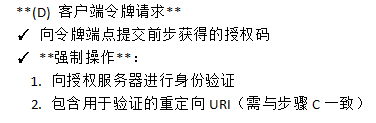
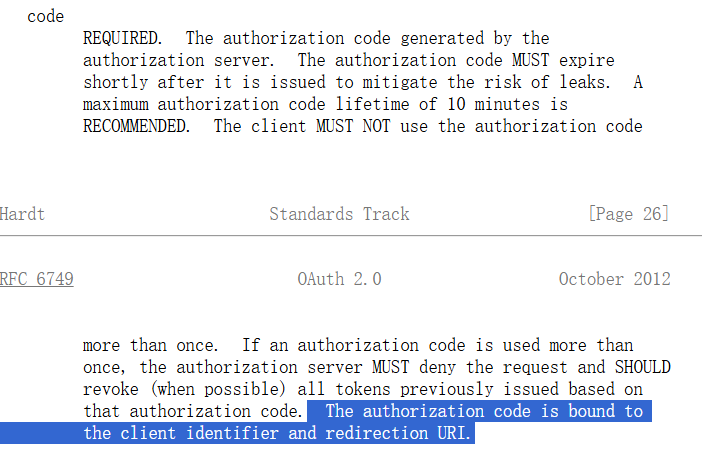


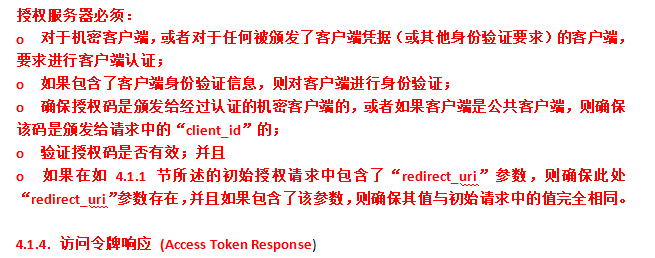
公开客户端的认证不靠谱



4.1.2

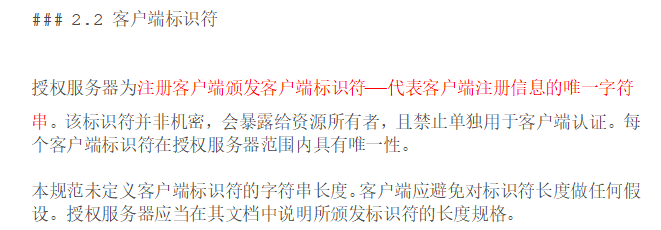


4.1.4



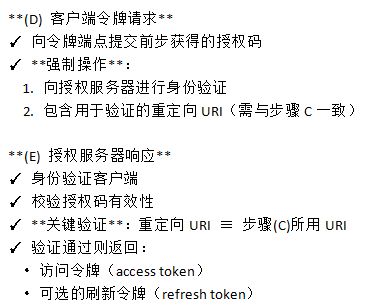
公开客户端可能出现问题

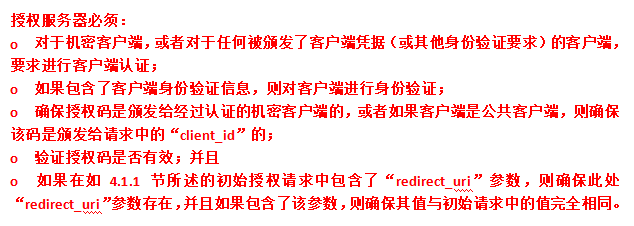
2.2



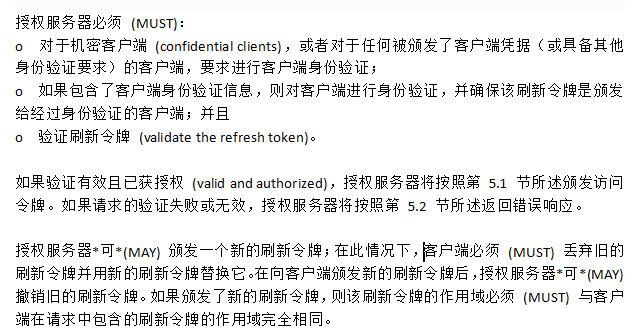
Client\_id并非机密

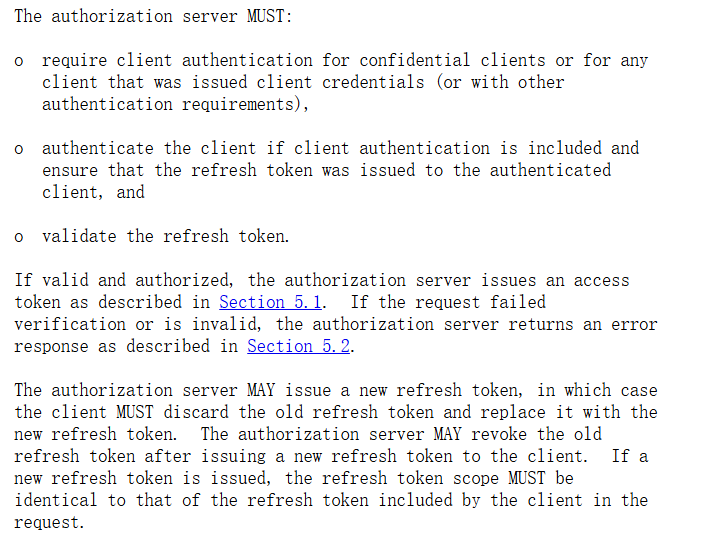
使用授权码申请token时需要对客户端进行身份验证



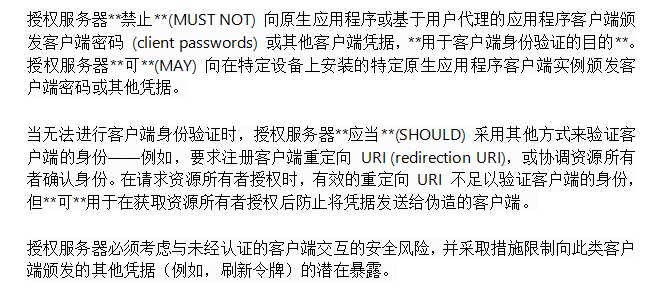


6 **Refreshing an Access Token**

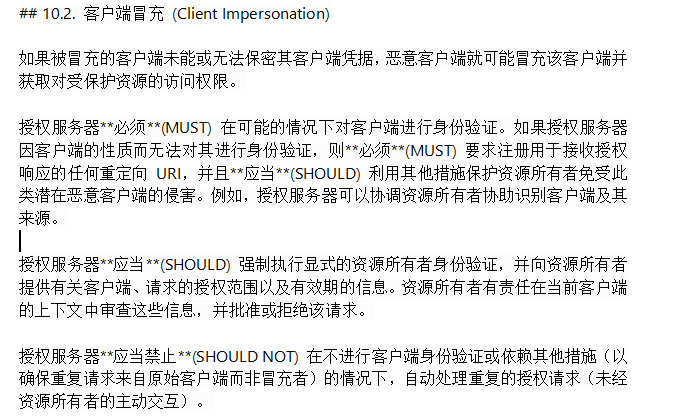




10.1

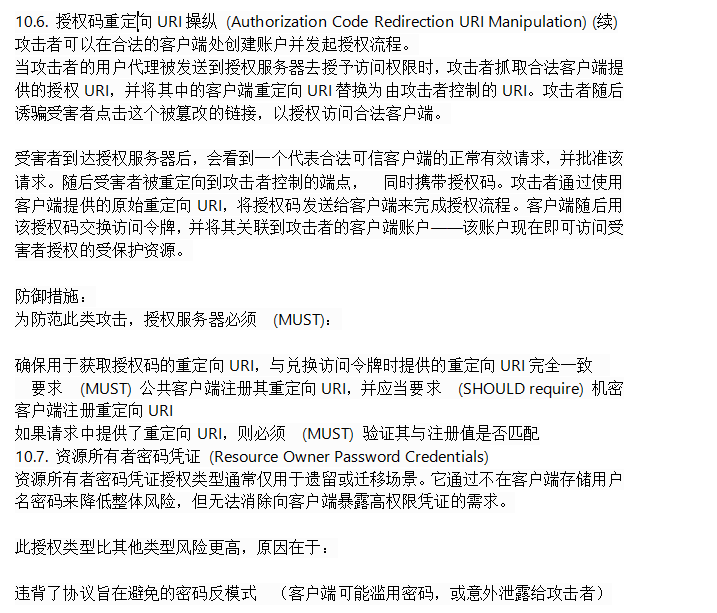


10.2









好的，我们来详细讲解一下这张图片中描述的 OAuth 2.0 授权码流程中的“重定向 URI 操作攻击”（Authorization Code Redirection URI Manipulation）。

​**核心问题：​**​ 攻击者通过篡改授权请求中的 redirect\_uri 参数，窃取受害者的授权码（Authorization Code），并最终获得该受害者授权给合法客户端的资源访问权限。

​**攻击背景：OAuth 2.0 授权码流程 (Authorization Code Grant)​**​

1. ​**用户（资源所有者）​**​ 使用 ​**用户代理**​（通常是浏览器）访问 ​**客户端应用**​（如一个第三方网站）。
2. 客户端需要访问用户在 ​**资源服务器**​（如 Google Drive）上的资源。
3. 客户端将用户重定向到 ​**授权服务器**​（通常是资源服务器的一部分，如 Google 登录页面）。
4. 在重定向请求中，客户端会包含重要的参数，特别是：
   1. response\_type=code：表示请求授权码。
   2. client\_id：标识客户端。
   3. redirect\_uri：告诉授权服务器在用户授权后，将用户代理（带授权码）重定向回哪里。
   4. scope：请求的权限范围。
   5. state：用于防止 CSRF 的随机值。
5. 用户在授权服务器上登录并授权客户端的请求。
6. 授权服务器将用户代理（这个用户代理是授权服务器的用户代理？）重定向回客户端指定的 redirect\_uri，并在 URL 的查询参数中附带上 ​**授权码（code）​**。
7. 客户端应用（通常是其后台服务器）从重定向 URL 中提取授权码。
8. 客户端应用使用这个授权码，加上自己的 client\_id 和 client\_secret（如果是机密客户端），向授权服务器的令牌端点（Token Endpoint）发起 ​**后台请求**，换取 ​**访问令牌（Access Token）​**​ 和 ​**刷新令牌（Refresh Token）​**。
9. 客户端应用使用访问令牌访问资源服务器上的受保护资源。

​**攻击步骤详解：​**​

攻击的目标是让授权服务器将包含受害者授权码的重定向发送到攻击者控制的端点，而不是合法的客户端端点。

1. ​**攻击者注册账户：​**​ 攻击者首先在 ​**合法的客户端应用**​ 上创建一个账户（例如，注册一个使用 Google 登录的第三方笔记应用）。
2. ​**攻击者发起授权请求：​**​ 攻击者登录该客户端应用，并触发 OAuth 登录流程（例如，点击“使用 Google 登录”）。客户端应用会生成一个指向授权服务器（如 Google）的授权请求 URL。这个 URL 包含了客户端的 client\_id 和它期望的 redirect\_uri（比如 https://legitimate-app.com/callback）。
3. ​**攻击者篡改重定向 URI：​**​ 当攻击者的浏览器被重定向到授权服务器时，攻击者 ​**拦截或复制**​ 这个授权请求 URL。然后，攻击者将 URL 中的 redirect\_uri 参数值修改为一个 ​**由攻击者控制的 URI**​（比如 https://attacker.com/steal-code）。
4. ​**攻击者构造恶意链接：​**​ 攻击者利用这个篡改后的 URL 构造一个 ​**恶意链接**。
5. ​**诱骗受害者点击：​**​ 攻击者通过钓鱼邮件、恶意网站、社交工程等手段，诱骗 ​**受害者**​ 点击这个恶意链接。
6. ​**受害者授权：​**​ 受害者点击链接后，其浏览器会被带到授权服务器（如 Google 登录页面）。受害者看到的是一个 ​**完全正常的、代表合法且受信任客户端（第一步的那个应用）的授权请求**。受害者通常会信任这个请求并点击“允许”进行授权。
7. ​**授权码发送给攻击者：​**​ 授权服务器根据篡改后的请求，将包含 ​**受害者授权码**​ 的重定向发送到攻击者控制的 URI (https://attacker.com/steal-code)。此时，​**攻击者的服务器成功接收到了受害者的授权码**。
8. ​**攻击者完成授权流程：​**​ 攻击者现在拥有了受害者的授权码。攻击者立即（在授权码过期前）使用 ​**原始的、合法的** redirect\_uri**​ (**https://legitimate-app.com/callback**) 构造一个请求，将授权码发送给 ​合法的客户端应用**。这模拟了正常流程中授权服务器将用户代理重定向回客户端的过程。
9. ​**客户端兑换令牌并关联攻击者账户：​**​ 合法的客户端应用收到授权码（来自攻击者伪造的重定向请求）。客户端应用像正常流程一样，用自己的 client\_id 和 client\_secret 向授权服务器请求访问令牌。授权服务器验证授权码有效，发放访问令牌（和刷新令牌）。​**关键点：​**​ 客户端应用会将这个访问令牌 ​**关联到最初发起请求的账户**，也就是 ​**攻击者自己的账户**​（在第一步创建的）。
10. ​**攻击者获得权限：​**​ 现在，攻击者通过自己的账户登录该客户端应用时，就拥有了 ​**受害者授权给该客户端的资源访问权限**。攻击者可以访问、修改或删除受害者授权范围内的资源（如受害者的 Google Drive 文件）。

​**攻击成功的关键点：​**​

* 授权服务器在颁发授权码时，只验证了 redirect\_uri 的格式或是否在注册列表内（如果注册了），但没有强制要求必须与后续兑换令牌时使用的 redirect\_uri 一致。
* 攻击者利用受害者完成授权，窃取其授权码。
* 攻击者使用窃取的授权码和 ​\*\*原始的、合法的 redirect\_uri\*\*​ 完成令牌兑换流程，欺骗了客户端应用，让客户端以为这个授权码是攻击者自己正常流程获得的，并将令牌绑定到攻击者账户。
* 受害者授权了客户端访问其资源，但最终获得访问权限的是攻击者的账户。

​**防范措施 (RFC 6749 要求)：​**​

为了防止这种攻击，OAuth 2.0 标准（RFC 6749）明确规定了授权服务器必须采取的措施：

​**重定向 URI 一致性校验：​**​

* + ​**核心要求：​**​ 授权服务器 ​**必须（MUST）​**​ 确保在 ​**兑换授权码获取访问令牌**​ 的请求中提供的 redirect\_uri 参数值，与当初 ​**获取该授权码**​ 时使用的 redirect\_uri 参数值 ​**完全一致**。
  + ​**作用：​**​ 即使攻击者在获取授权码阶段篡改了 redirect\_uri 窃取了授权码，当他在兑换令牌阶段必须提供原始的 redirect\_uri（才能通过客户端验证）时，授权服务器会发现这个 redirect\_uri 与当初颁发授权码时记录的 redirect\_uri（攻击者篡改的那个）不一致，从而拒绝发放令牌。

​**客户端重定向 URI 注册：​**​

* + ​**公开客户端要求：​**​ 授权服务器 ​**必须（MUST）​**​ 要求 ​**公开客户端**​（如单页应用、移动原生应用）​**注册**​ 它们使用的重定向 URI。
  + ​**机密客户端要求：​**​ 授权服务器 ​**应该（SHOULD）​**​ 要求 ​**机密客户端**​（有后端服务器保护 client\_secret 的 Web 应用）也注册其重定向 URI。
  + ​**作用：​**​ 注册建立了客户端允许使用的重定向 URI 白名单。这是后续验证的基础。

​**请求中的重定向 URI 验证：​**​

* + ​**强制验证：​**​ 如果在授权请求中提供了 redirect\_uri 参数（通常是提供的），授权服务器 ​**必须（MUST）​**​ 验证该值是否与为该客户端 ​**注册**​ 的重定向 URI 之一相匹配（例如，精确匹配或基于注册模式匹配）。
  + ​**作用：​**​ 防止攻击者使用任意未注册的 URI（如攻击者自己的域名）作为重定向目标。即使攻击者篡改，也只能改成另一个在客户端注册列表中的 URI（通常很难是攻击者控制的）。

​**总结：​**​

这种攻击利用了 OAuth 授权码流程中 redirect\_uri 参数在授权请求阶段和令牌兑换阶段可能不一致的漏洞。攻击者通过社会工程诱骗受害者，窃取其授权码，并利用合法客户端的流程最终将受害者的资源访问权限绑定到攻击者自己的账户上。

RFC 6749 通过强制要求授权服务器在令牌兑换时校验 redirect\_uri 的一致性、强制/建议客户端注册重定向 URI、并验证请求中的 redirect\_uri 是否已注册，有效地封堵了这个漏洞。任何符合标准的 OAuth 2.0 实现都必须包含这些安全措施。开发者（客户端和授权服务器）在实现 OAuth 时，必须严格遵守这些规定以确保安全性。

在第6步中，授权服务器将用户代理（浏览器）重定向回客户端指定的 redirect\_uri 并附带授权码（Authorization Code），这是整个OAuth 2.0授权码流程的核心交付环节。以下是该步骤的详细技术解析：

### ​****步骤详解：授权服务器的重定向响应****​

当用户同意授权后，授权服务器需要将授权码安全交付给客户端。具体实现如下：

​**构造重定向URL**​  
授权服务器生成一个重定向URL，格式为：  
{redirect\_uri}?code={authorization\_code}&state={client\_state}  
​**关键参数**​：

* 1. ​\*\*code\*\*​：唯一的授权码（通常是一次性的、短时效字符串，如5-10分钟）
  2. ​\*\*state\*\*​：原样返回客户端最初在授权请求中传递的值（用于防御CSRF）

示例：  
https://client-app.com/callback?code=SplxlOBeZQQYbYS6WxSbIA&state=xyz123abc

​**强制执行的安全校验**​  
在返回重定向前，授权服务器**必须**验证：

* 1. ✅ redirect\_uri 已在客户端注册列表中（防止跳转到攻击者域名）
  2. ✅ redirect\_uri 的协议、域名、端口、路径完全匹配注册值（**禁止模糊匹配**）
  3. ✅ 请求中的 state 参数未被篡改（校验来源真实性）

​**发送HTTP重定向响应**​  
授权服务器返回 ​**HTTP 302 Found**​ 状态码，并在 Location 头中指定目标URL：

HTTP/1.1 302 Found

Location: https://client-app.com/callback?code=SplxlOBeZQQYbYS6WxSbIA&state=xyz123abc

* 1. 浏览器会自动根据 Location 头跳转到该URL

### ​****为何使用URL查询参数(Query Parameter)传递？​****​

​**兼容性设计**​

* 1. 适用于所有类型客户端（Web/SPA/移动端），无需复杂跨域机制
  2. 浏览器可自动触发重定向，无需额外JavaScript操作

​**安全性考量**​

* 1. ​**非敏感数据**​：授权码（code）本身并非访问令牌，需配合 client\_secret 兑换（机密客户端）
  2. ​**短时效性**​：授权码有效期通常仅数分钟（即使泄露，攻击窗口小）
  3. ​**绑定关系**​：授权码与特定客户端、重定向URI绑定（服务器端验证关联性）

### ​****攻击者视角下的风险点****​

尽管设计上相对安全，若实现不当仍可能被利用：

| **​风险场景​** | **​防护措施​** |
| --- | --- |
| 未校验 redirect\_uri 注册 | 强制精确匹配注册URI |
| 未绑定 code 与 redirect\_uri | 令牌兑换时验证一致性（RFC 6749 §4.1.3） |
| 浏览器历史记录泄露授权码 | 使用PKCE增加 code\_challenge 验证（OAuth 2.1强制要求） |
| 第三方脚本窃取URL中的code | 客户端应限制回调页面的JS权限 |

### ​****客户端如何安全接收授权码？​****​

​**Web服务器端客户端（机密客户端）​**​

sequenceDiagram

participant Browser

participant ClientBackend

Browser->>ClientBackend: GET /callback?code=xxx&state=xyz

ClientBackend->>ClientBackend: 校验state参数

ClientBackend->>AuthServer: POST /token 用code兑换令牌

​**单页应用（SPA）等公开客户端**​

* 1. 前端JS解析URL参数后立即清除
  2. 通过隐藏的iframe或Web Worker将code发送至后端
  3. ​**必须启用PKCE**​（RFC 7636）防御授权码拦截攻击

### ​****关键安全要求总结****​

​**强制精确匹配**​  
redirect\_uri 必须与注册值逐字符一致（包括末尾斜杠和大小写）

​**授权码绑定三重信息**​  
服务器端需关联：授权码 + 客户端ID + 重定向URI

​**禁止前端存储**​  
客户端获得code后必须立即兑换令牌，不可持久化

​**必须启用PKCE**​（OAuth 2.1规范）  
通过 code\_verifier 确保请求者与初始授权者为同一实体

### ​****为何不是通过PostMessage或其它方式？​****​

* 传统OAuth 2.0设计需兼容所有客户端类型（包括无JS环境）
* 现代应用可使用 ​**OAuth 2.0 for Browser-Based Apps**​ 草案(Browser RFC)中的替代方案，例如：

// 授权服务器返回HTML包含JS片段

window.opener.postMessage({type: 'oauth', code: 'xxx'}, 'https://client-app.com');

* 但该方案仍需依赖PKCE，且与传统Web流程兼容性存在挑战。

通过以上机制，授权服务器在保证用户体验（自动重定向）的同时，尽可能安全地将授权码传递到可信的客户端环境。​**核心防御思想在于：授权码本身不具备直接访问能力，且严格绑定客户端的认证上下文**。

您的问题非常好，这涉及到OAuth协议中的关键术语界定。在**第6步的重定向操作中，用户代理（User Agent）特指资源所有者（用户）的浏览器，而非授权服务器的用户代理**。以下是详细解析：

### ​****用户代理的身份界定****​

| **​参与方​** | **​用户代理归属​** | **​说明​** |
| --- | --- | --- |
| 资源所有者（用户） | ✅ ​**用户拥有**​ | 用户的浏览器（如Chrome/Safari）直接处理重定向操作 |
| 授权服务器 | ❌ ​**不拥有**​ | 授权服务器仅通过HTTP响应指令**触发**用户浏览器的跳转行为 |
| 客户端应用 | ❌ ​**不拥有**​ | 客户端也无法直接控制浏览器，只能等待重定向回调 |

### ​****关键流程技术拆解****​

​**授权服务器的动作（无用户代理）​**​  
授权服务器**不主动操作任何用户代理**，它只做一件事：  
→ ​**向用户的浏览器返回HTTP 302重定向响应**​

HTTP/1.1 302 Found

Location: https://client-app.com/callback?code=XYZ123&state=ABC789

​**用户浏览器的动作（真正的用户代理）​**​  
用户的浏览器收到302响应后**自主执行**​：  
→ ​**自动跳转到**Location**头指定的URL（携带code和state）​**​  
→ 此时控制权完全在**用户自己的浏览器**中

​**通信路径可视化**​

sequenceDiagram

participant 用户浏览器 as 用户浏览器 (User Agent)

participant 授权服务器 as 授权服务器

participant 客户端 as 客户端回调端点

用户浏览器->>授权服务器: POST 用户授权确认（无代理）

授权服务器-->>用户浏览器: HTTP 302 + Location: {redirect\_uri}?code=XX&state=YY

注意 right of 授权服务器: \*\*授权服务器不拥有用户代理\*\*<br/>仅返回指令

用户浏览器->>客户端: 自动跳转到Location（携带参数）

注意 right of 用户浏览器: \*\*浏览器作为用户代理执行重定向\*\*

### ​****为什么说用户代理属于用户？​****​

​**OAuth协议定义**​（RFC 6749 §1）：

"User Agent: The client component that communicates with the resource owner, typically a web browser."  
​**明确指定用户代理是资源所有者（用户）使用的工具**。

​**核心特征**​：

* 1. 代理用户的HTTP请求行为
  2. 存储用户的Cookies/会话状态
  3. 执行用户触发的页面跳转（此处跳转由授权服务器触发，但由用户浏览器执行）

### ​****常见误解澄清****​

| **误解说法** | **真相** |
| --- | --- |
| “授权服务器用它的用户代理发送code” | 授权服务器不拥有用户代理，只返回HTTP指令 |
| “重定向是服务端之间的直接通信” | 重定向必须经用户浏览器中转（HTTP是无状态协议） |
| “客户端能主动抓取code” | 客户端被动等待**用户浏览器**携带code访问其注册的回调端点 |

### ​****关键安全意义****​

​**安全边界**​：  
code的传递始终通过**用户浏览器**完成，因此：

* 1. 若redirect\_uri被篡改 → code流向攻击者控制的地址（原攻击原理）
  2. 若用户浏览器被恶意扩展监控 → code可能被窃取  
     → 这解释了为何需要**PKCE**​（Proof Key for Code Exchange）等额外防护

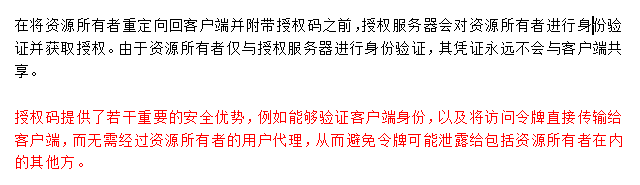
​**隐私保护**​：  
授权服务器不会（也无法）直接操作用户浏览器，只能通过标准HTTP机制（302重定向）请求浏览器跳转。

### 总结

在OAuth流程中：  
✅ ​**用户代理特指用户的浏览器**​  
✅ ​**授权服务器仅通过HTTP响应指令触发浏览器行为**​  
✅ ​**重定向操作的实际执行者是用户设备上的浏览器**​

这种设计保障了协议的中立性（不依赖特定服务端能力），但也要求开发者严格遵循redirect\_uri验证规范，防止恶意利用用户浏览器作为攻击跳板。

1.3.1



Client\_id与Code绑定时，需要对client\_id验证serect值



