http_authentication Documentation

发布 *v1.0*

yuanjh

2021 年 01 月 08 日

Contents:

web 研发安全方面，http 认证鉴权知识点集锦

CHAPTER 1

http 认证鉴权 01 基本认证和摘要认证

基本认证与摘要认证用于在 HTTP 报文交互中，服务端确认客户端身份。

1.1 基本认证

Base64(user:pwd) 后，放在 Http 头的 Authorization 中发送给服务端来作认证.

用 Base64 纯只是防君子不防小人的做法。所以只适合用在一些不那么要求安全性的场合。

1.2 摘要认证

digest authentication（HTTP1.1 提出的基本认证的替代方法）

这个认证可以看做是基本认证的增强版本，不包含密码的明文传递。

引入了一系列安全增强的选项；“保护质量”(qop)、随机数计数器由客户端增加、以及客户生成的随机数。

$$HA1 = MD5\Big(A1\Big) = MD5\Big(username : realm : password\Big)$$

如果 qop 值为“auth”或未指定，那么 HA2 为

$$HA2 = MD5\Big(A2\Big) = MD5\Big(method : digestURI\Big)$$

如果 qop 值为“auth-int”，那么 HA2 为

$$HA2 = MD5\Big(A2\Big) = MD5\Big(method : digestURI : MD5(entityBody)\Big)$$

如果 qop 值为“auth”或“auth-int”，那么如下计算 response:

$$response = MD5\Big(HA1 : nonce : nonceCount : clientNonce : qop : HA2\Big)$$

如果 qop 未指定，那么如下计算 response:

$$response = MD5\Big(HA1 : nonce : HA2\Big)$$

在 HTTP 摘要认证中使用 MD5 加密是为了达成”不可逆的”，也就是说，当输出已知的时候，确定原始的输入应该是相当困难的。如果密码本身太过简单，也许可以过尝试所有可能的输入来找到对应的输出（穷举攻击），甚至可以通过字典或者适当的查找表加快查找速度。

认证过程

客户端请求一个需要认证的页面，但是不提供用户名和密码。通常这是由于用户简单的输入了一个地址或者在页面中点击了某个超链接。
服务器返回 401 "Unauthorized" 响应代码，并提供认证域 (realm)，以及一个随机生成的、只使用一次的数值，称为密码随机数 nonce。
此时，浏览器会向用户提示认证域 (realm)（通常是所访问的计算机或系统的描述），并且提示用户名和密码。用户此时可以选择取消。
一旦提供了用户名和密码，客户端会重新发送同样的请求，但是添加了一个认证头包括了响应代码。
注意：客户端可能已经拥有了用户名和密码，因此不需要提示用户，比如以前存储在浏览器里的。

客户端请求 (无认证):

```
GET /dir/index.html HTTP/1.0
Host: localhost
(跟随一个新行，形式为一个回车再跟一个换行)
```

服务器响应:

```
HTTP/1.0 401 Unauthorized
Server: HTTPd/0.9
Date: Sun, 10 Apr 2005 20:26:47 GMT
WWW-Authenticate: Digest realm="testrealm@host.com",    //认证域
qop="auth,auth-int",    //保护质量
```

(continues on next page)

(续上页)

```
nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",  //服务器密码随机数
opaque="5ccc069c403ebaf9f0171e9517f40e41"
Content-Type: text/html
Content-Length: 311


<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
 "http://www.w3.org/TR/1999/REC-html401-19991224/loose.dtd">
<HTML>
  <HEAD>
<TITLE>Error</TITLE>
<META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=ISO-8859-1">
  </HEAD>
  <BODY><H1>401 Unauthorized.</H1></BODY>
</HTML>
```

客户端请求 (用户名“Mufasa”, 密码“Circle Of Life”):

```
GET /dir/index.html HTTP/1.0
Host: localhost
Authorization: Digest username="Mufasa",# 用户输入或浏览器保存
 realm="testrealm@host.com",# 上一步: 服务器响应里的
 nonce="dcd98b7102dd2f0e8b11d0f600bfb0c093",# 上一步: 服务器响应里的
 uri="/dir/index.html",
 qop=auth,
 nc=00000001,//请求计数
 cnonce="0a4f113b",   //客户端密码随机数
 response="6629fae49393a05397450978507c4ef1",
 opaque="5ccc069c403ebaf9f0171e9517f40e41"
(跟随一个新行, 形式如前所述)。
```

这里面的均是报文明文，可见里面没有 password

服务器响应:

```
HTTP/1.0 200 OK
Server: HTTPd/0.9
Date: Sun, 10 Apr 2005 20:27:03 GMT
Content-Type: text/html
Content-Length: 7984
(随后是一个空行, 然后是所请求受限制的 HTML 页面)
```

response 值由三步计算而成。当多个数值合并的时候，使用冒号作为分割符:

```
1、对用户名、认证域 (realm) 以及密码的合并值计算 MD5 哈希值，结果称为 HA1(安全相关)。
若算法是：MD5
则 A1=<user>:<realm>:<password>
若算法是：MD5-sess
则 A1=MD5(<user>:<realm>:<password>):<nonce>:<cnonce>

2、对 HTTP 方法以及 URI 的摘要的合并值计算 MD5 哈希值，例如，"GET" 和 "/dir/index.html"，结果称为 HA2(报文相关)。
A2 表示是与报文自身相关的信息，比如 URL，请求反复和报文实体的主体部分，A2 加入摘要计算主要目的是有助于防止反复，资源或者报文被篡改。
若 qop 未定义或者 auth：
A2=<request-method>:<uri-directive-value>
若 qop 为 auth:-int
A2=<request-method>:<uri-directive-value>:MD5(<request-entity-body>)

3、对 HA1、服务器密码随机数 (nonce)、请求计数 (nc)、客户端密码随机数 (cnonce)、保护质量 (qop) 以及 HA2 的合并值计算 MD5 哈希值。结果即为客户端提供的 response 值。
摘要的计算规则：
若 qop 没有定义：摘要 response=MD5(MD5(A1):<nonce>:MD5(A2))
若 qop 为 auth：摘要 response=MD5(MD5(A1):<nonce>:<nc>:<cnonce>:<qop>:MD5(A2))
若 qop 为 auth-int：摘要 response= MD5(MD5(A1):<nonce>:<nc>:<cnonce>:<qop>:MD5(A2))
```

因为服务器拥有与客户端同样的信息，因此服务器可以进行同样的计算，以验证客户端提交的 response 值的正确性。在上面给出的例子中，结果是如下计算的。(MD5() 表示用于计算 MD5 哈希值的函数；“\” 表示接下一行；引号并不参与计算)

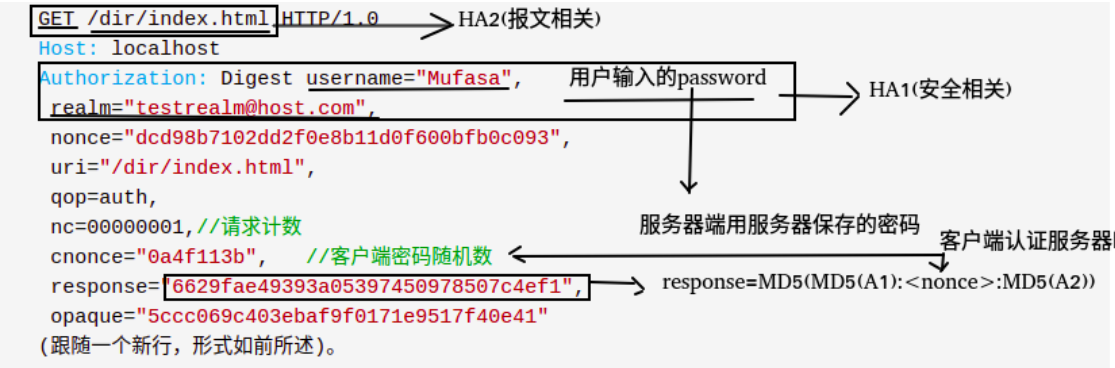
```
HA1 = MD5( "Mufasa:testrealm@host.com:Circle Of Life" )
       = 939e7578ed9e3c518a452acee763bce9

HA2 = MD5( "GET:/dir/index.html" )
       = 39aff3a2bab6126f332b942af96d3366

Response = MD5( "939e7578ed9e3c518a452acee763bce9:\
                         dcd98b7102dd2f0e8b11d0f600bfb0c093:\
00000001:0a4f113b:auth:\
39aff3a2bab6126f332b942af96d3366" )
= 6629fae49393a05397450978507c4ef1
```

此时客户端可以提交一个新的请求，重复使用服务器密码随机数 (nonce)（服务器仅在每次“401”响应后发行新的 nonce)，但是提供新的客户端密码随机数 (cnonce)。在后续的请求中，十六进制请求计数器 (nc) 必须比前一次使用的时候要大，否则攻击者可以简单的使用同样的认证信息重放老的请求。由服务器来确保在每个发出的密码随机数 nonce 时，计数器是在增加的，并拒绝掉任何错误的请求。显然，改变 HTTP 方法和/或计数器数值都会导致不同的 response 值。

服务器应当记住最近所生成的服务器密码随机数 nonce 的值。也可以在发行每一个密码随机数 nonce 后，记

住过一段时间让它们过期。如果客户端使用了一个过期的值，服务器应该响应“401”状态号，并且在认证头中添加 stale=TRUE，表明客户端应当使用新提供的服务器密码随机数 nonce 重发请求，而不必提示用户其它用户名和口令。

综合图



1.3 其他认证方式

Cookie

Cookie 认证机制就是为一次请求认证在服务端创建一个 Session 对象，同时在客户端的浏览器端创建了一个 Cookie 对象；通过客户端带上来 Cookie 对象来与服务器端的 session 对象匹配来实现状态管理的。默认的，当我们关闭浏览器的时候，cookie 会被删除。但可以通过修改 cookie 的 expire time 使 cookie 在一定时间内有效；

OAuth

OAuth（开放授权）是一个开放的授权标准，允许用户让第三方应用访问该用户在某一 web 服务上存储的私密的资源（如照片，视频，联系人列表），而无需将用户名和密码提供给第三方应用。

这种基于 OAuth 的认证机制适用于个人消费者类的互联网产品，如社交类 APP 等应用

Token

json_web_token

CSRF

HTTP 常用认证机制

1.4 参考

HTTP 基本认证和摘要认证：https://blog.csdn.net/xcl168/article/details/49475381

详解 HTTP 中的摘要认证机制：https://blog.csdn.net/tenfyguo/article/details/8661517

HTTP 授权验证：https://www.jianshu.com/p/ebc297b51b3e

HTTP 认证与 https 简介：https://www.cnblogs.com/xzwblog/p/6834663.html

CHAPTER 2

http 认证鉴权 02CAS 和 SSO(单点登录)

单点登录（Single Sign-On , 简称 SSO ）是目前比较流行的服务于企业业务整合的解决方案之一，SSO 使得在多个应用系统中，用户只需要 **登录一次**就可以访问所有相互信任的应用系统

2.1 主要角色

1、User （多个）

2、Web 应用（多个）

3、SSO 认证中心（**1 个**）

2.2 实现模式的原则

SSO 实现模式一般包括以下三个原则：

1、所有的认证登录都在 SSO 认证中心进行；

2、SSO 认证中心通过一些方法来告诉 Web 应用当前访问用户究竟是不是已通过认证的用户；

3、SSO 认证中心和所有的 Web 应用建立一种信任关系，也就是说 web 应用必须信任认证中心。（单点信任）

2.3 CAS 的基本原理

结构

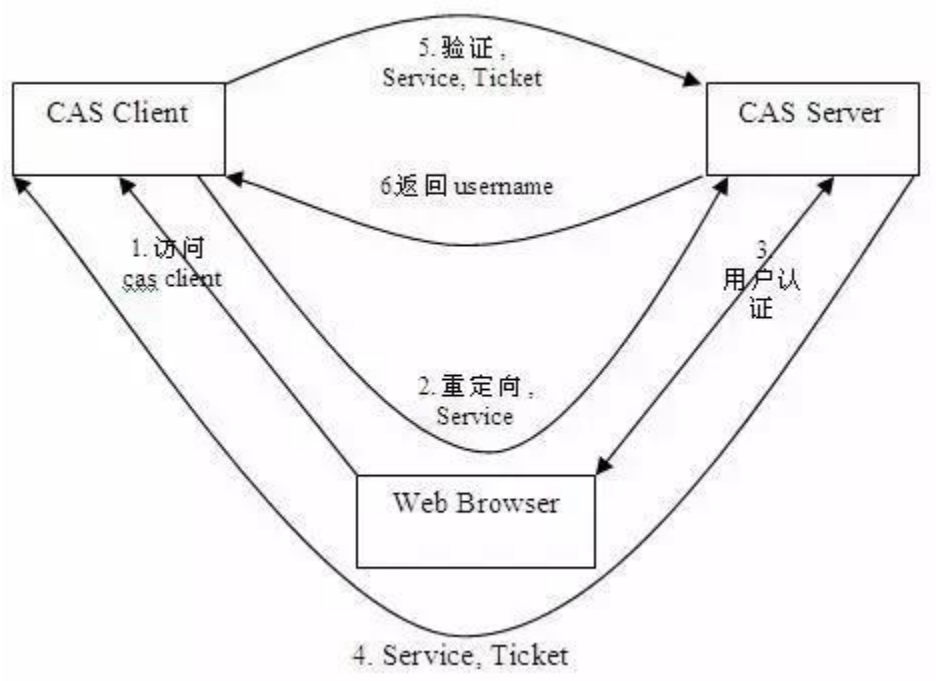
CAS 分为两部分，CAS Server 和 CAS Client

```
CAS Server 用来负责用户的认证工作，就像是把第一次登录用户的一个标识存在这里，以便此用户在其他系统登录
时验证其需不需要再次登录。
CAS Client 就是我们自己的应用，需要接入 CAS Server 端。当用户访问我们的应用时，首先需要重定向到
CAS Server 端进行验证，要是原来登陆过，就免去登录，重定向到下游系统，否则进行用户名密码登陆操作。
```

术语

```
Ticket Granting ticket (TGT) ：可以认为是 CAS Server 根据用户名密码生成的一张票，存在 server 端
Ticket-granting cookie (TGC) ：其实就是一个 cookie，存放用户身份信息，由 server 发给 client 端
Service ticket (ST) ：由 TGT 生成的一次性票据，用于验证，只能用一次。相当于 server 发给 client
一张票，然后 client 拿着这是个票再来找 server 验证，看看是不是 server 签发的。就像是我给了你一张我的
照片，然后你拿照片再来问我，这个照片是不是你。。。没错，就是这么无聊。
```

基础协议图



如上图：CAS Client 与受保护的客户端应用部署在一起，以 Filter 方式保护 Web 应用的受保护资源，过滤从客户端过来的每一个 Web 请求，同时，CAS Client 会分析 HTTP 请求中是否包含请求 Service Ticket(ST 上图中的 Ticket)，如果没有，则说明该用户是没有经过认证的；于是 CAS Client 会重定向用户请求到 CAS Server（Step 2），并传递 Service （要访问的目的资源地址）。Step 3 是用户认证过程，如果用户提供了正确的 Credentials，CAS Server 随机产生一个相当长度、唯一、不可伪造的 Service Ticket，并缓存以待将来验证，并且重定向用户到 Service 所在地址（附带刚才产生的 Service Ticket），**并为客户端浏览器设置一个 Ticket Granted Cookie**

(**TGC**)；CAS Client 在拿到 Service 和新产生的 Ticket 过后，在 Step 5 和 Step6 中与 CAS Server 进行身份核实，以确保 Service Ticket 的合法性。

在该协议中，所有与 CAS Server 的交互均采用 SSL 协议，以确保 ST 和 TGC 的安全性。协议工作过程中会有 **2 次重定向**的过程。但是 CAS Client 与 CAS Server 之间进行 Ticket 验证的过程对于用户是透明的（使用 HttpsURLConnection）。

2.4 CAS 如何实现 SSO

当用户访问另一个应用的服务再次被重定向到 CAS Server 的时候，CAS Server 会主动获到这个 TGC cookie，然后做下面的事情：

如果 User 持有 TGC 且其还没失效，那么就走基础协议图的 Step4，达到了 SSO 的效果；

如果 TGC 失效，那么用户还是要重新认证 (走基础协议图的 Step3) 。

2.5 CAS 安全性

CAS 的安全性仅仅依赖于 SSL 。使用的是 secure cookie 。

TGC/PGT 安全性

对于一个 CAS 用户来说，最重要是要保护它的 TGC，如果 TGC 不慎被 CAS Server 以外的实体获得，Hacker 能够找到该 TGC，然后冒充 CAS 用户访问 **所有**授权资源。PGT 的角色跟 TGC 是一样的。

从基础模式可以看出，TGC 是 CAS Server 通过 SSL 方式发送给终端用户，因此，要截取 TGC 难度非常大，从而确保 CAS 的安全性。

TGT 的存活周期默认为 120 分钟

ST/PT 安全性

ST （Service Ticket）是通过 Http 传送的，因此网络中的其他人可以 Sniffer 到其他人的 Ticket 。CAS 通过以下几方面来使 ST 变得更加安全（事实上都是可以配置的）：

1、ST 只能使用一次

CAS 协议规定，无论 Service Ticket 验证是否成功，CAS Server 都会清除服务端缓存中的该 Ticket，从而可以确保一个 Service Ticket 不被使用两次。

2、ST 在一段时间内失效

CAS 规定 ST 只能存活一定的时间，然后 CAS Server 会让它失效。默认有效时间为 5 分钟。

3、ST 是基于随机数生成的

ST 必须足够随机，如果 ST 生成规则被猜出，Hacker 就等于绕过 CAS 认证，直接访问对应的服务。

2.6 CAS 和 SSO 关系

SSO 仅仅是一种架构，一种设计，而 **CAS 则是实现 SSO 的一种手段**。两者是**抽象与具体的关系**。当然，除了 CAS 之外，实现 SSO 还有其他手段，比如简单的 cookie。

2.7 参考

一篇文章彻底弄懂 CAS 实现 SSO 单点登录原理：https://www.cnblogs.com/wangsongbai/p/10299655.html

什么是 SSO 与 CAS?：https://www.cnblogs.com/btgyoyo/p/10722010.html

基于 CAS 实现 SSO 单点登录：https://zhuanlan.zhihu.com/p/25007591

CHAPTER 3

http 认证鉴权 03OAuth2 入门

OAuth（Open Authorization，开放授权）是为用户资源的授权定义了一个安全、开放及简单的标准，第三方无需知道用户的账号及密码，就可获取到用户的授权信息。

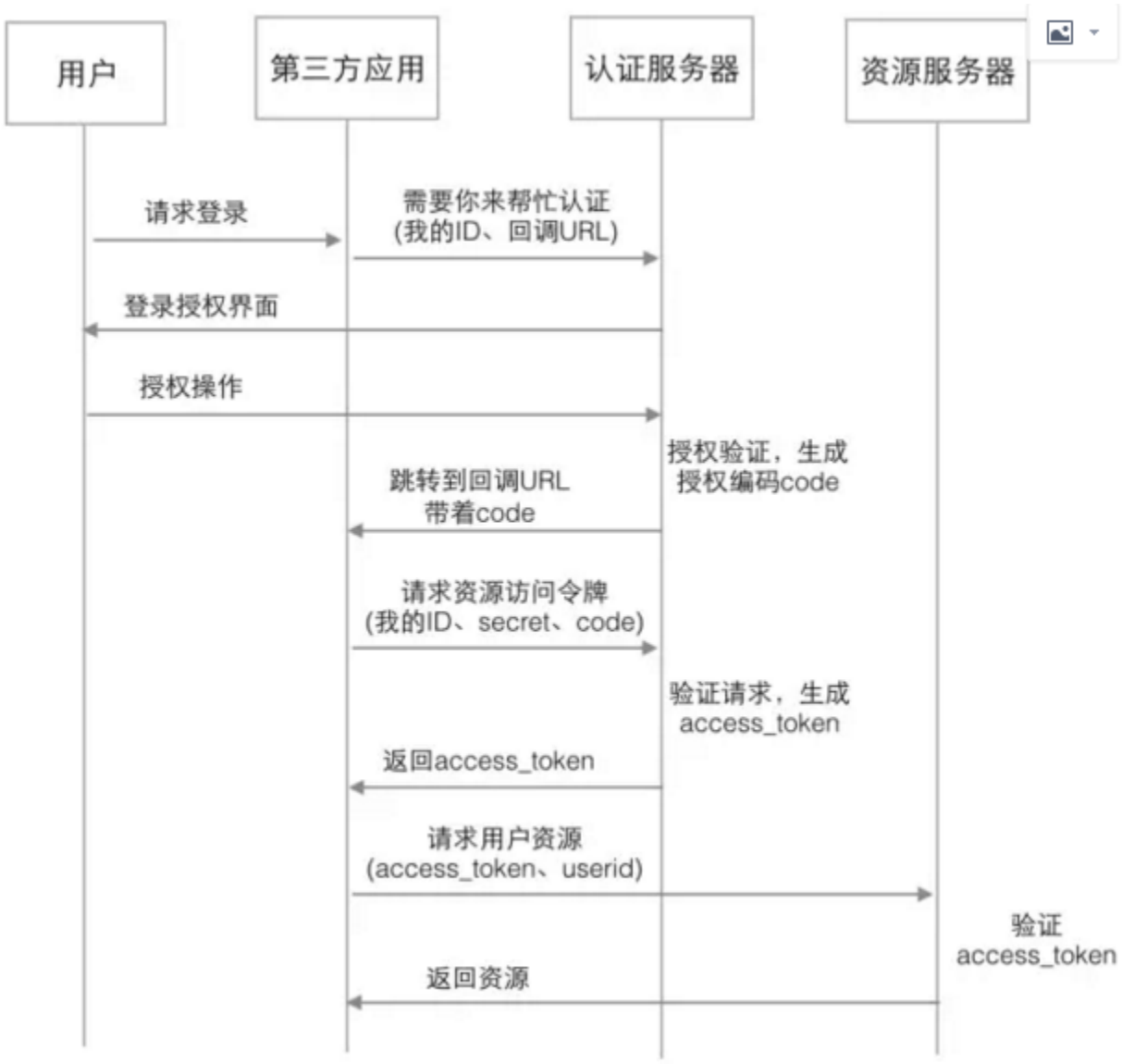
OAuth 在”客户端”与”服务提供商”之间，设置了一个授权层，”客户端”不能直接登录”服务提供商”，只能登录授权层，以此将用户与客户端区分开来，”客户端”登录授权层是使用令牌（token），”客户端”登录授权层以后，”服务提供商”根据令牌的权限范围和有效期，向”客户端”开放用户储存的资料

3.1 实现机制

在 OAuth2 的授权机制中有 4 个核心对象

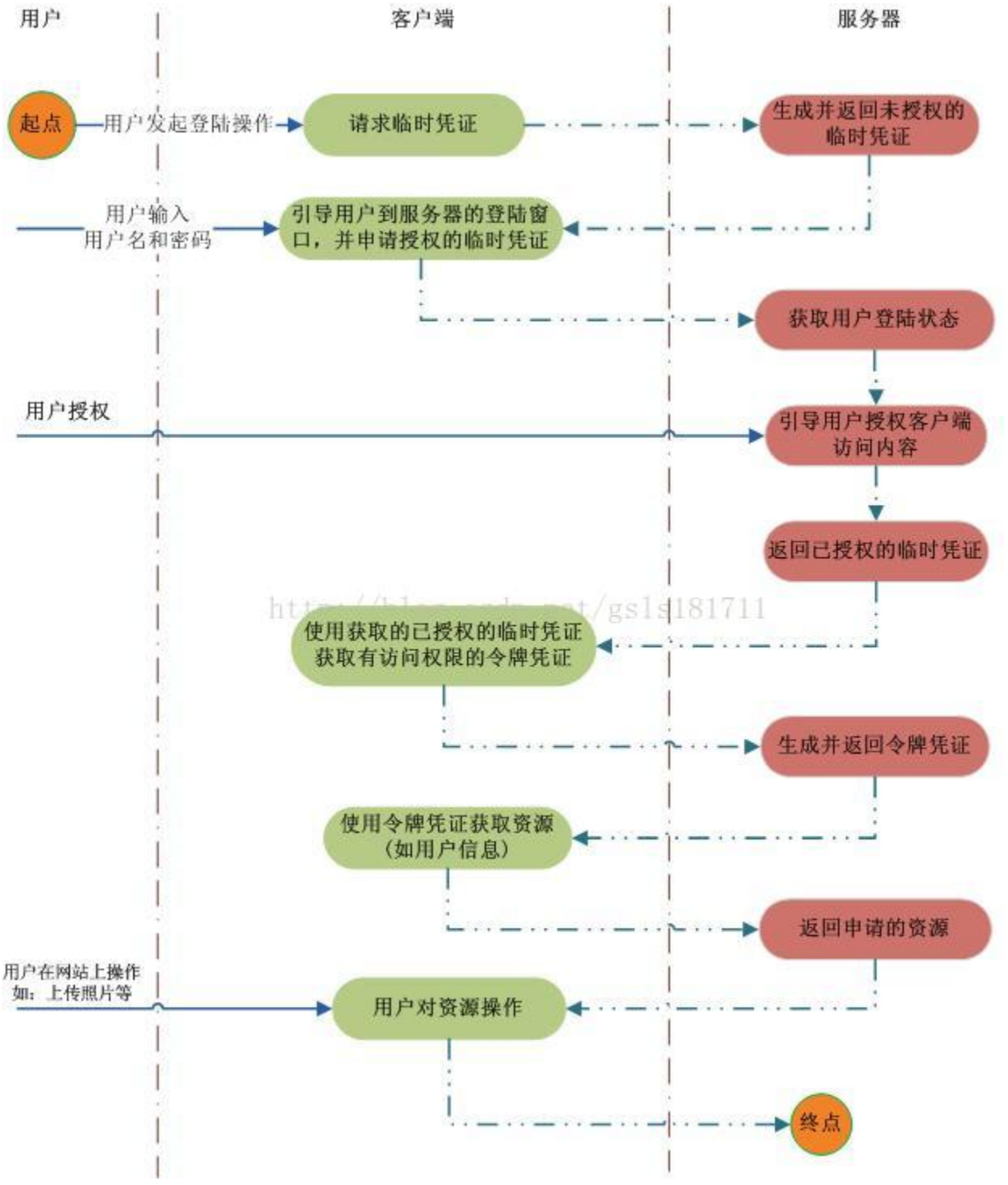
```
(1) Resource Owner（资源拥有者：用户）
(2) Client （第三方接入平台：请求者，例如网站）
(3) Resource Server （资源服务器，存储例如用户信息等资源）
(4) Authorization Server （认证服务器）
```

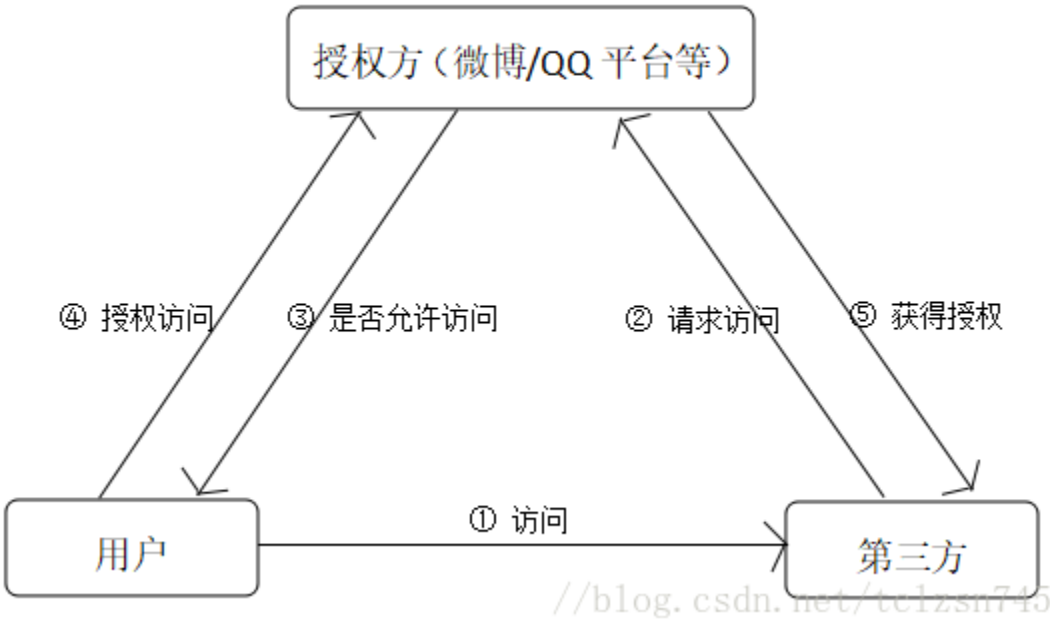
步骤:



(1) 用户在第三方应用上点击登录，应用向认证服务器发送请求，说有用户希望进行授权操作，同时说明自己是谁、用户授权完成后的回调 url
(2) 认证服务器展示给用户自己的授权界面
(3) 用户进行授权操作，认证服务器验证成功后，生成一个授权编码 code，并跳转到第三方的回调 url
(4) 第三方应用拿到 code 后，连同自己在平台上的身份信息（ID 密码）发送给认证服务器，再一次进行验证请求，说明自己的身份正确，并且用户也已经授权我了，来换取访问用户资源的权限
(5) 认证服务器对请求信息进行验证，如果没问题，就生成访问资源服务器的令牌 access_token，交给第三方应用
(6) 第三方应用使用 access_token 向资源服务器请求资源
(7) 资源服务器验证 access_token 成功后返回响应资源

其他流程图


用户
客户端
服务器
起点
用户发起登陆操作
请求临时凭证
生成并返回未授权的临时凭证
用户输入用户名和密码
引导用户到服务器的登陆窗口，并申请授权的临时凭证
获取用户登陆状态
用户授权
引导用户授权客户端访问内容
返回已授权的临时凭证
使用获取的已授权的临时凭证获取有访问权限的令牌凭证
生成并返回令牌凭证
使用令牌凭证获取资源（如用户信息）
返回申请的资源
用户在网站上操作如：上传照片等
用户对资源操作
终点



授权码模式（authorization code）是功能最完整、流程最严密的授权模式。它的特点就是通过客户端的后台服务器，与”服务提供商”的认证服务器进行互动。

```
+----------+
| Resource |
|   Owner  |
|          |
+----------+
     ^
     |
    (B)
+----|-----+          Client Identifier      +---------------+
|         -+----(A)-- & Redirection URI ---->|               |
|  User-   |                                 | Authorization |
|  Agent  -+----(B)-- User authenticates --->|     Server    |
|          |                                 |               |
|         -+----(C)-- Authorization Code ---<|               |
+-|----|---+                                 +---------------+
  |    |                                         ^      v
 (A)  (C)                                        |      |
  |    |                                         |      |
  ^    v                                         |      |
+---------+                                      |      |
|         |>---(D)-- Authorization Code ---------'      |
|  Client |          & Redirection URI                  |
|         |                                             |
|         |<---(E)----- Access Token -------------------'
+---------+       (w/ Optional Refresh Token)
```

步骤

（A）用户访问客户端，后者将前者导向认证服务器。
（B）用户选择是否给予客户端授权。
（C）假设用户给予授权，认证服务器将用户导向客户端事先指定的" 重定向 URI"（redirection URI），同时附上一个授权码。
（D）客户端收到授权码，附上早先的" 重定向 URI"，向认证服务器申请令牌。这一步是在客户端的后台的服务器上完成的，对用户不可见。
（E）认证服务器核对了授权码和重定向 URI，确认无误后，向客户端发送访问令牌（access token）和更新令牌（refresh token）。

3.2 参考

OAuth2 认证原理:https://blog.csdn.net/fsy9595887/article/details/85114508

OAuth2 实现原理:https://www.cnblogs.com/chinanetwind/articles/9457842.html

OAuth2.0 认证原理浅析:https://blog.csdn.net/tclzsn7456/article/details/79550249

CAS 与 OAuth2 的区别：https://www.cnblogs.com/boboooo/p/9041355.html

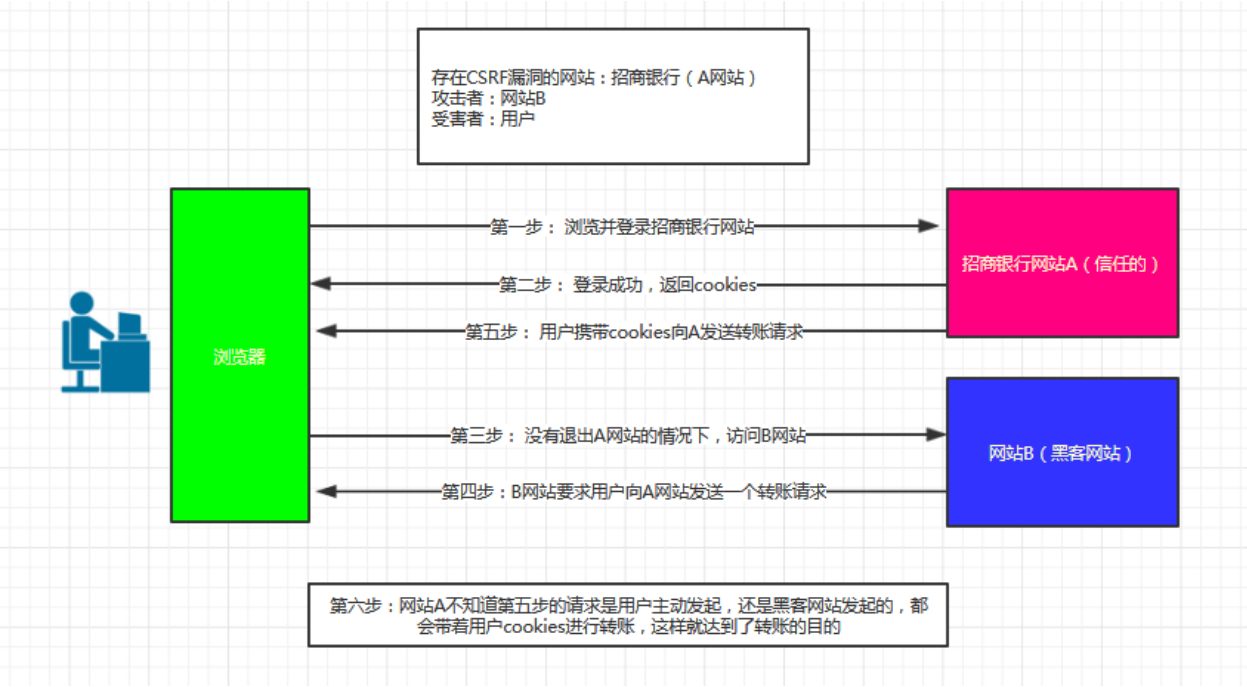
理解 OAuth 2.0：https://www.ruanyifeng.com/blog/2014/05/oauth_2_0.html

CHAPTER 4

http 认证鉴权 04CSRF 跨站请求伪造

攻击者盗用了你的身份，以你的名义发送恶意请求，对服务器来说这个请求是完全合法的，但是却完成了攻击者所期望的一个操作，比如以你的名义发送邮件、发消息，盗取你的账号，添加系统管理员，甚至于购买商品、虚拟货币转账等。如下：其中 Web A 为存在 CSRF 漏洞的网站，Web B 为攻击者构建的恶意网站，User C 为 Web A 网站的合法用户

4.1 CSRF 攻击原理



从上图可以看出，要完成一次 CSRF 攻击，受害者必须依次完成两个步骤：

```
1. 登录受信任网站 A，并在本地生成 Cookie。
2. 在不登出 A 的情况下，访问危险网站 B。
```

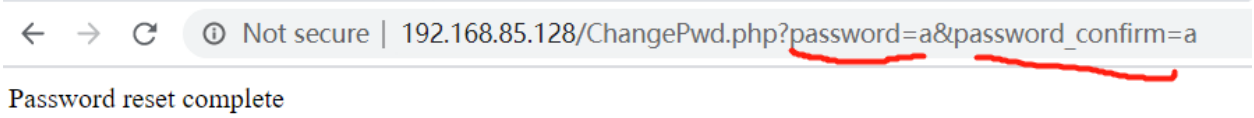
看到这里，你也许会说："如果我不满足以上两个条件中的一个，我就不会受到 CSRF 的攻击"。是的，确实如此，但你不能保证以下情况不会发生：

```
1. 你不能保证你登录了一个网站后，不再打开一个 tab 页面并访问另外的网站。
2. 你不能保证你关闭浏览器了后，你本地的 Cookie 立刻过期，你上次的会话已经结束。(事实上，** 关闭浏览器
不能结束一个会话 **，但大多数人都会错误的认为关闭浏览器就等于退出登录/结束会话了.....)
3. 上图中所谓的攻击网站，可能是一个存在其他漏洞的可信任的经常被人访问的网站。
```

4.2 攻击实例 (get 为例,post 亦可, 见参考文献)

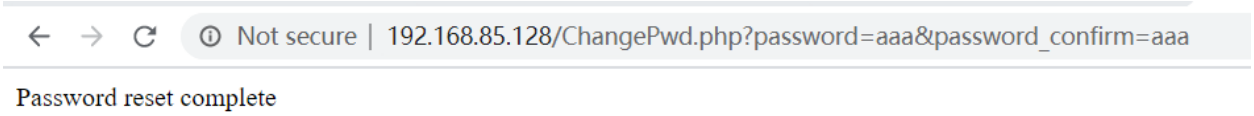
已有网站界面如下：

假设现在用户 a 需要修改密码，用户输入密码以后，界面如下：



用户 A 通过**抓包或者观察 URL** 发现，当修改密码时，向服务器发送了两个参数，password = a 和 password_confirm = a。那么如果将这个链接发送给别人，就可以修改别人的密码了。

于是，用户 A 将 URL http://192.168.85.128/ChangePwd.php?password=aaa&password_confirm=aaa 发送给了用户 admin，并附上一些**诱导点开**的话。由于**此网站有“保持登陆状态”的设置**，当用户 admin 点击了这条 URL 的时候，发现自己的密码已经被修改了。



如果 admin 并没有发现问题，关闭了网页。结果下次登陆网站的时候，发现自己的账户密码已经被修改了，仅仅是因为点击了一条 URL!

而发送链接的用户 A，也可以轻松使用修改后的密码进行登陆了。

4.3 解决 CSRF 攻击

使用 csrf_token 校验

4.3.1 flask 的 csrf 配置

1. 客户端和浏览器向后端发送请求时，后端往往会在响应中的 cookie 设置 csrf_token 的值，可以使用请求钩子实现，在 cookie 中设置 csrf_token

```
from flask_wtf.csrf import generate_csrf
@app.after_request
def after_request(resp):
    # 调用表单方法，获取csrt_token
    csrf_token = generate_csrf()
    # cookie设置csrf值
    resp.set_cookie("csrf_token", csrf_token)
    return resp
```


2.flask_wtf 中为我们提供了 CSRF 保护，可以直接调用开启对 app 的保护

```
from flask_wtf import CSRFProtect
# 设置csrf对app 进行保护
CSRFProtect(app)
```


一旦开启 CSRF 保护，就要设置秘钥：SECRET_KEY

```
# 设置秘钥
SECRET_KEY = "jdfsaohfuds"
```


csrf 验证

1. 表单提交方式

服务器通过请求钩子在 cookie 中设置了 csrf_token, 实际上是在 session 中存储了未加密的 csrf_token，并且将生成的 sessionID 编号存储在 cookie 中

在表单中我们添加了 csrf 的隐藏字段，在浏览器再次访问服务器时：

```
1. 获取到表单中的 csrf_token(加密的)，使用 secret_key 进行解密，得到解密后的 csrf_token
2. 通过 cookie 中的 sessionID，取到服务器内部存储的 session 中的 csrf_token(未加密的)
3. 将两者的值进行比较
```

2.ajax 提交请求方式

在 js 里面，获取到 cookie 中的 csrf_token, 将其添加到 ajax 的请求头中

```
$.ajax({
    url: "/news/news_collect",
    type: "post",
    contentType: "application/json",
    headers: {
        "X-CSRFToken": getCookie("csrf_token")
    }
```


验证过程:

```
1. 获取请求头中的 csrf_token (加密的)，然后使用 secret_key 解密，得到解密后 csrf_token。
2. 通过 cookie 中的 sessionID，取到服务器内部存储的 session 中的 csrf_token(未加密的)。
3. 将两者的值进行比较
```

4.3.2 django 的 csrf 配置

配置修改

django:settings.py

```
'django.middleware.csrf.CsrfViewMiddleware' # 认证系统, 如果不加, 则不进行认证
```

django 模板渲染: 先进行 django 模板渲染 (render)，再返回浏览器

```
<form action="" method="post">
    {% csrf_token %}  # post 请求加上 csrf_token(安全令牌), 每次 POST 请求都会令牌比对, 写在
form 表单的任意位置
    username: <input type="text" name="username">
    password: <input type="text" name="password">
    <input type="submit">
</form>
```

浏览器 get 请求, 服务器响应包含 post 请求的 html 页面, 服务器后端会自动保存一份 name=csrfmiddlewaretoken, 和 value 值的数据以备浏览器 post 认证，浏览器 post 请求时会加上 name 和 value 值，服务器端收到 post 请求后会比对 value 值，如果匹配则响应 post 请求，如果匹配不成功则拒绝响应.

csrf_token 验证：post

方法 1: 前端 form 表单中取隐藏标签属性值放入 data 中 post 到后端, contentType: urlencoded 适用

```
$.ajax({
        data:{
                csrfmiddlewaretoken:$('[name="csrfmiddlewaretoken"]').val()
                }
        })
```

方法 2:ajaxSetup：django 将 csrftoken 传送到前端，前端 post 时携带这个值,

```
$.ajaxSetup({data:csrfmiddlewaretoken='{{csrf_token}}})
```

方法 3: 发送 contenttype 类型数据时, 通过获取响应返回的 cookie 中的字符串，放置在请求头中发送。需要引入一个 jquery.cokkie.js 插件—–json，form-data 适用

```
{%load static%}
<script src="{% static 'js/jquery.cookie.js'%}"></script>

$.ajax{{
        headers:{"X-CSRFToken":$.cookie("csrftoken")},
}}
```

4.4 参考

浅谈 CSRF（Cross-site request forgery）跨站请求伪造（写的非常好）：https://www.cnblogs.com/liuqingzheng/p/9505044.html

web 安全学习笔记（九）CSRF（Cross-Site Request Forgery）跨站请求伪造：https://blog.csdn.net/qycc3391/article/details/104741756

csrf 验证机制：https://blog.csdn.net/wireless911/article/details/81589202

csrf 认证：https://www.cnblogs.com/relaxlee/p/12842639.html

CSRF 认证的几种方法：https://blog.csdn.net/mildddd/article/details/81083088

CHAPTER 5

Indices and tables

- genindex
- modindex
- search