OAuth CSRF攻击实验

# 【实验目的】

通过一个简单的OAuth CSRF攻击实验，来熟悉OAuth的协议流程，并通过攻击实验来理解OAuth协议的安全性。

# 【实验环境】

IDE：IntelliJ IDEA 2018.3.5 x64

JDK：1.8

OS：Windows 10 x64

数据库：mysql-5.6.45-winx64

服务器：apache-tomcat-8.5.45

IdP服务器地址：<http://localhost:8080>

RP服务器地址：<http://localhost:8081>

Attack服务器地址：[http://localhost:8082](http://localhost:8081)

# 【实验内容】

通过OAuth CSRF攻击实验，达到如下效果：

攻击者通过诱导用户点击恶意链接，使得用户的账号通过OAuth协议与攻击的IdP账号绑定在一起，从而可以使得攻击者用自己的IdP账号通过OAuth登录到用户的账号。

* 具体实验步骤：

Step 1. 攻击者登录到RP，并且选择绑定自己的IdP账号

Step 2. RP将攻击者重定向到IdP，攻击者登录IdP，IdP向他显示是否授权RP访问的页面

Step 3. 攻击者在点击”同意授权“之后，截获IdP服务器返回的含有Authorization code参数的HTTP响应

Step 4.攻击者精心构造一个Web页面，它会触发向IdP发起令牌申请的请求，而这个请求中的Authorization Code参数正是上一步截获到的code

Step 5.攻击者将这个Web页面放到互联网上，等待或者诱骗受害者来访问

Step 6.受害者之前登录了RP网站，只是没有把自己的账号和其他社交账号绑定起来。在受害者访问了攻击者准备的这个Web页面，令牌申请流程在张三的浏览器里被顺利触发，RP从IdP那里获取到access\_token，但是这个token以及通过它进一步获取到的用户信息却都是攻击者的。

Step 7. RP将攻击者的IdP账号同受害者的RP账号关联绑定起来

Step 8. 从此以后，攻击者就可以用自己的IdP账号通过OAuth登录到受害者在RP中的账号，冒充受害者的身份执行各种操作。

* 具体的攻击流程图如下图所示：

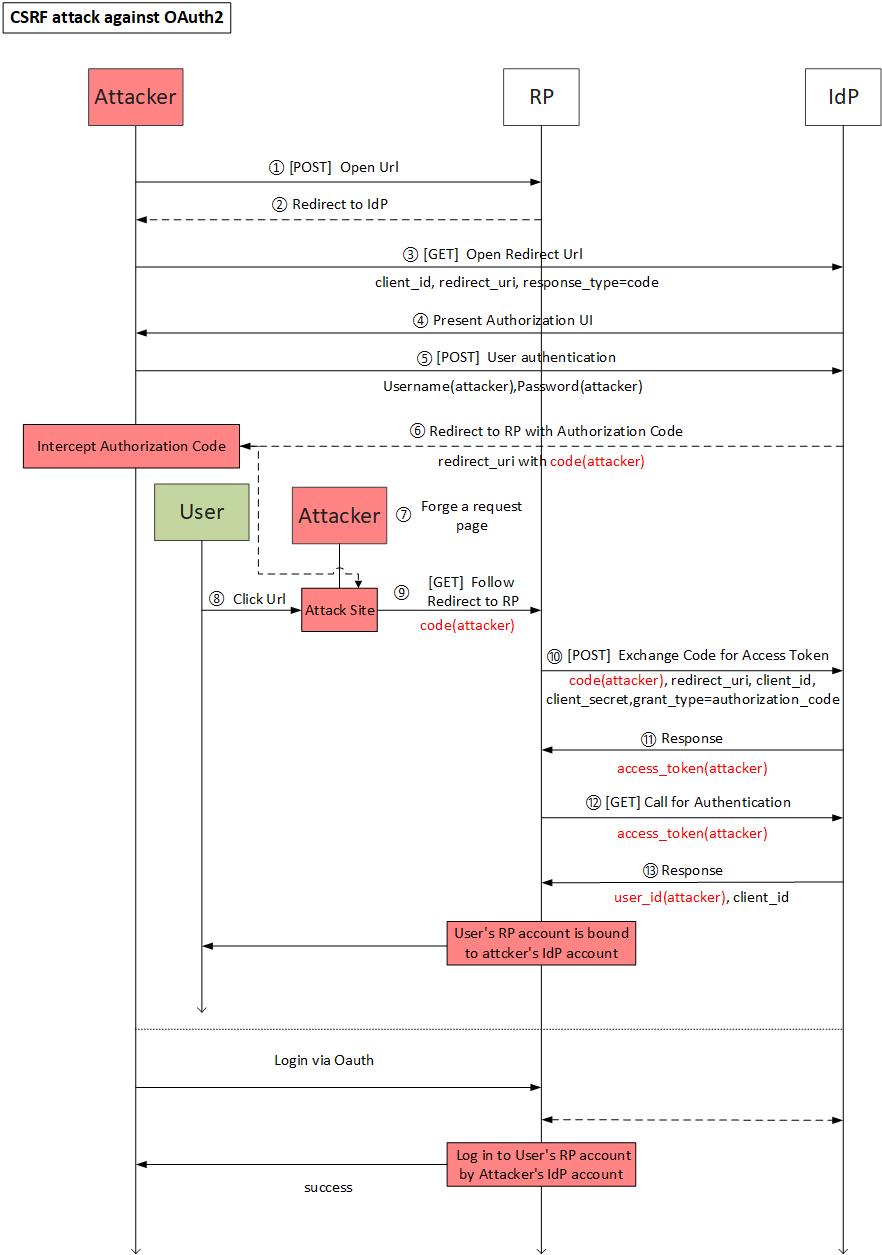
localhost:8082/attack

http://localhost:8081/binding\_request?code=[**code**]&username=[username]& scope=binding

localhost:8080

localhost:8081

localhost:8082



注：第9步中，攻击者应该获取到当前登录状态下用户的cookie，并且将cookie写入请求头中，RP会检查cookie来判断用户是否是登录状态，从而判断是否同意进行绑定请求，但目前从实现上来讲，获取cookie，并检查cookie较为困难，因此在实际实现中，cookie只是已简单的参数形式，传给RP，RP也不检查cookie。

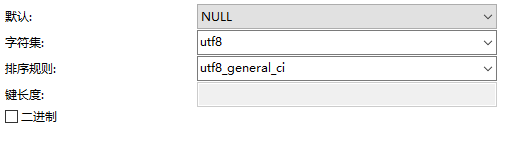
# 【数据库创建】

* RP数据库：建立名为RP的数据库，并且建立user表，其字段如下：

user@rp

|  |  |  |  |
| --- | --- | --- | --- |
| id | username | password | idp |
| 编号 | RP用户名 | RP密码 | 身份认证提供商 |



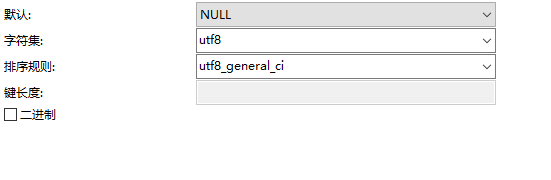


* IdP数据库：建立名为shiro的数据库，建立oauth2\_client表，其字段如下：

oauth2\_client@shiro

|  |  |  |  |
| --- | --- | --- | --- |
| id | client\_name | client\_id | client\_secret |
| 编号 | 第三方应用名称 | 第三方应用id | 第三方应用密钥 |



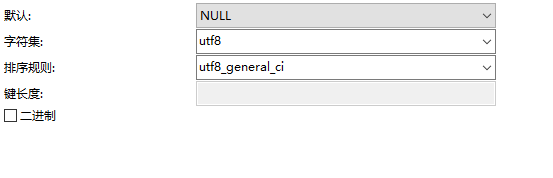


* 建立oauth2\_user表，其字段如下：

oauth2\_user@shiro

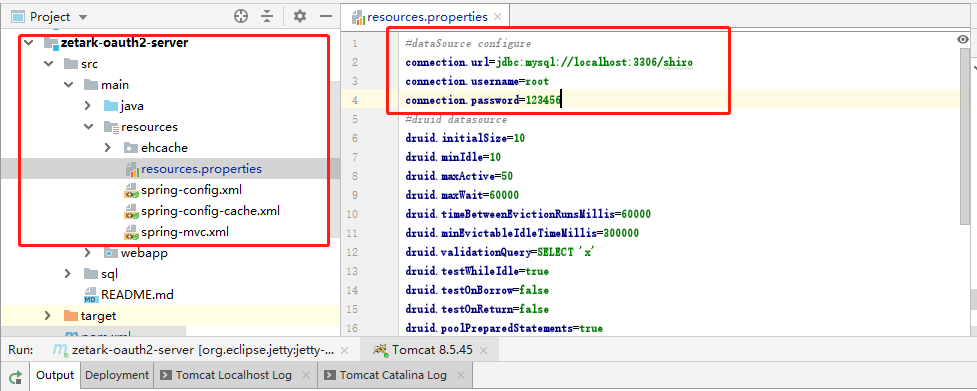
|  |  |  |  |
| --- | --- | --- | --- |
| id | username | password | salt |
| 编号 | IdP用户名 | IdP密码 | 盐 |





创建完数据库，检查项目中的数据库配置是否正确

检查IdP中的数据库配置

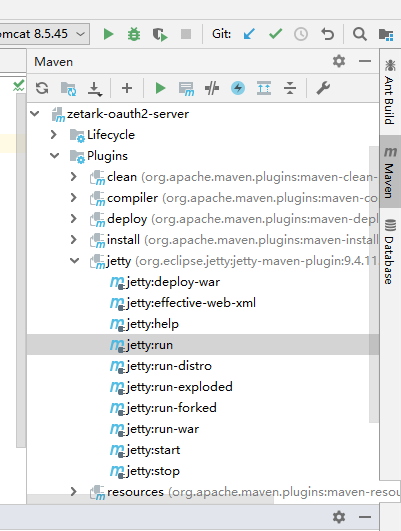


检查RP中的数据库配置

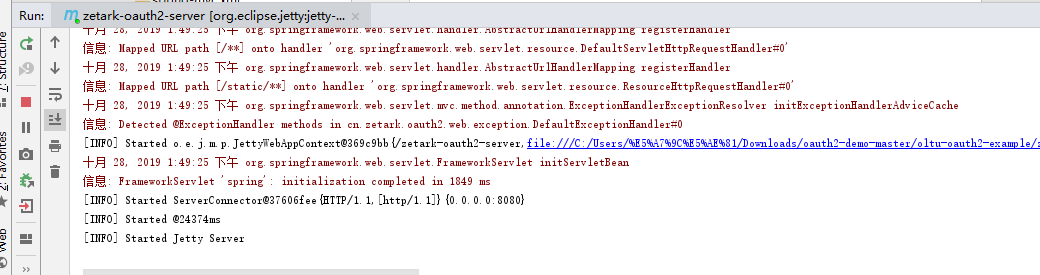


# 【服务器启动】

1. 启动IdP服务器，启动方式如下：

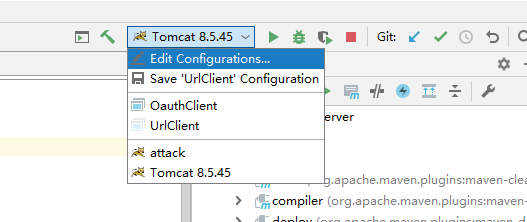


双击jetty:run，启动服务器，成功如下图所示：

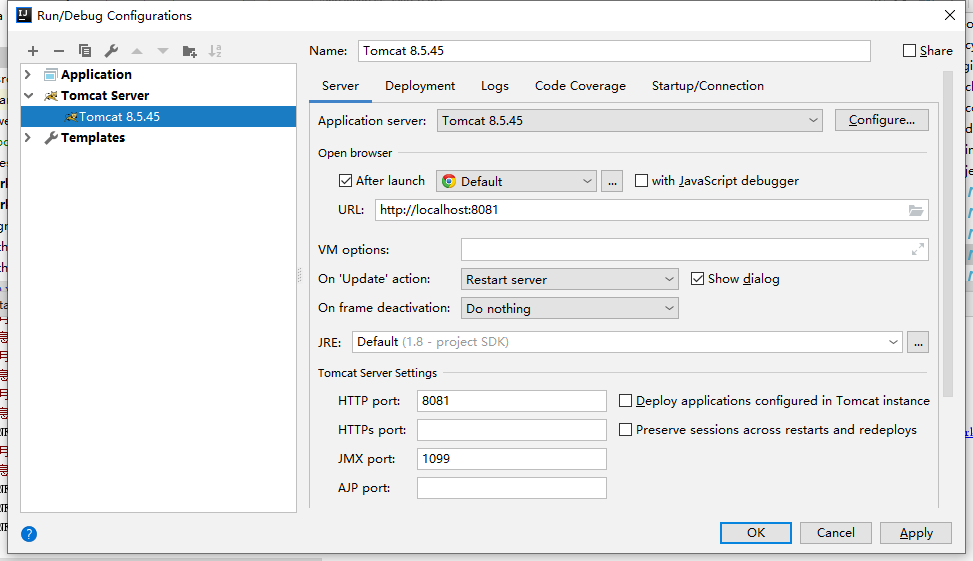


1. 启动RP服务器

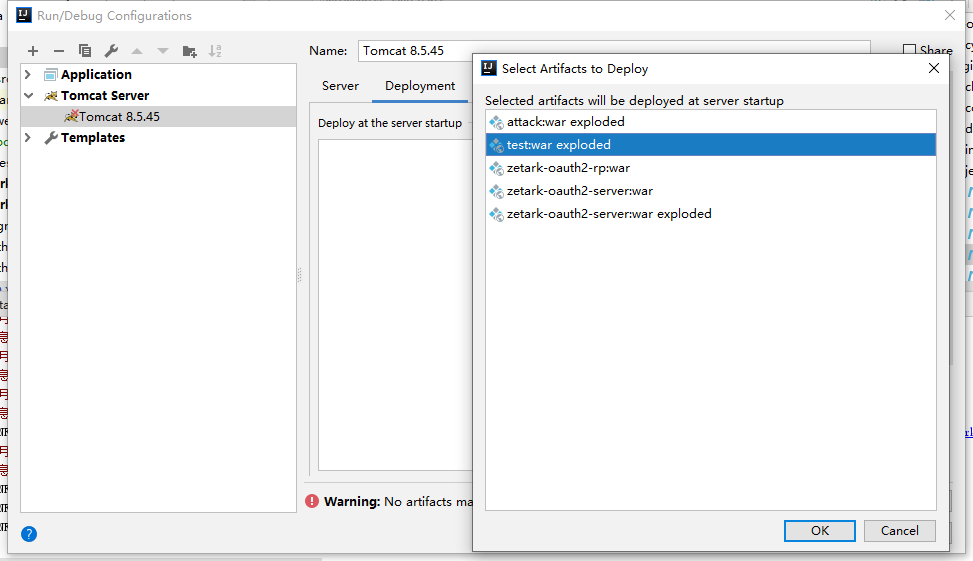
首先将工程部署到tomcat服务器上，点击 Edit Configurations

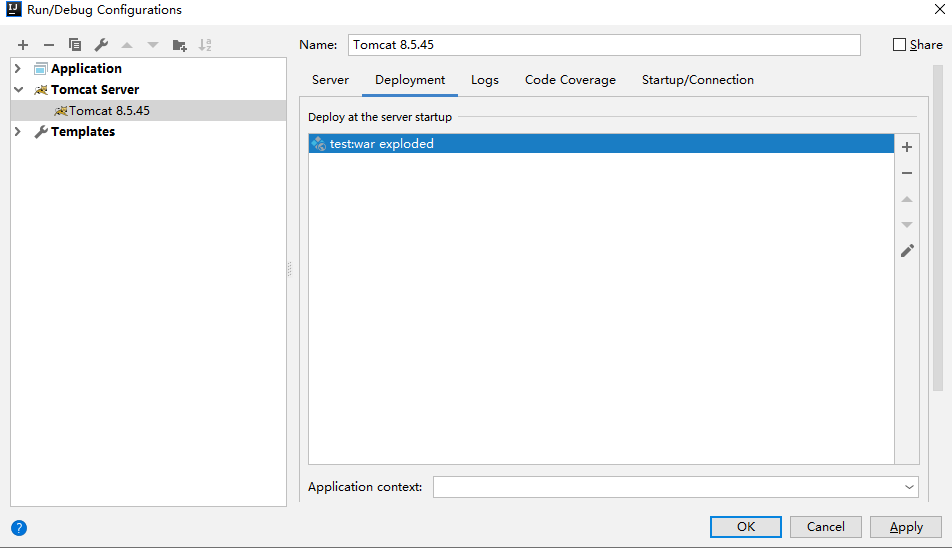


进行如下配置



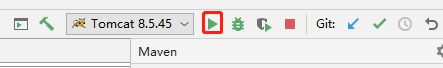
将项目部署到tomcat。



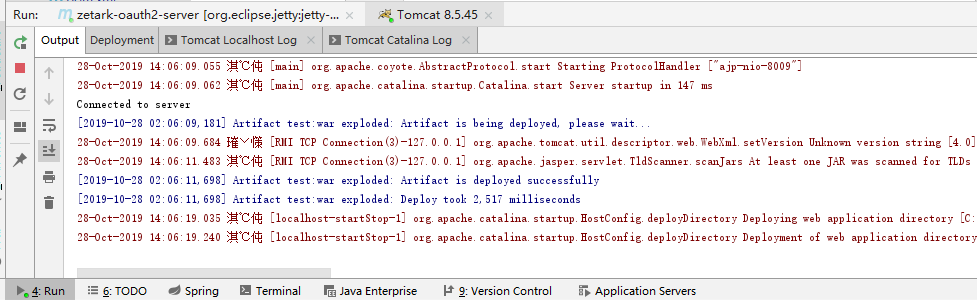


注意application context为空。

随后启动服务器



成功如下图所示



# 【实验流程】

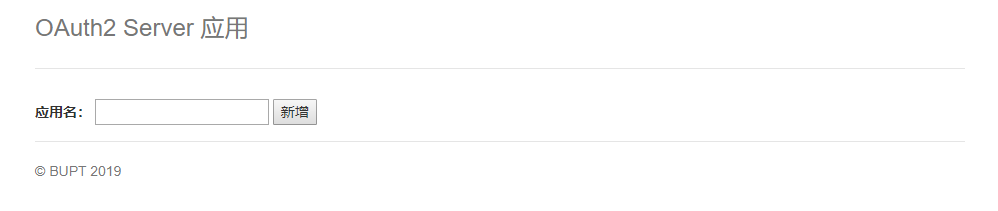
## 注册RP，获得ClientID和Secret

1. 打开IdP网站<http://localhost:8080/zetark-oauth2-server/client>

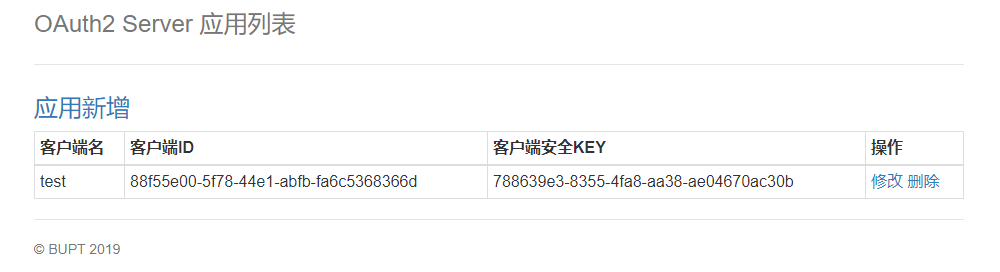
进行第三方应用（RP）注册



1. 点击应用新增，输入第三方应用（RP）的名称



1. 注册完毕获得客户端ID（client\_id）和客户端安全KEY（clinet\_secret），RP需要记录这两个参数，用于后续的OAuth认证。



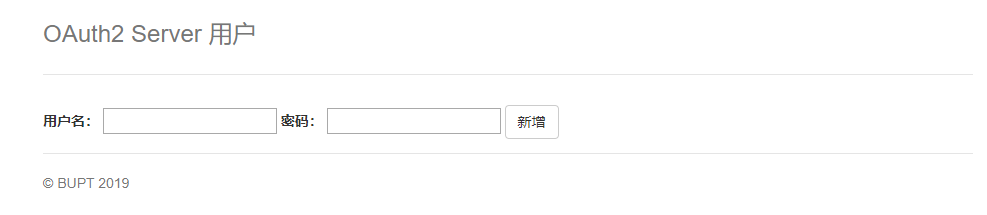
## 攻击者：注册IdP账号

1. 打开IdP网站<http://localhost:8080/zetark-oauth2-server/user>

进行账号注册，该账号将用于绑定RP账号



1. 点击用户新增，注册用户



1. 注册成功后，显示用户



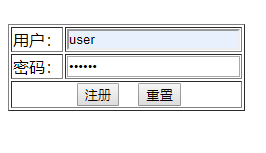
## 用户注册RP账号

1. 打开RP网站<http://localhost:8081/>

进行账号注册



1. 点击注册，注册用户Alice



1. 注册成功后，返回登录页面，进行登录



## 攻击者完成攻击前的状态

1. 我们首先尝试，在IdP(attacker)账号没有绑定RP(user)账号的情况下，用IdP(attacker)账号来登录RP网站



点击使用IdP授权，我们发现IdP显示，应用test（RP）正在请求接入



输入IdP账号，用户名attacker，点击登录并授权，验证通过后会显示以下界面



返回显示该用户未绑定。确实，我们的attcker用户只是IdP用户，并没有绑定过RP用户，因此无法用IdP的账号来登录RP网站。

## 攻击者的操作流程

1. 接下来进行CSRF攻击，首先我们让用户user登录到rp网站



1. 攻击者在用户（user）保持登录状态时，打开绑定用户的链接：

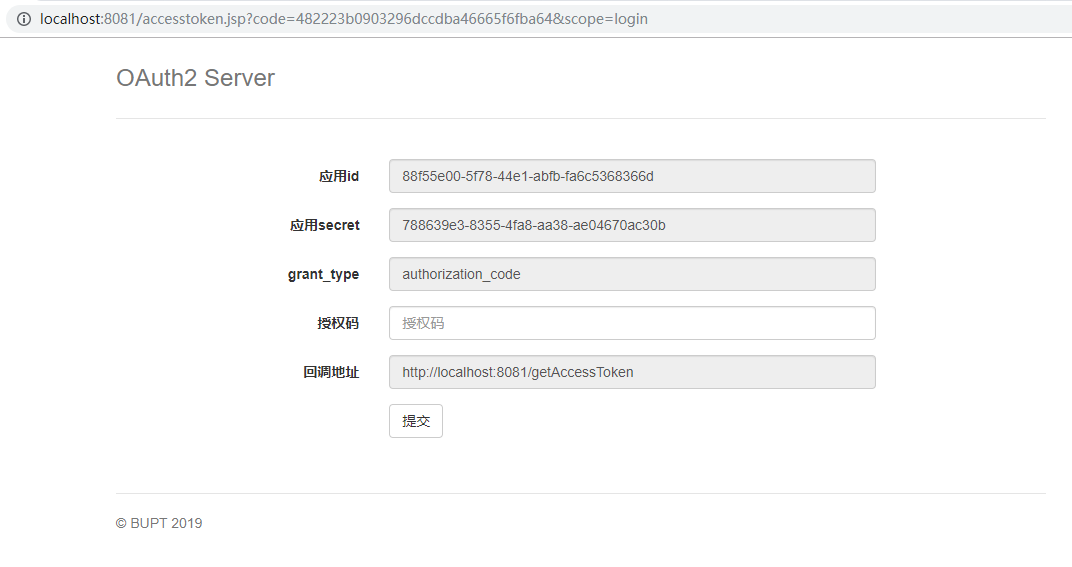
<http://idp.com:8080/zetark-oauth2-server/authorize?client_id=88f55e00-5f78-44e1-abfb-fa6c5368366d&response_type=code&redirect_uri=http://rp.com:8081/accesstoken.jsp&scope=bindlogin>

[http://localhost:8080/zetark-oauth2-server/authorize?client\_id=88f55e00-5f78-44e1-abfb-fa6c5368366d&response\_type=code&redirect\_uri=http://localhost:8081/accesstoken.jsp](http://localhost:8080/zetark-oauth2-server/authorize?client_id=88f55e00-5f78-44e1-abfb-fa6c5368366d&response_type=code%20&redirect_uri=http://localhost:8081/accesstoken.jsp)

输入攻击者在IdP上的账号（attacker）



1. 点击登录并授权，验证通过后，获得返回的授权码（code）



http://localhost:8081/accesstoken.jsp?code=c50831f1e31935146650ec8117ef3677&scope=bindlogin

此处code= 482223b0903296dccdba46665f6fba64

1. 将code放入事先写好的攻击页面中，并重新运行



## 用户登录RP并访问攻击者的网站

1. 等待用户访问攻击者的网站，点击恶意链接（<http://localhost:8082/attack>）

攻击者伪造一个post请求发给RP网站：<http://localhost:8081/binding_request>，其中post请求含有：授权码（code），受害者RP的用户名（rp\_user\_name）

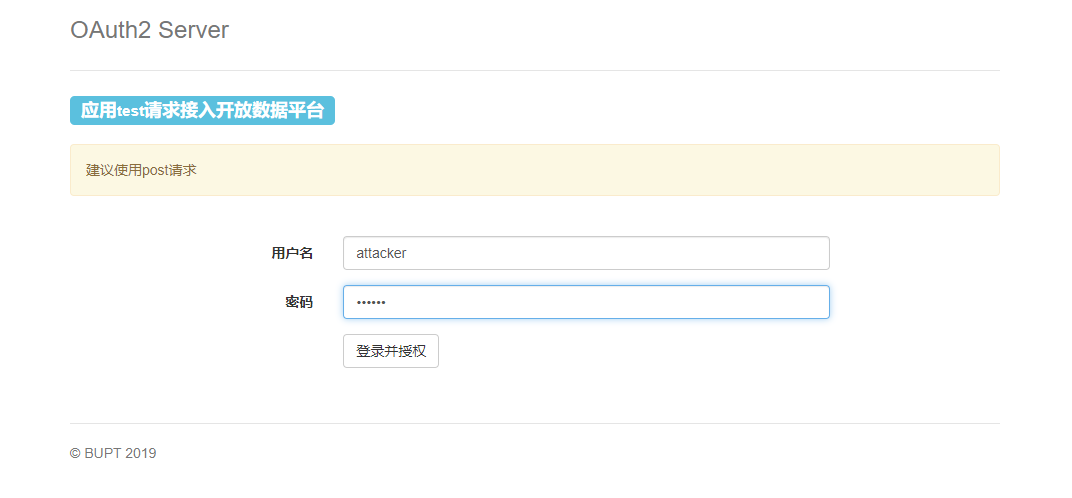
1. 当用户点击恶意链接中，当前RP用户就会与攻击者的IdP账号绑定，

完成CSRF攻击。

## 攻击者入侵用户的RP账户

1. 攻击者就可以用攻击者的IdP账号登入到用户的RP账号中去。





发现登录成功。



# 【实验结论】

由于OAuth2的认证流程是分为好几步来完成的，当第三方应用在收到一个GET请求时，除了能知道当前用户的cookie，以及URL中的Authorization Code之外，难以分辨出这个请求到底是用户本人的意愿，还是攻击者利用用户的身份伪造出来的请求。于是乎，攻击者就能使用移花接木的手段，提前准备一个含有自己的Authorization Code请求，并让受害者的浏览器来接着完成后续的令牌申请流程

* **攻击者攻击前提条件**

1. 在攻击过程中，受害者在RP上的用户会话(User Session)必须是有效的，也就是说，受害者在受到攻击前已经登录了RP。
2. 整个攻击必须在短时间内完成，因为OAuth2提供者颁发的Authorization Code有效期很短，OAuth2官方推荐的时间是不大于10分钟，而一旦Authorization Code过期那么后续的攻击也就不能进行下去了。
3. 一个Authorization Code只能被使用一次，如果OAuth2提供者收到重复的Authorization Code，它会拒绝当前的令牌申请请求。不止如此，根据OAuth2官方推荐，它还可以把和这个已经使用过的Authorization Code相关联的access\_token全部撤销掉，进一步降低安全风险。

* **预防方法**

要防止这样的攻击其实很容易，作为第三方应用的开发者，只需在OAuth2认证过程中加入state参数，并验证它的参数值即可。具体细节如下：

在将用户重定向到OAuth2的Authorization Endpoint去的时候，为用户生成一个随机的字符串，并作为state参数加入到URL中。在收到OAuth2服务提供者返回的Authorization Code请求的时候，验证接收到的state参数值。如果是正确合法的请求，那么此时接受到的参数值应该和上一步提到的为该用户生成的state参数值完全一致，否则就是异常请求。state参数值需要具备下面几个特性：

1. 不可预测性：足够的随机，使得攻击者难以猜到正确的参数值
2. 关联性：state参数值和当前用户会话（user session）是相互关联的
3. 唯一性：每个用户，甚至每次请求生成的state参数值都是唯一的
4. 时效性：state参数一旦被使用则立即失效