print("sql_request:", sql_request)
    existing_user = db.cursor().execute(sql_request).fetchone()
    # print(f"existing_user={existing_user}")

    # 检查用户名是否已存在
    if existing_user:
        sql_request = f"SELECT * FROM users WHERE user_id = '{user_id}' AND password = '{password}'"
        print("sql_request:", sql_request)
        comment = db.cursor().execute(sql_request).fetchone()
        if comment:
            print("登录成功",comment)
            return f"Welcome back!{comment[0]}"
        else:
            print("密码错误")
            return "Password Error!"
    else:
        return f"User {user_id} is not existed!"
```

## 攻击展示

我在网页中增加了用户登录和注册的内容，首先注册了用户 test1 和 test2，密码分别为123和456，用户名和密码信息存在数据库表 `users` 中。可以看到，通过正确的用户名和密码可以登录，而在密码输入错误时也会报错


127.0.0.1:5000
Web安全实验
你可以查询并且发布评论
搜索内容
提交
所有的评论如下:
评论
提交新评论
User ID
User Password
注册
test2
456
登录
Welcome back!test2


127.0.0.1:5000
Web安全实验
你可以查询并且发布评论
搜索内容
提交
所有的评论如下:
评论
提交新评论
User ID
User Password
注册
User ID
User Password
登录
Password Error!

# Web安全实验

你可以查询并且发布评论

搜索内容 提交

## 所有的评论如下:

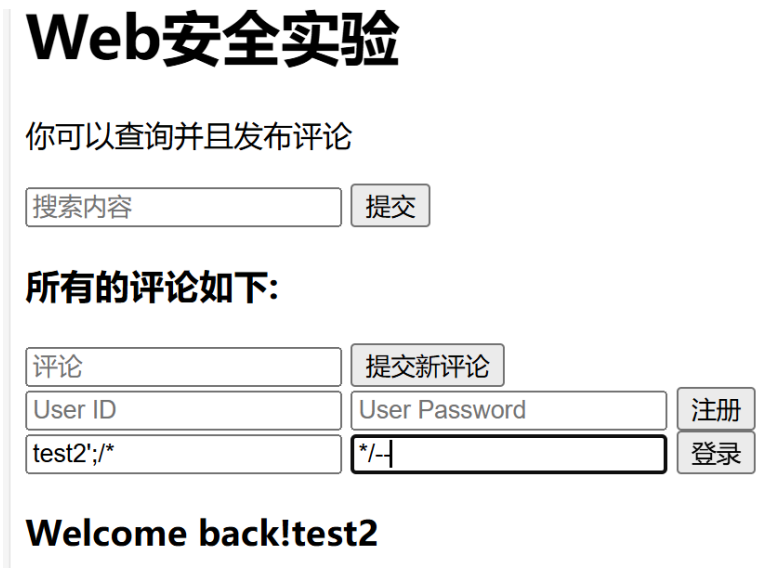
评论 提交新评论

User ID User Password 注册

User ID User Password 登录

## User test3 is not existed!

这里展示两种攻击手段，**第一种是输入指定用户名后通过注释掉数据库查询语句中的密码列，在不输入密码的情况下实现登录**




对于数据库来说，它实际上接收到的查询语句为，其中第一条为

```
sql_request: SELECT * FROM users WHERE user_id = 'test2';/*'
sql_request: SELECT * FROM users WHERE user_id = 'test2';/*' AND password =
'*/--'
```

由于注释掉了密码段，因此查询时不需要密码也可以直接返回用户内容

第二种是不指定用户名，可以从数据库中取出任意用户的用户名（这个取决于前端代码中sqlite语句的实现，我的实现是返回第一个匹配的，因此返回的是第一个用户 test1）

# Web安全实验

你可以查询并且发布评论

搜索内容 提交

## 所有的评论如下:

评论 提交新评论

User ID User Password 注册

'OR 1=1; /* */-- 登录

## Welcome back!test1

后端对应的 `sqlite` 语句为

```
sql_request: SELECT * FROM users WHERE user_id = ''OR 1=1; /*'
sql_request: SELECT * FROM users WHERE user_id = ''OR 1=1; /*' AND password = '*/--'
```

由于 `OR 1=1` 恒为真，因此可以实现在不输入用户名的情况下登录

## 防御方法

通过使用参数化查询方法，数据库系统会将用户提供的参数作为数据而不是代码来处理，从而避免了问题。

具体来说，我将登录代码从下面的字符串拼接方式改成了上面的参数化查询，从而避免了 sql 注入攻击，可以看到，再输入相同的内容，后端会把用户名当作为数据查询，而非代码执行

```
comment = db.cursor().execute('SELECT * FROM users WHERE user_id = ? AND password = ?', (user_id, password)).fetchone()
# sql_request = f"SELECT * FROM users WHERE user_id = '{user_id}' AND password = '{password}'"
# print("sql_request:", sql_request)
# comment = db.cursor().execute(sql_request).fetchone()
```

## 所有的评论如下:

评论 提交新评论

User ID User Password 注册

User ID User Password 登录

## User 'OR 1=1; /* is not existed!


# 设计一个CSRF攻击范例，并且演示如何防御

CSRF（Cross-Site Request Forgery，跨站请求伪造）攻击是一种网络攻击方式，攻击者通过伪装成受信任用户，诱使受害者在不知情的情况下执行一些未授权的操作。具体来说，用户首先在某个受信任网站登录，在浏览器中保存该会话信息（如cookie），然后用户会访问恶意网站，网站诱导用户点击恶意链接，包含精心构造的请求，这些请求以受害者的身份发送到受信任的网站上，由于之前保存过会话的信息，因此这个请求会被认定是受害者做得合法操作。网站收到请求后，会执行一些操作，于是达到了恶意网站操纵用户的目的

## 攻击成功示例

我设计了两个页面，页面A为合法页面，该页面在用户登录后会分配一个cookie，在评论时会根据cookie是否存在来决定是否允许用户评论，端口号5000，页面B为攻击页面，包含一个按钮，点击按钮会执行恶意请求，指向5000端口的页面A。

```
<h1>CSRF Attack Page</h1>
<form id="csrfForm" action="http://127.0.0.1:5000/" method="POST">
    <input type="hidden" name="comment" value="This is a CSRF attack comment!">
    <input type="submit" value="点我实现csrf攻击" />
</form>
```

当我在A页面完成登录操作后，cookie会存储在浏览器中，再点击B页面的按钮，POST请求会发送给页面A，由于cookie鉴权通过，成功将 This is a CSRF attack comment! 写入评论区中

# Web安全实验

你可以查询并且发布评论

搜索内容 提交

## 所有的评论如下:

<script>alert('存储型XSS')</script>

123

test1

This is a CSRF attack comment!

## 防御方法

我采用 CSRF令牌来防御攻击，该令牌在A页面提交请求表单时生成并加在表单中，处理POST请求时，会验证该token的合法性，进而判断操作是否为用户在合法网站A提交的。由于CSRF令牌并不是以cookie的形式存入浏览器的，而是直接包含在前端html以及发送的请求中，因此无法被网站B预知和伪造，从而实现了防御的目的

```
def generate_csrf_token():
    if '_csrf_token' not in session:
        session['_csrf_token'] = hmac.new(app.secret_key, os.urandom(64),
hashlib.sha256).hexdigest()
    return session['_csrf_token']
def validate_csrf_token(token):
    return token == session.get('_csrf_token')
```

```
<form action="/" method="POST">
  <input type="hidden" name="csrf_token" value="{{ csrf_token }}">
  <input type="text" name="comment"
         placeholder="评论" autocomplete="off" />
  <input type="submit" value="提交新评论" />
</form>
```

可以看到，重新执行csrf攻击，会由于校验不通过进而报错。