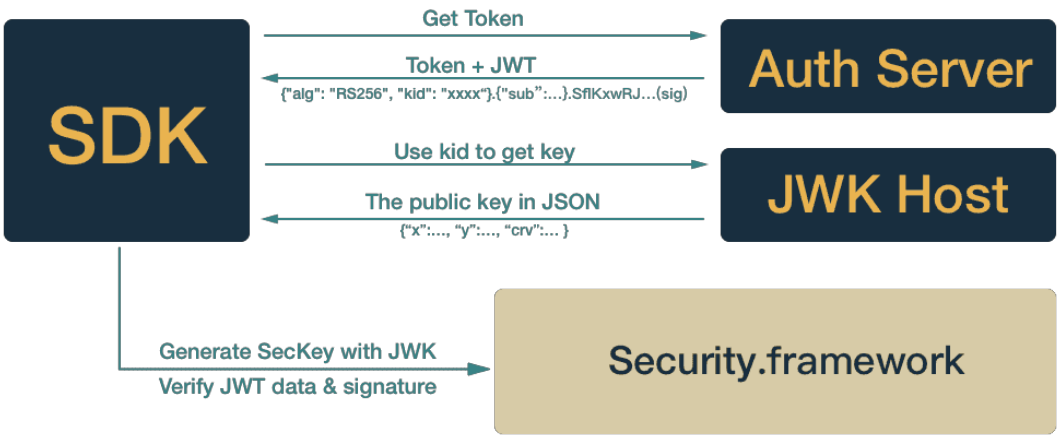
基础 - 什么是 JWT 以及 JOSE

概述

事情的缘由很简单，工作上在做 LINE SDK 的开发，在拿 token 的时候有一步额外的验证：从 Server 会发回一个 JWT (JSON Web Token)，客户端需要对这个 JWT 进行签名和内容的验证，以确保信息没有被人篡改。Server 在签名中使用的算法类型会在 JWT 中写明，验证签名所需要的公钥 ID 也可以在 JWT 中找到。这个公钥是以 JWK (JSON Web Key) 的形式公开，客户端拿到 JWK 后即可在本地对收到的 JWT 进行验证。用一张图的话，大概是这样：



步骤

> 如果你现在对下面说步骤不理解的话 (这挺正常的，毕竟这篇文章都还没正式开始 😂)，可以先跳过这部分，等我们有一些基础知识以后再回头看看就好。如果你很清楚这些步骤的话，那真是好棒棒，你应该能无压力阅读该系列剩余部分内容了。

LINE SDK 里使用 JWT 验证用户的逻辑如下：

1. 向登录服务器请求 access token，登录服务器返回 access token，同时返回一个 JWT。
2. JWT 中包含应该使用的算法和密钥的 ID。通过密钥 ID，去找预先定义好的 Host 拿到 JWK 形式的该 ID 的密钥。
3. 将 1 的 JWT 和 2 的密钥转换为 Security.framework 接受的形式，进行签名验证。

这个过程想法很简单，但会涉及到一系列比较基础的密码学知识和标准的阅读，难度不大，但是枯燥乏味。另外，由于 iOS 并没有直接将 JWK 转换为 native 的 `SecKey` 的方式，自己也没有任何密码学的基础，所以在处理密钥转换上也花了一些工夫。为了后来者能比较顺利地处理相关内容（包括 JWT 解析验证，JWK 特别是 RSA 和 EC 算法的密钥转换等），也为了过一段时间自己还能有地方回忆这些内容，所以将一些关键的理论知识和步骤记录下来。

系列文章的内容

整个系列会比较长，为了阅读压力小一些，我会分成三个部分：

1. 基础 - 什么是 JWT 以及 JOSE（本文）
2. 理论 - JOSE 中的签名和验证流程
3. 实践 - 如何使用 Security.framework 处理 JOSE 中的验证

全部读完的话应该能对网络相关的密码学有一个肤浅的了解，特别是常见的签名算法和密钥种类，编码规则，怎么处理拿到的密钥，怎么做签名验证等等。如果你在工作中有相关需求，但不知道如何下手的话，可以仔细阅读整个系列，并参看开源的 LINE SDK Swift 的相关实现，甚至直接 copy 部分代码（如果可以的话，也请顺便点一下 star）。如果你只是感兴趣想要简单了解的话，可以只看 JOSE 和 JWT 的基础概念和理论流程部分的内容，作为知识面的扩展，等以后有实际需要了再回头看实践部分的内容。

在文章结尾，我还列举了一些常见的问题，包括笔者自己在学习时的思考和最后的选择。如果您有什么见解，也欢迎发表在评论里，我会继续总结和补充。

> 声明：笔者自身对密码学也是初学，而本文介绍的密码学知识也都是自己的一些理解，同时尽量不涉及过于原理性的内容，一切以普通工程师实用为目标原则。其中可以想象在很多地方会有理解的错误，还请多包涵。如您发现问题，也往不吝赐教指正，感激不尽。

JWT 以及 JOSE

什么是 JWT

估计大部分 Swift 的开发者对 JWT 会比较陌生，所以先简单介绍一下它是什么，以及可以用来做什么。JWT (JSON Web Token) 是一个编码后的字符串，比如：

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM
5MDIyfQ.
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

一个典型的 JWT 由三部分组成，通过点号 `.` 进行分割。每个部分都是经过 **Base64Url** 编码的字符串。第一部分 (Header) 和第二部分 (Payload) 在解码后应该是有效的 JSON，最后一部分 (签名) 是通过一定算法作用在前两部分上所得到的签名数据。接收方可以通过这个签名数据来验证 token 的 Header 及 Payload 部分的数据是否可信。

> 为了视觉上看起来轻松一些，在上面的 JWT 例子中每个点号后加入了换行。实际的 JWT 中不应该存在任何换行的情况。

> 严格来说，JWT 有两种实现，分别是 JWS (JSON Web Signature) 和 JWE (JSON Web Encryption)。由于 JWS 的应用更为广泛，所以一般说起 JWT 大家默认会认为是 JWS。JWS 的 Payload 是 Base64Url 的明文，而 JWE 的数据则是经过加密的。相对地，相比于 JWS 的三个部分，JWE 有五个部分组成。本文中提到 JWT 的时候，所指的都是用于签名认证的 JWS 实现。

> 关于 Base64Url 编码和处理，在本文后面部分会再提到。

Header

Header 包含了 JWT 的一些元信息。我们可以尝试将上面的 `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9` 这个 Header 解码，得到：

```
{"alg":"HS256","typ":"JWT"}
```

> 关于在数据的不同格式之间互相转换 (明文，Base64，Hex Bytes 等)，我推荐这个非常不错的 web app。

在 JWT Header 中，”alg” 是必须指定的值，它表示这个 JWT 的签名方式。上例中 JWT 使用的是 `HS256` 进行签名，也就是使用 SHA-256 作为摘要算法的 HMAC。常见的选择还有 `RS256`， `ES256` 等等。总结一下：

- `HSXXX` 或者说 HMAC：一种对称算法 (symmetric algorithm)，也就是加密密钥和解密密钥是同一个。类似于我们创建 zip 文件时设定的密码，验证方需要知道和签名方同样的密钥，才能得到正确的验证结果。
- `RSXXX`：使用 RSA 进行签名。RSA 是一种基于极大整数做因数分解的非对称算法 (asymmetric algorithm)。相比于对称算法的 HMAC 只有一对密钥，RSA 使用成对的公钥 (public key) 和私钥 (private key) 来进行签名和验证。大多数 HTTPS 中验证证书和加密传输数据使用的是 RSA 算法。
- `ESXXX`：使用 椭圆曲线数字签名算法 (ECDSA) 进行签名。和 RSA 类似，它也是一种非对称算法。不过它是基于椭圆曲线的。ECDSA 最著名的使用场景是比特币的数字签名。
- `PSXXX`：和 `RSXXX` 类似使用 RSA 算法，但是使用 PSS 作为 padding 进行签名。作为对比，`RSXXX` 中使用的是 PKCS1-v1_5 的 padding。

> 如果你对这些介绍一头雾水，也不必担心。关于各个算法的一些更细节的内容，会在后面实践部分再详细说明。现在，你只需要知道 Header 中 “alg” key 为我们指明了签名所使用的签名算法和散列算法。我们之后需要依据这里的指示来验证签名。

除了 “alg” 外，在 Header 中发行方还可以放入其他有帮助的内容。JWS 的标准定义了一些预留的 Header key。在本文中，除了 “alg” 以外，我们还会用到 “kid”，它用来表示在验证时所需要的，从 JWK Host 中获取的公钥的 key ID。现在我们先集中于 JWT 的构造，之后在 JWK 的部分我们再对它的使用进行介绍。

Payload

Payload 是想要进行交换的实际有意义的数据部分。上面例子解码后的 Payload 部分是：

```
{"sub":"1234567890","name":"John Doe","iat":1516239022}
```

和 Header 类似，payload 中也有一些预先定义和保留的 key，我们称它们为 claim。常见的预定义的 key 包括有：

- “iss” (Issuer)：JWT 的签发者名字，一般是公司名或者项目名
- “sub” (Subject)：JWT 的主题
- “exp” (Expiration Time)：过期时间，在这个时间之后应当视为无效
- “iat” (Issued At)：发行时间，在这个时间之前应当视为无效

当然，你还可以在 Payload 里添加任何你想要传递的信息。

我们在验证签名后，就可以检查 Payload 里的各个条目是否有效：比如发行者名字是否正确，这个 JWT 是否在有效期内等等。因为一旦签名检查通过，我们就可以保证 Payload 的东西是可靠的，所以这很适合用来进行消息验证。

> 注意，在 JWS 里，Header 和 Payload 是 Base64Url 编码的**明文**，所以你不应该用 JWS 来传输任何敏感信息。如果你需要加密，应该选择 JWE。

Signature

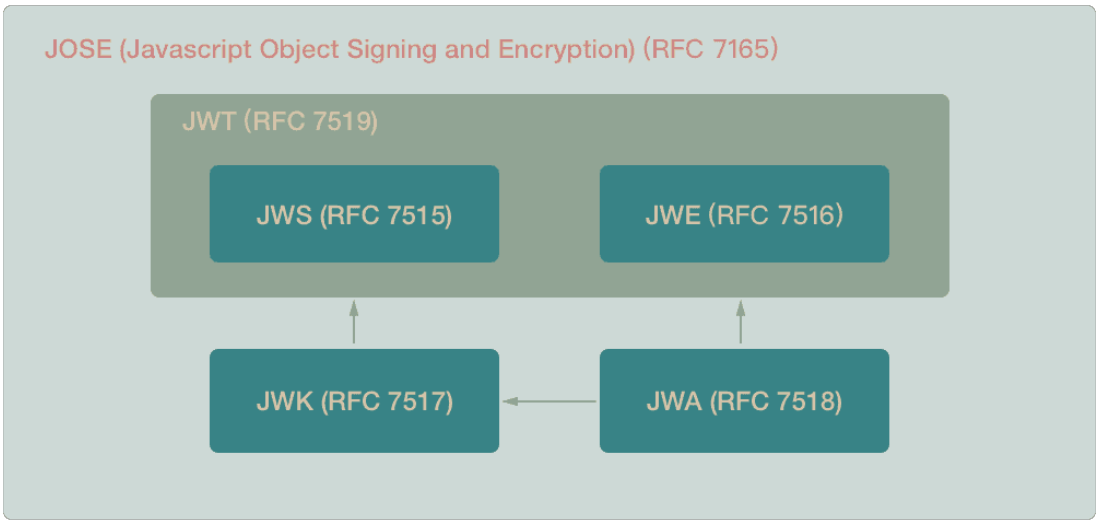
一个 JWT 的最后一部分是签名。首先对 Header 和 Payload 的原文进行 Base64Url 编码，然后用 `.` 将它们连接起来，最后扔给签名散列算法进行签名，把签名得到的数据再 Base64Url 编码，就能得到这个签名了。写成伪代码的话，是这样的：

```
// 比如使用 RS256 签名：
let 签名数据: Data = RS256签名算法(Base64Url(string:
Header).Base64Url(string: Payload), 私钥)
let 签名: String = Base64Url(data: 签名数据)
```

最后，把编码后的 Header，Payload 和 Signature 都用 `.` 连在一起，就是我们收发的 JWT 了。

什么是 JOSE

JWT 其实是 JOSE 这个更大的概念中的一个组成部分。JOSE (Javascript Object Signing and Encryption) 定义了一系列标准，用来规范在网络传输中使用 JSON 的方式。我们在上面介绍过了JWS 和 JWE，在这一系列概念中还有两个比较重要，而且相互关联的概念：JWK 和 JWA。它们一起组成了整个 JOSE 体系。



JWK

不管签名验证还是加密解密，都离不开密钥。JWK (JSON Web Key) 解决的是如何使用 JSON 来表示一个密钥这件事。

RSA 的公钥由模数 (modulus) 和指数 (exponent) 组成，一个典型的代表 RSA 公钥的 JWK 如下：

```
{
  "alg": "RS256",
  "n": "ryQICCl6NZ5gDKrnSztO3Hy8PEUcuyvg_ikC-
VcIo2SFFSf18a3IMYldIugqqqZCs4_4uVW3sbdLs_6PfgdX7O9D22ZiFWHPYA2k2N74
4MNiCD1UE-tJyllUhSblK48bn-v1oZHCM0nYQ2NqUkvSj-
hwUU3RiWl7x3D2s9wSdNt7XUtW05a_FXehsPSiJfKvHJJnGOX0BgTvkLnkAOTdOrUZ_
wK69Dzu4IvrN4vs9Nes8vbwPa_ddZEzGR0cQMt0JBkhk9kU_qwqUseP1QRJ5I1jR4g8
aYPL_ke9K35PxZWuDp3U0UPAZ3PjFAh-5T-fc7gzCs9dPzSHloruU-glFQ",
  "use": "sig",
  "kid": "b863b534069bfc0207197bcf831320d1cdc2cee2",
  "e": "AQAB",
  "kty": "RSA"
}
```

模数 `n` 和指数 `e` 构成了密钥最关键的数据部分，这两部分都是 Base64Url 编码的大数字。

> 关于 RSA 的原理，不在本文范围内，你可以在其他很多地方找到相关信息。
>
> 如果你接触过几个 RSA 密钥，可能会发现“e”的值基本都是“AQAB”。这并不是巧合，这是数字 65537 (0x 01 00 01) 的 Base64Url 表示。选择 AQAB 作为指数已经是业界标准，它同时兼顾了运算效率和安全性能。同样，这部分内容也超出了本文范畴。

类似地，一个典型的 ECDSA 的 JWK 内容如下：

```
{
  "kty":"EC",
  "alg":"ES256",
  "use":"sig",
  "kid":"3829b108279b26bcfcc8971e348d116",
  "crv":"P-256",
  "x":"EVs_o5-uQbTjL3chynL4wXgUg2R9q9UU8I5mEovUf84",
  "y":"AJBnuQ4EiMnCqfMPWiZqB4QdbAd0E7oH50VpuZ1P087G"
}
```

决定一个 ECDSA 公钥的参数有三个：“crv”定义使用的密钥所使用的加密曲线，一般可能值为“P-256”，”P-384”和“P-521”。”x”和“y”是选取的椭圆曲线点的座标值，根据曲线“crv”的不同，这个值的长度也会有区别；另外，推荐使用的散列算法也会随着“crv”的变化有所不同：

crv	x/y 的字节长度	散列算法
P-256	32	SHA-256
P-384	48	SHA-384
P-521	66	SHA-512

> 注意 `P-521` 对应的是 `SHA-512`，不是 `SHA-521`（不存在 521 位的散列算法 😂）

同样，使用的曲线也决定了签名的长度。在使用 ECDSA 对数据签名时，通过椭圆曲线计算得到 r 和 s 两个值。这两个值的字节长度也应该符合上表。

> 细心的同学可能会发现上面的 ECDSA 密钥中 “y” 的值转换为 hex 表示后是 33 个字节：
>
> ```
> 00 90 67 b9 0e 04 88 c9 c2 a9 f3 0f 5a 26 6a 07 84
> 1d 6c 07 74 13 ba 07 e7 45 69 b9 9d 4f d3 ce c6
> ```
>
> 我们知道，在密钥中 “x” 和 “y” 都是大的整数，但是在某些安全框架的实现 (比如一些版本的 OpenSSL) 中，使用的会是普通的整数类型 (Int)，而非无符号整数 (UInt)。而如果一个数字首 bit 为 1 的话，在有符号的整数系统中会被认为是负数。在这里，”y” 原本第一个 byte 其实是 `0x90` (bit 表示是 0b_1001_0000)，首 bit 为 1，为了避免被误认为负数，有的实现会在前面添加 `0x00`。但是实际上把这样一个 33 byte 的值作为 “y” 放在 JWK 中，是不符合标准的。如果你遇到了这种情况，可以和负责服务器的小伙伴商量一下让他先处理一下，给你正确的 key。当然，你也可以自己在客户端检查和处理长度不符合预期的问题，以增强本地代码的健壮性。
>
> 在这个例子中，如果服务器在生成 JWK 时就帮我们处理了 `0x00` 的问题的话，那么 “y” 的值应该是
>
> `kGe5DgSIycKp8w9aJmoHhB1sB3QTugfnRWm5nU_TzsY`
>
> 我们还会在后面看到更多的处理 `0x00` 添加或删除的情况，对于首字节是 `0x80` (`0b_1000_0000`) 或者以上的值，我们可能都需要考虑具体实现是接受 Int 还是 UInt 的问题。

JWA

JWA (JSON Web Algorithms) 定义的就是在 JWT 和 JWK 中涉及的算法了，它为每种算法定义了具体可能存在哪些参数，和参数的表示规则。比如上面 JWK 例子中的 “n”，”e”，”x”，”y”，”crv” 都是在 JWA 标准中定义的。它为如何使用 JWK，如何验证 JWT 提供支持和指导。

除了 RSA 和 ECDSA 以外，JWA 里还定义了 AES 相关的加密算法，不过这部分内容和 JWS 没什么关系。另外，在签名算法定义的后面，也附带了如果使用签名和如何进行验证的简单说明。我们在之后会对 JOSE 中的签名和验证过程进行更详细的解释。

小结

本文简述了 JWT 和 JOSE 的相关基础概念。您现在对 JWT 是什么，JOSE 有哪些组成部分，以及它们大概长什么样有一定了解。

你可以访问 JWT.io 来实际试试看创建和验证一个 JWT 的过程。如果你想要更深入了解 JWT 的内容和定义的话，JWT.io 还提供了免费的 JWT Handbook，里面有更详细的介绍。我们在系列文章的最后还会对 JWT 的应用场景，适用范围和存在的风险进行补充说明。

系列文章后面两篇，会分别针对 JOSE 中的签名和验证过程)以及作为 iOS 开发者如何使用 Security.frame 来处理 JOSE 相关的概念实践进行更详细的说明。