微服务系统安全

1. 对外API 安全设计

①身份认证

①-1. API KEY

API Key 就是经过用户身份认证后，服务端给客户端分配一个 API Key用于后续调用时验证身份

一般的处理流程

Client 端

Server 端

Client 端向服务端注册，服务端给客户端发送响应的api\_key以及security\_key，注意保存不要泄露，然后客户端根据 api\_key、secrity\_key、timestrap、rest\_uri 采用 hmacsha256 算法得到一个 hash 值 sign，构造途中的 url 发送给服务端。

服务端收到该请求后，首先验证 api\_key 是否存在，存在则获取该 api\_key 的 security\_key，接着验证 timestrap 是否超过时间限制，可依据系统成而定，这样就防止了部分重放攻击，途中的 rest\_api 是从 url 获取的为 /rest/v1/interface/eth0，最后计算 sign 值，之后和 url 中的 sign 值做校验。这样的设计就防止了数据被篡改。通过这种 API Key 的设计方式加了时间戳防止部分重放，加了校验，防止数据被篡改，同时避免了传输用户名和密码，当然也会有一定的开销。

①-2. Oauth2

OAuth 协议适用于为外部应用授权访问本站资源的情况。

OAuth 2.0的运行流程如下图

摘自RFC 6749。

名词解释

（1）Third-party application：第三方应用程序，本文中又称"客户端"（client）。

（2）HTTP service：HTTP服务提供商，本文中简称"服务提供商"。

（3）Resource Owner：资源所有者，本文中又称"用户"（user）。

（4）User Agent：用户代理，本文中就是指浏览器。

（5）Authorization server：认证服务器，即服务提供商专门用来处理认证的服务器。

（6）Resource server：资源服务器，即服务提供商存放用户生成的资源的服务器。它与认证服务器，可以是同一台服务器，也可以是不同的服务器。

上述流程的（B）【用户同意给予客户端授权】有四种模式

* 授权码模式（authorization code）

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。

（A）用户访问客户端，后者将前者导向认证服务器。

客户端申请认证的URI，包含以下参数：

response\_type：表示授权类型，必选项，此处的值固定为"code"

client\_id：表示客户端的ID，必选项

redirect\_uri：表示重定向URI，可选项

scope：表示申请的权限范围，可选项

state：表示客户端的当前状态，可以指定任意值，认证服务器会原封不动地返回这个值。

（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。

服务器回应客户端的URI，包含以下参数：

code：表示授权码，必选项。该码的有效期应该很短，通常设为10分钟，客户端只能使用该码一次，否则会被授权服务器拒绝。该码与客户端ID和重定向URI，是一一对应关系。

state：如果客户端的请求中包含这个参数，认证服务器的回应也必须一模一样包含这个参数。

（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。

客户端向认证服务器申请令牌的HTTP请求，包含以下参数：

grant\_type：表示使用的授权模式，必选项，此处的值固定为"authorization\_code"。

code：表示上一步获得的授权码，必选项。

redirect\_uri：表示重定向URI，必选项，且必须与A步骤中的该参数值保持一致。

client\_id：表示客户端ID，必选项。

（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）。

认证服务器发送的HTTP回复，包含以下参数：

access\_token：表示访问令牌，必选项。

token\_type：表示令牌类型，该值大小写不敏感，必选项，可以是bearer类型或mac类型。

expires\_in：表示过期时间，单位为秒。如果省略该参数，必须其他方式设置过期时间。

refresh\_token：表示更新令牌，用来获取下一次的访问令牌，可选项。

scope：表示权限范围，如果与客户端申请的范围一致，此项可省略。

* 隐式模式（implicit）

不通过第三方应用程序的服务器，直接在浏览器中向认证服务器申请令牌，跳过了”授权码”这个步骤，因此得名。所有步骤在浏览器中完成，令牌对访问者是可见的，且客户端不需要认证。



（A）客户端将用户导向认证服务器。

客户端发出的HTTP请求，包含以下参数：

response\_type：表示授权类型，此处的值固定为"token"，必选项。

client\_id：表示客户端的ID，必选项。

redirect\_uri：表示重定向的URI，可选项。

scope：表示权限范围，可选项。

state：表示客户端的当前状态，可以指定任意值，认证服务器会原封不动地返回这个值。

（B）用户决定是否给于客户端授权。

（C）假设用户给予授权，认证服务器将用户导向客户端指定的”重定向URI”，并在URI片断(即URI中#号之后的内容)部分包含了访问令牌。

认证服务器回应客户端的URI，包含以下参数：

access\_token：表示访问令牌，必选项。

token\_type：表示令牌类型，该值大小写不敏感，必选项。

expires\_in：表示过期时间，单位为秒。如果省略该参数，必须其他方式设置过期时间。

scope：表示权限范围，如果与客户端申请的范围一致，此项可省略。

state：如果客户端的请求中包含这个参数，认证服务器的回应也必须一模一样包含这个参数。

（D）浏览器向资源服务器发出请求，其中不包括上一步收到的URI片断。

（E）资源服务器返回一个网页，其中包含的代码可以获取URI片断中的令牌。

（F）浏览器执行上一步获得的脚本，提取出令牌。

（G）浏览器将令牌发给客户端。

* 密码模式（resource owner password credentials）

密码模式就是将密码托管给第三方App，但是必须要保证第三方App高度可信。

* 客户端模式（client credentials）

这种模式不需要终端用户的参与，只是Client和Server端的交互。通常只用于Client状态的获取。

OAuth2的漏洞

对于授权码和access\_token的篡改，在OAuth1中是反复的对Code和Token进行签名，来保证Token不会被篡改，但是OAuth2中却没有，因为OAuth2是基于Https的，所以如果没有Https的支持OAuth2可能还不如OAuth1.

对于redirect\_uri的校验，OAuth1中没有提到redirect\_uri的校验，那么OAuth2中要求进行redirect\_uri的校验。但是如果校验规则过松，也会导致跳转的安全问题。 例如：校验的时候只校验根域名，或者二级域名，但是第三方App对自己的域名保护的不好，导致二级域名被hack那么此时授权码和Token会被窃取。 校验规则不严谨，例如www.baidu.com 但是redirect\_uri为：www.a.com.\www.baidu.com，这样授权就被a.com钓走了。

对于CSRF攻击（跨站请求伪造）：由于这个授权过程服务器和Client和用户之间有几次交互，但是在得到授权码的时候需要一次回跳，但是这次回跳是可以被阻塞的。

攻击者使用自己的账户申请第三方授权登陆

授权后服务端返回授权码，但是此时组织授权回跳，此时Client并没有接到授权码，也就是阻断了授权流程

攻击者将此跳转链接发给一个正在处于在Client登陆状态的账户

诱骗正常用户点击，那么此时攻击者第三方账户和被攻击账户进行绑定（相当于账户绑定了第三方的账户）

攻击者再次进行第三方授权登陆。这样就劫持了诱骗的账户。转账？删好友？等等就所以搞了。  
解决办法：  
在进行授权码申请或者是Token申请的时候带上state参数，服务器返回请求时要求携带state参数，在Client处理授权的时候校验此参数。参数可以是当前账户的SessionId，或Cookie的签名串。在Client申请accessToken会验证相关state和当前用户的关系，这样就防止了篡改。

OAuth的校验流程为什么这么复杂，直接授权之后redirect回accessToken不就结了吗？为什么还要返回auth\_code之后请求accessToken？  
首先，redirect是不安全的，随时可以暂停回调而拿到accessToken，拿到了accessToken也就意味着拿到了授权，但是auth\_code是和client相对应的，那么即使拿到了auth\_code还需要再次申请accessToken，申请accessToken时需要校验Client和state。同时协议设计的原则就是只有Client能拿到accessToken而用户是拿不到的。

①-3. JWT

JWT 是 JSON Web Token，用于发送可通过数字签名和认证的东西，它包含一个紧凑的、URL 安全的 JSON 对象，服务端可通过解析该值来验证是否有操作权限、是否过期等安全性检查。由于其紧凑的特点，可放在 url 中或者 HTTP Authorization 头中，具体算法如下图：

②:授权

身份认证后就是授权，根据不同身份，授予不同的访问权限。

垂直权限：如 admin 用户、普通用户、auditor 用户

roles : [

ADMIN: {

permit: [‘/^((\/system\/(clouds|device)$/‘]

deny: ['/^(\/system\/audit)$/']

},

AUDIT: {

permit: ['/^(\/system\/audit)$/']

deny: ['/^((\/system\/(clouds|device).\*)$/']

}

]

平行权限：如用户 A 获取用户 B 的身份信息或者更改其他用户信息，对于这些敏感数据接口都需要加上对用户的判断，这一步一般都在具体的逻辑中实现

③URL 过滤，参数过滤

④重要功能加密传输，采用 SSL 保证传输安全

⑤请求速率限制

⑥错误处理

对于非法的、导致系统出错的等各种请求进行记录，还有一些重要操作，比如登录、注册等都通过日志接口输出展示。

2.微服务之间的认证

①微服务间的相互访问不进行认证和鉴权  
微服务提供的数据敏感程度不是很高，则依托网络安全措施，来保障微服务间的通讯安全

②服务间的访问采用颁发访问凭证进行安全控制

微服务在使用的维度上，我们定义服务的调用方和服务的提供方。服务的提供方对调用方颁发访问凭证，提供方对访问严格控制；没有访问凭证的访问，拒绝访问；根据不同的访问凭证类型，安全控制不同的访问类型。

③服务间的访问采用Service Account进行安全控制

Istio-Auth提供强大的服务间认证和终端用户认证，使用交互TLS，内置身份和证书管理。可以升级服务网格中的未加密流量，并为运维人员提供基于服务身份而不是网络控制来执行策略的能力。Istio的未来版本将增加细粒度的访问控制和审计，以使用各种访问控制机制（包括基于属性和角色的访问控制以及授权钩子）来控制和监视访问您的服务，API或资源的人员。

心得：

微服务架构的安全都是围绕保护API来展开的。系统安全涉及到很多方面需要考虑。

参考

<https://www.infoq.cn/article/EyJ8IcgMP3J3PM9uF4y4>

<https://tools.ietf.org/html/rfc6749>

<https://www.jianshu.com/p/d74ce6ca0c33>

<https://www.ruanyifeng.com/blog/2014/05/oauth_2_0.html>

<https://toutiao.io/posts/n15ny2/preview>