常见 Web 漏洞演示 ★★★

实验报告

计04 何秉翔 2020010944

1. 环境设置

我们所使用的浏览器为 chrome，版本为 113.0.5672.92 (Official Build) (64-bit) (cohort: Stable)，后端运行在 windows 10 powershell 5.1.19041.2673 上，Python 版本为 Python 3.9.1，flask 版本为 2.0.1。

2. 实验目的

演示三种常见的 Web 攻击：XSS、CSRF、SQL 注入，并体会现代浏览器如何防御这些漏洞。实验用到的源码以及相关的输入文件可在 清华云盘 获得。

3. 关键步骤 && 关键源代码 && 实验结果

3.1 XSS 攻击及防御

XSS 的攻击包括两种，一种是反射型 XSS，也被称为非持久型 XSS，只会攻击一次，不会保存在数据库中；另一种是存储型 XSS，也被称为持久型 XSS，可执行代码将永久保存在服务器中，每次网站被打开时都会执行相应逻辑。实验提供的框架已经能够成功进行这两种 XSS 攻击，然后我们将通过**转义字符**的方法进行 XSS 防御，下面演示如下：

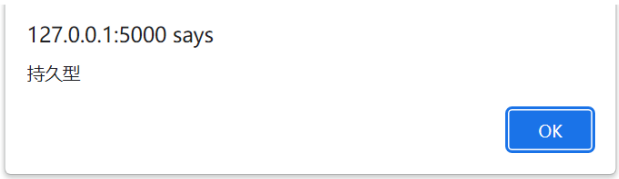
1. 非持久型 XSS：

我们只需在"搜索内容"输入框输入 <script>alert('非持久型')</script>，然后点击"提交"，即可实现非持久型 XSS 攻击。具体而言，输入的 js 代码被前端当成搜索内容渲染到了前端并执行。

由于未保存到数据库，因此仅执行这一次，重新刷新网页后注入的代码已消失：

2. 持久型 XSS：

我们只需在"评论"输入框输入 <script>alert('持久型')</script>，然后点击"提交"，即可实现持久型 XSS 攻击。具体而言，此时输入的 js 代码通过后端 add_comment 保存到数据库里，在每一次打开网页时，后端将数据库里保存的评论都交给前端渲染，因此每一次打开网页都将执行一遍注入的代码：



3. 通过**转义字符**进行 XSS 防御：

对于用户输入的字符串，我们需要判断其中有无执行代码的成分，比如对于 <script> 标签等 js 代码，因此我们考虑对 <、''、\ 等字符进行转义，因此我们考虑对用户可能的两部分输入，"搜索内容"以及"评论"进行转义，再对转义后的结果进行处理。为此，我们使用 html 库的 escape() 方法实现如下：

```python
import html

defend = True # 开启防御

if request.method == 'POST':
    # 对 comment 进行转义
    comment = html.escape(request.form['comment']) if defend else request.form['comment']
    add_comment(comment)

search_query = request.args.get('q')
# 在 search_query 非 None 时进行转义
search_query = html.escape(search_query) if search_query != None and defend else
search_query
comments = get_comments(search_query)
```

此时再尝试**持久型** XSS 攻击，结果如下：

再尝试**非持久型** XSS 攻击，结果如下：

3.2 SQL 攻击及防御

首先，我们新增登录功能如下：

- 数据库新增 users 数据表：

```python
db.cursor().execute('CREATE TABLE IF NOT EXISTS users '
                    '(id INTEGER PRIMARY KEY, '
                    'username TEXT, '
                    'password TEXT)')
```

- 后端新增注册处理接口：

```python
# 注册
@app.route('/register', methods=['POST'])
def register():
    username = request.form['username']
    password = request.form['password']

    db = connect_db()
    db.cursor().execute('INSERT INTO users (username, password) '
                        'VALUES (?, ?)', (username, password))
    db.commit()
    return redirect('/')
```

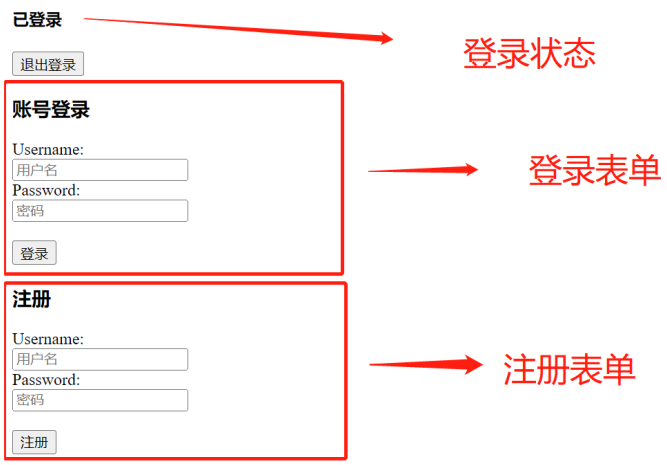
- 后端新增登录处理接口 `/login`，我们通过请求给出的 `username` 和 `password` 来构造数据库查询语句，并执行。

```python
def login():
    ...
    if defend: # 如果开启防御
        query = 'SELECT * FROM users WHERE username = ? AND password = ?'
        params = (username, password)
        cursor.execute(query, params)
    else: # 未开启防御
        query = 'SELECT * FROM users WHERE username = \'' + str(username) + '\' AND password = \'' + str(password) + '\''
        cursor.execute(query)

    user_info = cursor.fetchall()
    login_status = True if len(user_info) > 0 else False
    ...
```

具体而言，从 `users` 数据表里选出所有的用户名和密码都与请求所匹配的记录，如果存在这个记录，则登录成功。

- 前端新增登录和注册表单：



正常操作过程如下：

1. 在注册表单输入用户名和密码，点击"注册"提交
2. 在账号登录表单输入已注册的用户名和密码，点击"登录"提交，登录成功则登录状态变为"已登录"，否则为"未登录"，并显示尝试登录的用户名。
3. 若想退出登录状态，则点击"退出登录"按钮即可，登录状态更新为"未登录"

SQL 注入攻击：

首先我们注册一个账户如下，用户名为 `admin`，密码为 `abc`：

```
sqlite> select * from users;
1|admin|abc
sqlite>
```

然后我们使用用户名 `admin' --` 来登录，目标是输入任何密码我们都能成功登录 `admin` 的账号，此时 `SQL` 注入导致整个数据库的 `username = admin` 的记录全部被筛选出来，忽视 `password` 字段，因此被无任何校验的后端判为登录成功。

`SQL` 注入防御：

我们采取以下方法进行防御：所有的查询语句使用数据库提供的参数化查询接口，参数化的语句使用参数，而不是将用户输入变量嵌入到 `SQL` 语句中，即不要直接拼接 `SQL` 语句，为此，我们重写 `login` 的后端处理函数，其中更改的关键部分为：

```
query = 'SELECT * FROM users WHERE username = ? AND password = ?'
params = (username, password)
cursor.execute(query, params)
```

在定义查询语句时，我们使用占位符 `?` 表示参数，而不是将用户输入变量嵌入到 `SQL` 语句中。然后，我们定义查询参数 `params`，并将其传递给 `execute()` 方法，以执行查询。最后，我们使用 `fetchall()` 方法获取查询结果，并对结果进行处理。这种方式可以有效地防范 `SQL` 注入攻击。

此时我们再用 `admin' --` 来登录，发现登录失败：

3.3 `CSRF` 攻击及防御

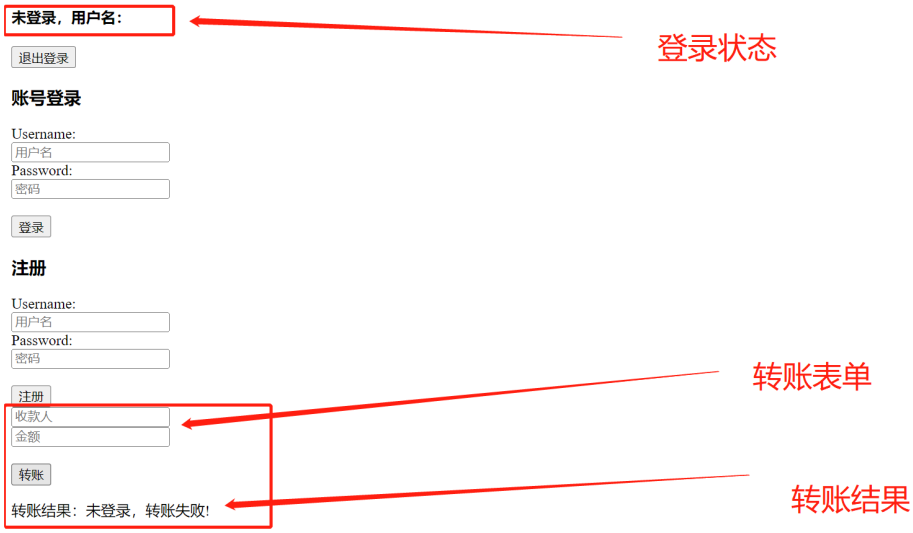
首先我们先添加关于转账的功能，在后端里我们添加 `/csrf` 的接口，用于进行转账，关键代码部分如下：

```
# csrf attack: 转账
@app.route('/csrf', methods=['POST'])
def csrf():
    account = request.form['account']
    amount = request.form['amount']
    username = request.cookies.get('username')

    result_str = ''
    if username is None or username == '':
        result_str = '未登录，转账失败！'
    else:
        result_str = '已登录，成功向 %s 转账 %s 元!' % (account, amount)
    ...
```

具体而言，我们通过 `cookie` 来获得用户的登录状态，若未登录，则转账失败，否则成功转账。

正常情况下，未登录时，转账效果如下：



已登录时，转账效果如下：

我们接下来在 `/login` 的接口内将 `cookie` 信息存好，为之后转账判断是否登录做准备：

```python
if login_status:
    if defend:
        response.set_cookie('username', username, samesite='Strict', secure=True)
    else:
        response.set_cookie('username', username)
else:
    if defend:
        response.set_cookie('username', '', expires=0, samesite='Strict', secure=True)
    else:
        response.set_cookie('username', '', expires=0)
```

根据是否开启防御模式，我们设置 `cookie` 的 `samesite` 属性，以防止 `CSRF` 攻击。接下来我们构造一个恶意网站 `eval.html`，其主要的 `<body>` 下代码如下，目标是向 `attacker` 转账 `10` 元：

```html
<body onload="document.getElementById('csrf').submit();">
    <form id="csrf" action="http://127.0.0.1:5000/csrf" method="POST">
        <input type="hidden" name="account" value="attacker"/>
        <input type="hidden" name="amount" value="10"/>
    </form>
</body>
```

该恶意网页由 `app_eval.py` 及 `eval.html` 来部署在 `127.0.0.1:5001`，来模拟跨站访问，虽然仍是同站，但在后面我们将模拟跨站访问。

当用户不小心跳转到该网页时，会自动向 `http://127.0.0.1:5000/csrf` 提交一个转账的表单。为此我们在原来的页面 `index.html` 内新增一个诱引用户点击的按钮：

```
<form action="http://127.0.0.1:5001" onclick="removeCookie()" method="POST">
  <input type="submit" value="点我，有㊙" />
</form>
```

该按钮点击后将请求发至 `http://127.0.0.1:5001`，然后自动向 `http://127.0.0.1:5000/csrf` 提交一个转账的表单，如果在用户未登录时点击"**点我，有㊙**"按钮，则转账失败：

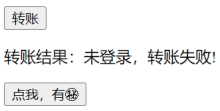
但是如果用户已经登录，则请求将携带着用户的 `cookie`，骗过 `/csrf` 转账的后端处理，导致意外转账的发生：

下面我们来展示如何进行防御，一般做法是给 `cookie` 加上 `samesite='Strict' or samesite='Lax'` 属性，但是我们部署的两个服务，一个是正常的 `127.0.0.1:5000`，另一个则是攻击者的"跨站"网站 `127.0.0.1:5001`，我们模拟跨站请求如下：

```
<form action="http://127.0.0.1:5001" onclick="removeCookie()" method="POST">
  <input type="submit" value="点我，有㊙" />
</form>

<script>
  function removeCookie() {
    var defend = '{{ defend }}';
    if(defend === "True") {
      document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";
    }
  }
</script>
```

当用户点击诱引按钮后，我们根据是否开启防御模式来进行 cookie 的设置，如果开启了防御模式，则将 cookie 设置为过期状态，相当于模拟跨站请求中的 cookie 被 samesite 设置下的浏览器所拦截；如果没开启防御模式，则正常携带 cookie 发送。如果开启了防御模式，则未登录状态下仍然无法转账，即便用户已经登录，在开启了防御机制下，也无法进行转账：



4．影响因素分析

1. 对输入进行过滤或者转义：如果应用缺乏充分的输入过滤，攻击者可能能够注入恶意代码或查询，并通过该应用执行任意操作。因此如果对于引号、尖括号、斜杠进行转义，将能有效防范 XSS 和 SQL 注入攻击。
2. 浏览器是否对 cookie 设置了 samesite 属性，若设置了 samesite，则能有效防范跨站请求伪造即 CSRF 攻击；另外浏览器通过阻止第三方网站请求接口，也可以防止 CSRF 攻击。