开放API认证方案v2.2

常用的开放API认证方案有HTTP协议簇中的HTTP Authentication [RFC2617]方案，还有目前使用比较广泛的OAuth 2.0[RFC6749]方案。

HTTP Authentication主要由浏览器和HTTP客户端支持，支持Basic [RFC7617]和Digest [RFC7616]两种认证，一般应用使用的比较少。
OAuth 2.0主要用于跨系统的单点登录认证和鉴权，也可以支持基于客户端凭证的鉴权，扩展协议可以支持JWT、证书或者联邦鉴权方案。
也有些开放API服务使用最简单的API Key/Token鉴权，由于Key和Token明文传输比较容易泄露，这种方案一般用在安全性要求不高的场景。

云平台上存在着SSL卸载和较多的代理网关的场景，基于明文密码或者access token的方式在存在比较明显的安全隐患，用户访问凭证比较容易泄露，
数据比较容易被篡改。所以当前各大云平台用的最多的还是AK(Access Key ID)/SK(Secret Access Key)签名认证方案，除了能实现比较安全的客户端认证方案，
还可以实现端到端的消息完整性签名，同时也不会产生额外的认证请求，总体方案也比较简单。AK/SK认证可以认为是云平台开放API事实上的标准。
实际上，Joyent公司早在2013年就提出了第一版HTTP签名认证的草案，后面Oracle和Amazon分别在2015和2019年对RFC草案进行修订和完善工作。
最新的HTTP消息签名、认证的RFC草案：https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-message-signatures-09

我们的开放API认证也采用AK/SK的认证方案。

1 AK/SK认证基本原理

用户在开放API平台上生成一组AK/SK，AK用于标识客户端，SK用于作为认证凭证和签名密钥。AK和SK一般是随机生成、编码之后的字符串，
其中SK的长度需要足够长，防止被暴力破解。然后将AK和SK分发给客户端程序使用。

客户端程序发送API请求的时候，需要使用SK对HTTP请求的方法、参数、头部、系统时间以及Body进行HMAC-SHA256签名，
然后将签名数据以及AK添加到HTTP头部的Authorization字段中，随着API请求发送到服务端。

服务端接收到API请求之后，通过AK查找到对应的SK，然后按照同样的方式对请求信息进行签名。如果计算出来的签名和客户端发送的签名一致，
则校验通过。否则，要么是客户端使用了错误的AK/SK，要么存在请求被篡改的情况。

除了校验签名，一般还会对HTTP请求时间进行校验。客户端发送请求时，要求在请求头中添加了请求时间。服务端接收到请求后，
会从HTTP头部中提取请求时间，如果请求时间和当前系统时间偏差太大，则认为签名已经失效。可以一定程度上缓解回放攻击。
但是由于依赖了客户端系统时间，需要保证客户端和服务器系统时间偏差不能太大，比如时间偏差不能超过10分钟。

2 AK/SK签名计算过程

1）构造HTTPS客户端请求，用于计算签名

可以直接使用各个语言的HTTP客户端库，先创建好request对象，方便后续计算签名。签名时可以直接从request对象中获取需要的数据。

这里以如下请求为例：

```
#Request Method
POST

#Request URL
https://192.168.80.80/auth/v5/token?query1=val1&query2=val2

#Request Body
{
"rand": string,
"domain": string,
"userName": string,
"clientName": string
}
```

2）基于HTTP请求，创建规范的待签名请求字符串CanonicalReq

CanonicalReq构造方法：

```
CanonicalReq =
 HTTPRequestMethod + '\n' +
 CanonicalURI + '\n' +
 CanonicalQueryString + '\n' +
 CanonicalHeaders + '\n' +
 SignedHeaders + '\n' +
 HashedPayload
```

CanonicalReq示例：

```
POST
/auth/v5/token
query1=val1&query2=val2
content-type:application/json;charset=utf-8
host:192.168.80.80
sign-date:20191115T033655Z
content-type;host;sign-date
1981d5a1848b70d0b5a64e221f627c9718deea8ff1d23d211b149f0ad0ea0365
```

3）对CanonicalReq计算哈希得到HashedCanonicalRequest 。使用SHA-256算法

HashedCanonicalRequest 计算方法：

```
HashedCanonicalRequest = Lowercase(HexEncode(Hash.SHA256
(CanonicalRequest)))
```

HashedCanonicalRequest 示例：

```
a74b20df5bed2a6ef85b45a1fb2f5df979781b31afe37d61f47e6e1c377d8aba
```

4）构造待签名字符串StringToSign

StringToSign构造方法：

```
StringToSign =
    Algorithm + \n +
    RequestDateTime + \n +
    HashedCanonicalRequest
```

Algorithm为签名使用的算法，我们使用的是HMAC-SHA256。

RequestDateTime为请求时间，需要按照ISO8601时间格式进行格式化，具体格式为：YYYYMMDD'T'HHMMSS'Z'。

StringToSign示例：

```
HMAC-SHA256
20191115T033655Z
a74b20df5bed2a6ef85b45a1fb2f5df979781b31afe37d61f47e6e1c377d8aba
```

5）计算签名

`signature`

```
signature = HexEncode(HMAC(Access Secret Key, string to sign))
```

`signature`

```
3eac3a0c35e496761838b031cd31c65a8507fee0806854c9f335e0ee92bdca94
```

6）签名信息添加到头部

签名信息格式：

```
Authorization: algorithm=Algorithm,Access=Access key,
SignedHeaders=SignedHeaders,Signature=signature
```

签名信息示例：

```
Authorization: algorithm=HMAC-SHA256,
Access=BD74E58C3141FCA7B80ED3513EBB1E22,SignedHeaders=content-type;host;
sign-date,
Signature=3eac3a0c35e496761838b031cd31c65a8507fee0806854c9f335e0ee92bdca
94
```

3 签名涉及的数据规范化

1）HTTPRequestMethod

HTTP方法不需要特别处理，标准的HTTP方法都是大写的，但是HTTP协议里面对method的定义是大小写敏感的，所以对于HTTP方法不需要特别处理。

2）CanonicalURI

根据RFC [3986]规范对URI路径进行规范化处理，移除冗余和相对路径部分，路径中每个部分必须为URI编码。如果URI路径不以“/”结尾，则在尾部添加“/”。

3）CanonicalQueryString

查询参数的规范化需要按照如下步骤进行处理：

a. 把所有查询参数按参数名称进行排序，如果名称相同的，需要按照value按升序排序。

b. 根据RFC [3986]规范对查询参数和value进行编码。

c. 按顺序构造标准的查询参数，先追加参数名称，再追加“=”，再追加参数value。如果参数没有value字段，则value为空。

d. 除了最后一个value外，其他的value之后需要追加“&”。

4）CanonicalHeaders

协议头名称需要转成小写，然后按照协议头名称按升序排序，然后按如下格式拼装成规范化的协议头：

```
CanonicalHeaders =
CanonicalHeadersEntry0 + CanonicalHeadersEntry1 + ... +
CanonicalHeadersEntryN
CanonicalHeadersEntry =
Lowercase(HeaderName) + ':' + Trimall(HeaderValue) + '\n'
```

签名必须包含的协议头包括：content-type、host和sign-date。更多的头部字段可以由客户端自定义，服务端不应该假定要签名的头部的类型。

5）SignedHeaders

协议头名称需要转成小写，然后按照协议头名称按升序排序，然后按如下格式拼装成签名头：

```
SignedHeaders = Lowercase(HeaderName0) + ';' + Lowercase(HeaderName1) +
";" + ... + Lowercase(HeaderNameN)
```

6）HashedPayload

负载数据哈希和HashedCanonicalRequest一样使用SHA-256算法，具体计算方式如下：

```
HashedPayload = Lowercase(HexEncode(Hash.SHA256(RequestPayload)))
```

4 AK/SK管理规范

由于SK属于对称秘钥，一旦泄露出去，会存在被冒用的风险。所以SK存储的时候需要进行加密，分发的时候也需要避免泄露出去。
对于负责签名验证的服务端来说，除了需要对SK进行加密存储外，可以考虑使用独立的签名验证服务减少攻击面。

生成AK/SK时，需要使用安全的随机函数，对于AK可以生成20个字节以上的alphanumeric字符。SK长度需要保证128位以上，
SK可以使用base64或者16进制进行编码。