[帮你深入理解OAuth2.0协议](http://blog.csdn.net/seccloud/article/details/8192707)

分类： [Web安全协议](http://blog.csdn.net/seccloud/article/category/1262564)2012-11-16 23:50 16447人阅读 [评论](http://blog.csdn.net/seccloud/article/details/8192707#comments)(9) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/seccloud/article/details/8192707#report)

[oAuth2.0](http://www.csdn.net/tag/oAuth2.0)[OpenAuthorization](http://www.csdn.net/tag/OpenAuthorization)[Web安全协议](http://www.csdn.net/tag/Web%e5%ae%89%e5%85%a8%e5%8d%8f%e8%ae%ae)[开放授权](http://www.csdn.net/tag/%e5%bc%80%e6%94%be%e6%8e%88%e6%9d%83)

**1. 引言**

如果你开车去酒店赴宴，你经常会苦于找不到停车位而耽误很多时间。是否有好办法可以避免这个问题呢？有的，听说有一些豪车的车主就不担心这个问题。豪车一般配备两种钥匙：主钥匙和泊车钥匙。当你到酒店后，只需要将泊车钥匙交给服务生，停车的事情就由服务生去处理。与主钥匙相比，这种泊车钥匙的使用功能是受限制的：它只能启动发动机并让车行驶一段有限的距离，可以锁车，但无法打开后备箱，无法使用车内其他设备。这里就体现了一种简单的“开放授权”思想：通过一把泊车钥匙，车主便能将汽车的部分使用功能（如启动发动机、行驶一段有限的距离）授权给服务生。

授权是一个古老的概念，它是一个多用户系统必须支持的功能特性。比如，Alice和Bob都是Google的用户，那么Alice应该可以将自己的照片授权给Bob访问。但请注意到，这种授权是一种封闭授权，它只支持系统内部用户之间的相互授权，而不能支持与其他外部系统或用户之间的授权。比如说，Alice想使用“网易印像服务”将她的部分照片冲印出来，她怎么能做到呢？

肯定有人会说，Alice可以将自己的Google用户名和密码告诉网易印像服务，事情不就解决了吗？是的，但只有毫不关注安全和隐私的同学才会出此“绝招”。那么我们就来想一想，这一“绝招”存在哪些问题？(1) 网易印像服务可能会缓存Alice的用户名和密码，而且可能没有加密保护。它一旦遭到攻击，Alice就会躺着中枪。(2) 网易印像服务可以访问Alice在Google上的所有资源，Alice无法对他们进行最小的权限控制，比如只允许访问某一张照片，1小时内访问有效。(3) Alice无法撤消她的单个授权，除非Alice更新密码。

在以Web服务为核心的云计算时代，像用户Alice的这种授权需求变得日益迫切与兴盛，“开放授权(Open Authorization)”也正因此而生，意在帮助Alice将她的资源授权给第三方应用，支持细粒度的权限控制，并且不会泄漏Alice的密码或其它认证凭据。

根据应用场景的不同，目前实现开放授权的方法分为两种：一种是使用OAuth协议[1]；另一种是使用IAM服务[2]。OAuth协议主要适用于针对个人用户对资源的开放授权，比如Google的用户Alice。OAuth的特点是“现场授权”或“在线授权”：客户端主要通过浏览器去访问资源，授权时需要认证Alice的资源所有者身份，并且需要Alice现场审批。OAuth一般在SNS服务中广泛使用，如微博。IAM服务则不同，它的特点是“预先授权”或“离线授权”：客户端主要通过REST API方式去访问资源，资源所有者可以预先知道第三方应用所需要的资源请求，一次授权之后，很少会变更。IAM服务一般在云计算服务中使用，如AWS服务、阿里云计算服务。

本文主要介绍OAuth开放授权。关于以IAM服务提供的开放授权，我将在另一篇博文中介绍。下面我来介绍OAuth 2.0协议、协议的实例化描述、安全性分析。

**2. OAuth 2.0 协议**  
  
OAuth 2.0 是目前比较流行的做法，它率先被Google, Yahoo, Microsoft, Facebook等使用。之所以标注为 2.0，是因为最初有一个1.0协议，但这个1.0协议被弄得太复杂，易用性差，所以没有得到普及。2.0是一个新的设计，协议简单清晰，但它并不兼容1.0，可以说与1.0没什么关系。所以，我就只介绍2.0。  
  
**2.1 协议的参与者**

从引言部分的描述我们可以看出，OAuth的参与实体至少有如下三个：

**· RO** (resource owner): 资源所有者，对资源具有授权能力的人。如上文中的用户Alice。

**· RS** (resource server): 资源服务器，它存储资源，并处理对资源的访问请求。如Google资源服务器，它所保管的资源就是用户Alice的照片。

**· Client**: 第三方应用，它获得RO的授权后便可以去访问RO的资源。如网易印像服务。

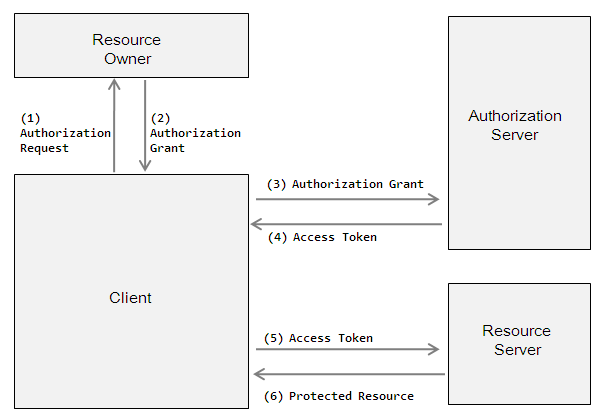
此外，为了支持开放授权功能以及更好地描述开放授权协议，OAuth引入了第四个参与实体：

**· AS** (authorization server): 授权服务器，它认证RO的身份，为RO提供授权审批流程，并最终颁发授权令牌(Access Token)。读者请注意，为了便于协议的描述，这里只是在逻辑上把AS与RS区分开来；在物理上，AS与RS的功能可以由同一个服务器来提供服务。

**2.2 授权类型**

在开放授权中，第三方应用(Client)可能是一个Web站点，也可能是在浏览器中运行的一段JavaScript代码，还可能是安装在本地的一个应用程序。这些第三方应用都有各自的安全特性。对于Web站点来说，它与RO浏览器是分离的，它可以自己保存协议中的敏感数据，这些密钥可以不暴露给RO；对于JavaScript代码和本地安全的应用程序来说，它本来就运行在RO的浏览器中，RO是可以访问到Client在协议中的敏感数据。

OAuth为了支持这些不同类型的第三方应用，提出了多种授权类型，如授权码 (Authorization Code Grant)、隐式授权 (Implicit Grant)、RO凭证授权 (Resource Owner Password Credentials Grant)、Client凭证授权 (Client Credentials Grant)。由于本文旨在帮助用户理解OAuth协议，所以我将先介绍这些授权类型的基本思路，然后选择其中最核心、最难理解、也是最广泛使用的一种授权类型——“授权码”，进行深入的介绍。

**2.3 OAuth协议 - 基本思路**  
  
  
[Figure 1: Abstract Protocol Flow]  
  
如图1所示，协议的基本流程如下：

(1) Client请求RO的授权，请求中一般包含：要访问的资源路径，操作类型，Client的身份等信息。

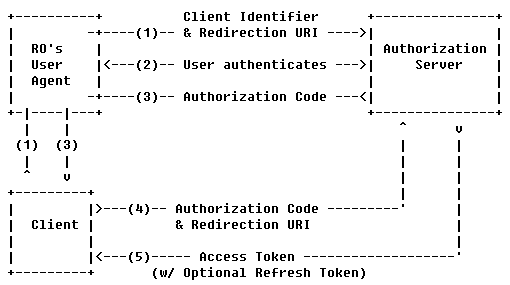
(2) RO批准授权，并将“授权证据”发送给Client。至于RO如何批准，这个是协议之外的事情。典型的做法是，AS提供授权审批界面，让RO显式批准。这个可以参考下一节实例化分析中的描述。

(3) Client向AS请求“访问令牌(Access Token)”。此时，Client需向AS提供RO的“授权证据”，以及Client自己身份的凭证。

(4) AS验证通过后，向Client返回“访问令牌”。访问令牌也有多种类型，若为bearer类型，那么谁持有访问令牌，谁就能访问资源。

(5) Client携带“访问令牌”访问RS上的资源。在令牌的有效期内，Client可以多次携带令牌去访问资源。

(6) RS验证令牌的有效性，比如是否伪造、是否越权、是否过期，验证通过后，才能提供服务。

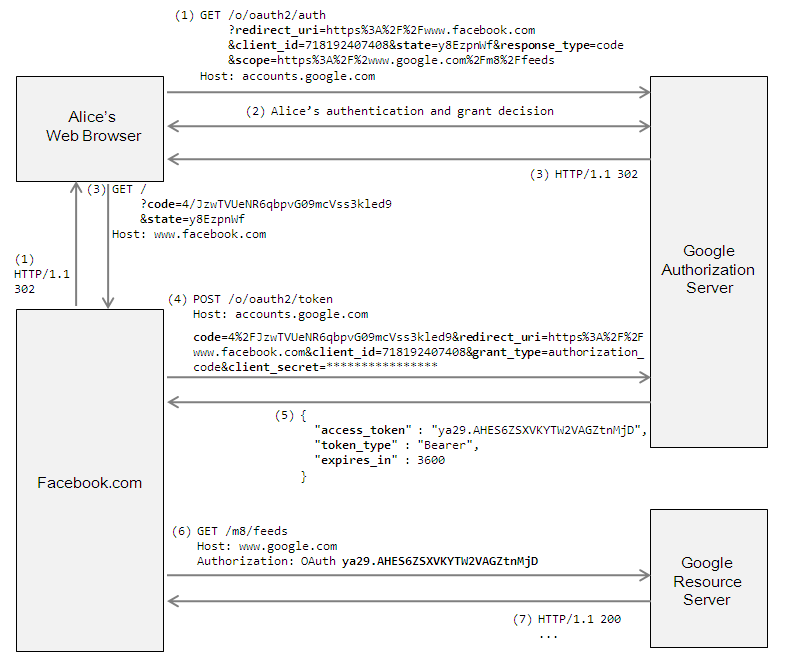
**2.4 授权码类型的开放授权**  
  
  
  
[Figure 2: Authorization Code Flow]  
  
如图2所示，授权码类型的开放授权协议流程描述如下：

(1) Client初始化协议的执行流程。首先通过HTTP 302来重定向RO用户代理到AS。Client在redirect\_uri中应包含如下参数：client\_id, scope (描述被访问的资源), redirect\_uri (即Client的URI), state (用于抵制CSRF攻击). 此外，请求中还可以包含access\_type和approval\_prompt参数。当approval\_prompt=force时，AS将提供交互页面，要求RO必须显式地批准（或拒绝）Client的此次请求。如果没有approval\_prompt参数，则默认为RO批准此次请求。当access\_type=offline时，AS将在颁发access\_token时，同时还会颁发一个refresh\_token。因为access\_token的有效期较短（如3600秒），为了优化协议执行流程，offline方式将允许Client直接持refresh\_token来换取一个新的access\_token。

(2) AS认证RO身份，并提供页面供RO决定是否批准或拒绝Client的此次请求（当approval\_prompt=force时）。

(3) 若请求被批准，AS使用步骤(1)中Client提供的redirect\_uri重定向RO用户代理到Client。redirect\_uri须包含authorization\_code，以及步骤1中Client提供的state。若请求被拒绝，AS将通过redirect\_uri返回相应的错误信息。

(4) Client拿authorization\_code去访问AS以交换所需的access\_token。Client请求信息中应包含用于认证Client身份所需的认证数据，以及上一步请求authorization\_code时所用的redirect\_uri。

(5) AS在收到authorization\_code时需要验证Client的身份，并验证收到的redirect\_uri与第3步请求authorization\_code时所使用的redirect\_uri相匹配。如果验证通过，AS将返回access\_token，以及refresh\_token（若access\_type=offline）。  
  
如果读者对这个流程的细节不甚清楚，那么可以先看第3节的一个实例化描述，然后再回来看这部分内容。  
  
**3. OAuth协议实例化描述**  
  
下面我以实例化方式来帮助读者理解授权码类型的授权协议的运行过程。假设:   
(1) Alice有一个有效的Google帐号；  
(2) Facebook.com已经在Google Authorization Server上注册了Client身份，已经获得(client\_id, client\_secret)，注意client\_secret是Client与AS之间的一个共享密钥。  
(3) Alice想授权Facebook.com查看她的联系人列表(https://www.google.com/m8/feeds)。  
  
图3展示了Alice、Facebook.com、Google资源服务器、以及Google OAuth授权服务器之间的协议运行过程。  
  
  
[Figure 3: An Instance of Authorization Code Flow]    
//若字体无法看清，请单击右键->选择查看原图

协议所涉及到的细节都已经在图3上了，所以不打算再做详细介绍了。若看懂了此图，OAuth2.0就理解了。

读者请注意，在步骤(4)中，Client需要拿“授权码”去换“授权令牌”时，Client需要向AS证明自己的身份，即证明自己就是步骤(2)中Alice批准授权时的Grantee。这个身份证明的方法主要有两种（图3中使用了第1种）：  
(1) 通过https直接将client\_secret发送给AS，因为client\_secret是由Client与AS所共享，所以只要传送client\_secret的信道安全即可。  
(2) 通过消息认证码来认证Client身份，典型的算法有HMAC-SHA1。在这种方式下，Client无需传送client\_secret，只需发送消息请求的signature即可。由于不需要向AS传递敏感数据，所以它只需要使用http即可。

此外，在步骤(2)中，Google授权服务器需要认证Alice的RO身份，并提供授权界面给Alice进行授权审批。今天Google提供的实例如图4、图5所示，仅供读者理解OAuth这种“现场授权”或"在线授权"的含义。

  
[Figure 4: RO's Identity Authentication]  
  
  
[Figure 5: RO's Authorization Decision]  
  
**4. OAuth设计上的安全性考虑  
  
4.1 为何引入authorization\_code？**  
  
协议设计中，为什么要使用authorization\_code来交换access\_token？这是读者容易想到的一个问题。也就是说，在协议的第3步，为什么不直接将access\_token通过重定向方式返回给Client呢？比如:  
  
HTTP/1.1 302  
Location:  
https://www.facebook.com/?access\_token=ya29.AHES6ZSXVKYTW2VAGZtnMjD&token\_type=Bearer&expires\_in=3600  
  
如果直接返回access\_token，协议将变得更加简洁，而且少一次Client与AS之间的交互，性能也更优。那为何不这么设计呢？协议文档[1]中并没有给出这样设计的理由，但也不难分析：

(1) 浏览器的redirect\_uri是一个不安全信道，此方式不适合于传递敏感数据（如access\_token）。因为uri可能通过HTTP referrer被传递给其它恶意站点，也可能存在于浏览器cacher或log文件中，这就给攻击者盗取access\_token带来了很多机会。另外，此协议也不应该假设RO用户代理的行为是可信赖的，因为RO的浏览器可能早已被攻击者植入了跨站脚本用来监听access\_token。因此，access\_token通过RO的用户代理传递给Client，会显著扩大access\_token被泄露的风险。 但authorization\_code可以通过redirect\_uri方式来传递，是因为authorization\_code并不像access\_token一样敏感。即使authorization\_code被泄露，攻击者也无法直接拿到access\_token，因为拿authorization\_code去交换access\_token是需要验证Client的真实身份。也就是说，除了Client之外，其他人拿authorization\_code是没有用的。 此外，access\_token应该只颁发给Client使用，其他任何主体（包括RO）都不应该获取access\_token。协议的设计应能保证Client是唯一有能力获取access\_token的主体。引入authorization\_code之后，便可以保证Client是access\_token的唯一持有人。当然，Client也是唯一的有义务需要保护access\_token不被泄露。

(2) 引入authorization\_code还会带来如下的好处。由于协议需要验证Client的身份，如果不引入authorization\_code，这个Client的身份认证只能通过第1步的redirect\_uri来传递。同样由于redirect\_uri是一个不安全信道，这就额外要求Client必须使用数字签名技术来进行身份认证，而不能用简单的密码或口令认证方式。引入authorization\_code之后，AS可以直接对Client进行身份认证（见步骤4和5），而且可以支持任意的Client认证方式（比如，简单地直接将Client端密钥发送给AS）。

在我们理解了上述安全性考虑之后，读者也许会有豁然开朗的感觉，懂得了引入authorization\_code的妙处。那么，是不是一定要引入authorization\_code才能解决这些安全问题呢？当然不是。笔者将会在另一篇博文给出一个直接返回access\_token的扩展授权类型解决方案，它在满足相同安全性的条件下，使协议更简洁，交互次数更少。  
  
**4.2 基于Web安全的考虑**  
  
OAuth协议设计不同于简单的网络安全协议的设计，因为OAuth需要考虑各种Web攻击，比如CSRF (Cross-Site Request Forgery), XSS (Cross Site Script), Clickjacking。要理解这些攻击原理，读者需要对浏览器安全（eg, Same Origin Policy, 同源策略）有基本理解。比如，在redirect\_uri中引入state参数就是从浏览器安全角度考虑的，有了它就可以抵制CSRF攻击。如果没有这个参数，攻击者便可以在redirect\_uri中注入攻击者提供的authorization\_code或access\_token，结果可能导致Client访问错误的资源（比如，将款项汇到一个错误的帐号）。  
  
基于Web安全的考虑，OAuth协议文档中已经有了比较全面的阐述，所以我不打算在此文中进行展开，有兴趣的读者请参考[1]。  
  
**5. 结语**  
  
本文对OAuth 2.0 开放授权协议及其设计上的安全性考虑做了一个基本的介绍，希望能给参与安全协议设计和开发的同学起到一点帮助。  
  
**参考文献**:  
  
[1] Hammer-Lahav, E., Recordon, D., and D. Hardt, "The OAuth 2.0 Authorization Framework", draft-ietf-oauth-v2-31 (work in progress), June 2012.  
[2] http://aws.amazon.com/iam/