JWT

介绍

为什么使用JWT

Q：为什么要使用 `Json Web Token`？ A：`Json Web Token` 简称 `JWT`

顾名思义，可以知道是它是用于身份认证的

Q：那么为什么要有身份认证？ A：我们知道`HTTP`是无状态的，打个比方：

有状态：

A：你今天中午吃的啥？ B：吃的大盘鸡。 （访问了“大盘鸡”） A：味道怎么样呀？ B：还不错，挺好吃的。（知道访问的具体情况如何）

无状态：

A：你今天中午吃的啥？ B：吃的大盘鸡。 （访问了“大盘鸡”） A：味道怎么样呀？ B：？？？啊？啥？啥味道怎么样？ （不知道访问的具体情况或者说没有“记住”）

怎么让HTTP记住曾经发生的事情？

这里的选择一般为：**cookie,session,jwt**

对于一般的cookie，如果我们的加密措施不当，很容易造成信息泄露，甚至信息伪造，这肯定不是我们期望的。

对于`session`：

客户端在服务端登陆成功之后，服务端会生成一个sessionID，返回给客户端，客户端将`sessionID`保存到`cookie`中，例如`phpsessid`，再次发起请求的时候，携带`cookie`中的`sessionID`到服务端，服务端会缓存该`session`（会话），当客户端请求到来的时候，服务端就知道是哪个用户的请求，并将处理的结果返回给客户端，完成通信。 但是这样的机制会存在一些问题：

1. **session保存在服务端，当客户访问量增加时，服务端就需要存储大量的session会话，对服务器有很大的考验**；
2. **当服务端为集群时，用户登陆其中一台服务器，会将session保存到该服务器的内存中，但是当用户的访问到其他服务器时，会无法访问**，通常采用缓存一致性技术来保证可以共享，或者采用第三方缓存来保存session，不方便。

于是就到了 `JWT` 出场的时候：

在身份验证中，当用户使用他们的凭证成功登录时，JSON Web Token 将被返回并且必须保存在本地（通常在本地存储中，但也可以使用 Cookie），而不是在传统方法中创建会话服务器并返回一个 cookie。

无论何时用户想要访问受保护的路由或资源，用户代理都应使用承载方案发送`JWT`，通常在授权`header`中。`header`的内容应该如下所示：

```
Authorization: Bearer <token>
```

这是一种无状态身份验证机制，因为用户状态永远不会保存在服务器内存中。服务器受保护的路由将在授权头中检查有效的JWT，如果存在，则允许用户访问受保护的资源。由于JWT是独立的，所有必要的信息都在那里，减少了多次查询数据库的需求。

这使我们可以完全依赖无状态的数据API，无论哪些域正在为API提供服务，因此跨源资源共享（CORS）不会成为问题，因为它不使用Cookie。

Json Web Token 结构

我们随便挑一个 `JWT` 看看它长什么样子：

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiYWRtaW5za3kiLCJwcml2Ijoib3RoZXIif
Q.AoTc1q2NAErgqk6EeTK4MGH7cANVVF9XTy0wLv8HpgUfNcdM-
etmv0Y9XmOuygF_ymV1rF6XQZzLrtkFqdMdP0NaZnTOYH35Yevaudx9bVpu9JHG4qeXo-
0TXBcpaPmBaM0V0GxyZRNIS2KwRkNaxAQDQnyTN-Yi3w8OVpJYBiI
```

base64 解码：

```
{"alg":"RS256","typ":"JWT"}{"name":"adminsky","priv":"other"}乱码
```

不难看出，`jwt`解码后分为3个部分，由三个点（.）分隔

分别为：

```
Header
Payload
Signature
```

Header

通常由两部分组成：令牌的类型，即 JWT 和正在使用的散列算法，如 `HMAC SHA256` 或 `RSA`。

正如json所显示

```
{
    "alg":"RS256",
    "typ":"JWT"
}
```

`alg` 为算法的缩写，`typ` 为类型的缩写

然后，这个JSON被Base64编码，形成 JSON Web Token 的第一部分。

Payload

令牌的第二部分是包含声明的有效负载。声明是关于实体（通常是用户）和其他元数据的声明。

这里是用户随意定义的数据

例如上面的举例

```
{
    "name":"adminsky",
    "priv":"other"
}
```

然后将有效载荷进行Base64编码以形成JSON Web Token的第二部分。

但是需要注意:对于已签名的令牌，尽管此信息受到篡改保护，但任何人都可以阅读。除非加密，否则不要将秘密信息放在JWT的有效内容或标题元素中。

Signature

要创建签名部分，必须采用header，payload，密钥

然后利用header中指定算法进行签名

例如HS256(HMAC SHA256),签名的构成为:

```
HMACSHA256(
  base64Encode(header) + "." +
  base64Encode(payload),
  secret)
```

然后将这部分base64编码形成JSON Web Token第三部分

Json Web Token攻击

既然JWT作为一种身份验证的手段，那么必然存在伪造身份的恶意攻击，那么我们下面探讨一下常见的JWT攻击手段

敏感信息泄露

由于Header和Payload部分是使用可逆base64方法编码的，因此任何能够看到令牌的人都可以读取数据。要读取内容，只需要将每个部分传递给base64解码函数。比如前面介绍时所举例子。

算法修改攻击（密钥混淆攻击）

我们知道JWT的header部分中，有签名算法标识alg。而alg是用于签名算法的选择，最后保证用户的数据不被篡改。但是在数据处理不正确的情况下，可能存在alg的恶意篡改，例如：

由于网站的不严谨，我们拿到了泄露的公钥pubkey。我们知道如果签名算法为RS256，那么会选择用私钥进行签名，用公钥进行解密验证。

假设我们只拿到了公钥，且公钥模数极大，不可被分解，那么如何进行攻击呢？

没有私钥我们是几乎不可能在RS256的情况下篡改数据的，因为第三部分签名需要私钥，所以我们可以尝试将RS256改为HS256，此时即非对称密码变为对称加密。只有非对称密码存在公私钥问题，而对称加密只有一个key。

此时以pubkey作为key对数据进行篡改，则会非常简单，而如果后端的验证也是根据header的alg选择算法，那么显然正中下怀。

举例如下：

利用泄露的公钥伪造HS256算法签名。

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJsb2dpbiI6InRpY2FycGkifQ.I3G9aRHfunXlZV2lyJ
vWkZO0I_A_OiaAAQakU_kjkJM
```

base64解码：

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

后端代码接收到后会使用RSA公钥+HS256算法进行签名验证。 如何抵御这种攻击？JWT配置应该只允许使用HMAC算法或公钥算法，决不能同时使用这两种算法。

特殊情况

JWT支持将算法设定为None。如果alg字段设为None，那么签名会被置空，这样任何token都是有效的。 **设定该功能的最初目的是为了方便调试。但是，若不在生产环境中关闭该功能，攻击者可以通过将alg字段设置为None来伪造他们想要的任何token，接着便可以使用伪造的token冒充任意用户登陆网站。**

举例如下：

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiYWRtaW4iLCJhY3Rpb24iOiJ1cGxvYWQif
Q.y2k9SJDRU81ybXm-anxpD2p1N-rKekDJtJGKGJlemjY
```

base64解码：

```
{"alg":"HS256","typ":"JWT"}
{"user":"admin","action":"upload"}
```

设置 alg: none不带签名，生成Token:

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiYWRtaW4iLCJhY3Rpb24iOiJ1cGxvYWQifQ
.
```

base64解码

```
{"typ":"JWT","alg":"none"}
{"user":"admin","action":"upload"}
```

然后查看页面是否仍然返回有效？如果页面返回有效，那么说明存在漏洞。 如何抵御这种攻击？ JWT配置应该指定所需的签名算法，不可指定"none"。

无效签名

当用户端提交请求给应用程序，**服务端可能没有对token签名进行校验**，这样，攻击者便可以通过提供无效签名简单地绕过安全机制。

示例：

一个很好的例子是网站上的"个人资料"页面，因为我们只有在被授权通过有效的JWT进行访问时才能访问此页面，我们将重放请求并寻找响应的变化以发现问题。

原请求：

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjoidGVzdCIsImFjdGlvbiI6InByb2ZpbGUif
Q.FjnAvQxzRKcahlw2EPd9o7teqX-fQSt7MZhT84hj7mU
```

user 字段改为 admin，重新生成新 token：

修改之后：

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjoiYWRtaW4iLCJhY3Rpb24iOiJwcm9maWxlI
n0._LRRXAfXtnagdyB1uRk-7CfkK1RESGwxqQCdwCNSPaI
```

```
{"typ": "JWT", "alg": "HS256"}.
{"user": "admin","action": "profile"}.
[新的签名]
```

将重新生成的Token发给服务端效验，如访问页面正常，则说明漏洞存在。

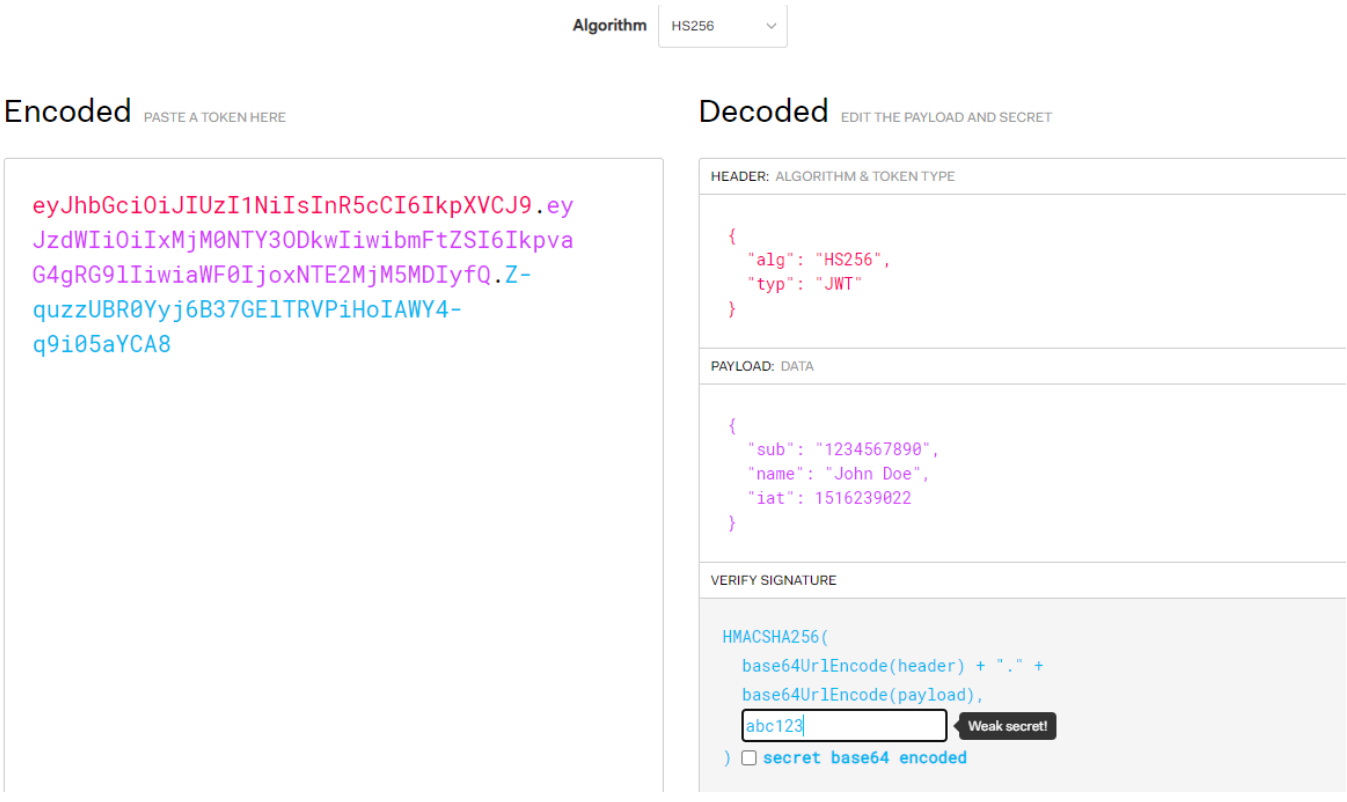
暴力破解密钥

HMAC签名密钥（例如HS256|HS384|HS512）使用对称加密，这意味着对令牌进行签名的密钥也用于对其进行验证。由于签名验证是一个自包含的过程，因此可以测试令牌本身的有效密钥，而不必将其发送回应用程序进行验证。

因此，HMAC JWT破解是离线的，通过JWT破解工具，可以快速检查已知的泄漏密码列表或默认密码。

```
python jwt_tool.py
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyIjoiYWRtaW4iLCJhY3Rpb24iOiJ1cGxvYWQif
Q.7ZbwdZXwfjm575fHGukkE09O8-eFY4bx-fEEBUK3XUE -C -d 1.txt
```

获得密钥，伪造任意用户的Token：



如果可以破解HMAC密钥，则可以伪造令牌中的任何内容，这个漏洞将会给系统带来非常严重的后果。

操纵KID

KID代表 密钥序号（Key ID）。它是JWT头部的一个可选字段，开发人员可以用它标识认证token的某一密钥。KID参数的正确用法如下所示：

```
{
    "alg": "HS256",
    "typ": "JWT",
    "kid": "1"          //使用密钥1验证token
}
```

由于此字段是由用户控制的，因此攻击者可能会操纵它并导致危险的后果。

目录遍历

由于KID通常用于从文件系统中检索密钥文件，因此，如果在使用前不清理KID，文件系统可能会遭到目录遍历攻击。这样，攻击者便能够在文件系统中指定任意文件作为认证的密钥。

```
"kid": "../../public/css/main.css"    //使用公共文件main.css验证token
```

例如，攻击者可以强行设定应用程序使用公开可用文件作为密钥，并用该文件给HMAC加密的token签名。

SQL注入

KID也可以用于在数据库中检索密钥。在该情况下，攻击者很可能会利用SQL注入来绕过JWT安全机制。 如果可以在KID参数上进行SQL注入，攻击者便能使用该注入返回任意值。

```
"kid":"aaaaaaa' UNION SELECT 'key';--"  //使用字符串"key"验证token
```

上面这个注入会导致应用程序返回字符串 key（因为数据库中不存在名为 aaaaaaa 的密钥）。然后使用字符串key作为密钥来认证token。

命令注入

对kid参数过滤不严也可能会出现命令注入问题，但是利用条件比较苛刻。如果服务器后端使用的是Ruby，在读取密钥文件时使用了open函数，通过构造参数就可能造成命令注入。

```
/path/to/key_file | whoami
```

类似情况还有很多，这只是其中一个例子。理论上，每当应用程序将未审查的头部文件参数传递给类似 system () ， exec () 的函数时，都会产生此种漏洞。

操纵头部参数

除KID外，JWT标准还能让开发人员通过URL指定密钥。

JKU头部参数

JKU全称是JWKSet URL，它是头部的一个可选字段，用于指定链接到一组加密token密钥的URL。若允许使用该字段且不设置限定条件，攻击者就能托管自己的密钥文件，并指定应用程序，用它来认证token。

```
jku URL->包含JWK集的文件->用于验证令牌的JWK
```

JWK头部参数

头部可选参数JWK（JSON Web Key）使得攻击者能将认证的密钥直接嵌入token中。

操纵X5U，X5C URL

同JKU或JWK头部类似，x5u和x5c头部参数允许攻击者用于验证Token的公钥证书或证书链。x5u以URI形式指定信息，而x5c允许将证书值嵌入token中。